
The Python Library Reference

Versión 3.8.20

**Guido van Rossum
and the Python development team**

septiembre 08, 2024

**Python Software Foundation
Email: docs@python.org**

1	Introducción	3
1.1	Notas sobre la disponibilidad	4
2	Funciones Built-in	5
3	Constantes incorporadas	27
3.1	Constantes agregadas por el módulo <code>site</code>	28
4	Tipos Integrados	29
4.1	Evaluar como valor verdadero/falso	29
4.2	Operaciones booleanas — <code>and</code> , <code>or</code> , <code>not</code>	30
4.3	Comparaciones	30
4.4	Tipos numéricos — <code>int</code> , <code>float</code> , <code>complex</code>	31
4.5	Tipos de iteradores	37
4.6	Tipos secuencia — <code>list</code> , <code>tuple</code> , <code>range</code>	37
4.7	Cadenas de caracteres — <code>str</code>	44
4.8	Tipos de secuencias binarias — <code>bytes</code> , <code>bytearray</code> y <code>memoryview</code>	54
4.9	Conjuntos — <code>set</code> , <code>frozenset</code>	75
4.10	Tipos Mapa — <code>dict</code>	77
4.11	Tipos Gestores de Contexto	82
4.12	Otros tipos predefinidos	83
4.13	Atributos especiales	85
4.14	Integer string conversion length limitation	86
5	Excepciones incorporadas	89
5.1	Clases base	90
5.2	Excepciones específicas	90
5.3	Advertencias	96
5.4	Jerarquía de excepción	97
6	Servicios de procesamiento de texto	99
6.1	<code>string</code> — Operaciones comunes de cadena de caracteres	99
6.2	<code>re</code> — Operaciones con expresiones regulares	110
6.3	<code>difflib</code> — Funciones auxiliares para calcular deltas	129
6.4	<code>textwrap</code> — Envoltura y relleno de texto	140
6.5	<code>unicodedata</code> — Base de datos Unicode	144
6.6	<code>stringprep</code> — Preparación de cadenas de Internet	145
6.7	<code>readline</code> — Interfaz <code>readline</code> de GNU	147
6.8	<code>rlcompleter</code> — Función de completado para GNU <code>readline</code>	151
7	Servicios de datos binarios	153

7.1	<code>struct</code> — Interpreta bytes como paquetes de datos binarios	153
7.2	<code>codecs</code> — Registro de códec y clases base	158
8	Tipos de datos	177
8.1	<code>datetime</code> — Tipos básicos de fecha y hora	177
8.2	<code>calendar</code> — Funciones generales relacionadas con el calendario	212
8.3	<code>collections</code> — Tipos de datos contenedor	217
8.4	<code>collections.abc</code> — Clases Base Abstractas para Contenedores	233
8.5	<code>heapq</code> — Algoritmo de colas montículos (<i>heap</i>)	237
8.6	<code>bisect</code> — Algoritmo de bisección de arreglos	241
8.7	<code>array</code> — Arreglos eficientes de valores numéricos	243
8.8	<code>weakref</code> — Referencias débiles	246
8.9	<code>types</code> — Creación de tipos dinámicos y nombres para tipos integrados	253
8.10	<code>copy</code> — Operaciones de copia superficial y profunda	258
8.11	<code>pprint</code> — Impresión bonita de datos	259
8.12	<code>reprlib</code> — Implementación <code>repr()</code> alternativa	264
8.13	<code>enum</code> — Soporte para enumeraciones	266
9	Módulos numéricos y matemáticos	285
9.1	<code>numbers</code> — Clase base abstracta numérica	285
9.2	<code>math</code> — Funciones matemáticas	288
9.3	<code>cmath</code> — Función matemática para números complejos	295
9.4	<code>decimal</code> — Decimal fixed point and floating point arithmetic	298
9.5	<code>fractions</code> — Números racionales	324
9.6	<code>random</code> — Generar números pseudoaleatorios	326
9.7	<code>statistics</code> — Funciones de estadística matemática	332
10	Módulos de programación funcional	343
10.1	<code>itertools</code> — Funciones que crean iteradores para bucles eficientes	343
10.2	<code>functools</code> — Funciones de orden superior y operaciones sobre objetos invocables	357
10.3	<code>operator</code> — Operadores estándar como funciones	365
11	Acceso a archivos y directorios	373
11.1	<code>pathlib</code> — Object-oriented filesystem paths	373
11.2	<code>os.path</code> — Manipulaciones comunes de nombre de ruta	389
11.3	<code>fileinput</code> — Iterar sobre líneas de múltiples flujos de entrada	394
11.4	<code>stat</code> — Interpretación de los resultados de <code>stat()</code>	396
11.5	<code>filecmp</code> — Comparaciones de Archivo y Directorio	401
11.6	<code>tempfile</code> — Generar archivos y directorios temporales	403
11.7	<code>glob</code> — Expansión del patrón de nombres de ruta de estilo Unix	407
11.8	<code>fnmatch</code> — Coincidencia de patrones de nombre de archivos Unix	409
11.9	<code>linecache</code> — Acceso aleatorio a líneas de texto	410
11.10	<code>shutil</code> — Operaciones de archivos de alto nivel	410
12	Persistencia de datos	421
12.1	<code>pickle</code> — Serialización de objetos Python	421
12.2	<code>copyreg</code> — Registrar funciones de soporte de <code>pickle</code>	437
12.3	<code>shelve</code> — Persistencia de objetos de Python	438
12.4	<code>marshal</code> — Serialización interna de objetos Python	441
12.5	<code>dbm</code> — Interfaces para «bases de datos» de Unix	442
12.6	<code>sqlite3</code> — DB-API 2.0 interfaz para bases de datos SQLite	446
13	Compresión de datos y archivado	469
13.1	<code>zlib</code> — Compresión compatible con gzip	469
13.2	<code>gzip</code> — Soporte para archivos gzip	473
13.3	<code>bz2</code> — Soporte para compresión bzip2	476
13.4	<code>lzma</code> — Compresión utilizando el algoritmo LZMA	479
13.5	<code>zipfile</code> — Trabajar con archivos ZIP	485
13.6	<code>tarfile</code> — Leer y escribir archivos tar	495

14	Formatos de archivo	511
14.1	csv — Lectura y escritura de archivos CSV	511
14.2	configparser — <i>Parser</i> para archivos de configuración	518
14.3	netrc — procesado del fichero netrc	535
14.4	xdrlib — Codificar y decodificar datos XDR	536
14.5	plistlib — Genera y analiza archivos .plist de Mac OS X	539
15	Servicios Criptográficos	543
15.1	hashlib — Hashes seguros y resúmenes de mensajes	543
15.2	hmac — <i>Hash</i> con clave para autenticación de mensajes	553
15.3	secrets — Genera números aleatorios seguros para trabajar con secretos criptográficos	555
16	Servicios genéricos del sistema operativo	559
16.1	os — Interfaces misceláneas del sistema operativo	559
16.2	io — Herramientas principales para trabajar con <i>streams</i>	610
16.3	time — Tiempo de acceso y conversiones	622
16.4	argparse — Analizador sintáctico (<i>Parser</i>) para las opciones, argumentos y sub-comandos de la línea de comandos	632
16.5	getopt — Analizador de estilo C para opciones de línea de comando	663
16.6	logging — Logging facility for Python	665
16.7	logging.config — Configuración de registro	681
16.8	logging.handlers — Gestores de <i>logging</i>	692
16.9	getpass — Entrada de contraseña portátil	704
16.10	curses — Manejo de terminales para pantallas de celdas de caracteres	705
16.11	curses.textpad — Widget de entrada de texto para programas de curses	722
16.12	curses.ascii — Utilidades para los caracteres ASCII	724
16.13	curses.panel — Una extensión de pila de panel para curses	726
16.14	platform — Acceso a los datos identificativos de la plataforma subyacente	727
16.15	errno — Símbolos estándar del sistema errno	730
16.16	ctypes — Una biblioteca de funciones foráneas para Python	736
17	Ejecución concurrente	769
17.1	threading — Paralelismo basado en hilos	769
17.2	multiprocessing — Paralelismo basado en procesos	782
17.3	multiprocessing.shared_memory — Proporciona memoria compartida para acceso directo a través de procesos	825
17.4	El paquete concurrent	830
17.5	concurrent.futures — Lanzamiento de tareas paralelas	830
17.6	subprocess — Gestión de subprocesos	836
17.7	sched — Eventos del planificador	854
17.8	queue — clase de cola sincronizada	855
17.9	contextvars — Variables de Contexto	858
17.10	_thread — API de bajo nivel para manejo de hilos	862
17.11	_dummy_thread — Reemplazo directo para el módulo _thread	864
17.12	dummy_threading — Reemplazo directo para el módulo threading	864
18	Comunicación en redes y entre procesos	865
18.1	asyncio — E/S Asíncrona	865
18.2	socket — Low-level networking interface	950
18.3	ssl — TLS/SSL wrapper for socket objects	973
18.4	select — Esperando la finalización de E/S	1007
18.5	selectors — Multiplexación de E/S de alto nivel	1014
18.6	asyncore — controlador de socket asincrónico	1017
18.7	asynchat — Asynchronous socket command/response handler	1021
18.8	signal — Set handlers for asynchronous events	1024
18.9	mmap — Soporte de archivos mapeados en memoria	1031
19	Manejo de Datos de Internet	1037
19.1	email — Un paquete de manejo de correo electrónico y MIME	1037

19.2	<code>json</code> — Codificador y decodificador JSON	1096
19.3	<code>mailcap</code> — Manejo de archivos Mailcap	1105
19.4	<code>mailbox</code> — Manipular buzones de correo en varios formatos	1106
19.5	<code>mimetypes</code> — Mapea nombres de archivo a tipos MIME	1124
19.6	<code>base64</code> — Codificaciones de datos Base16, Base32, Base64, y Base85	1127
19.7	<code>binhex</code> — Codificar y decodificar archivos binhex4	1130
19.8	<code>binascii</code> — Convertir entre binario y ASCII	1130
19.9	<code>quopri</code> — Codificar y decodificar datos MIME imprimibles entre comillas	1133
19.10	<code>uu</code> — Codifica y decodifica archivos UUEncode	1133
20	Herramientas Para Procesar Formatos de Marcado Estructurado	1135
20.1	<code>html</code> — Compatibilidad con el Lenguaje de marcado de hipertexto	1135
20.2	<code>html.parser</code> — Analizador simple de HTML y XHTML	1136
20.3	<code>html.entities</code> — Definiciones de entidades generales HTML	1140
20.4	Módulos de procesamiento XML	1140
20.5	<code>xml.etree.ElementTree</code> — The ElementTree XML API	1142
20.6	<code>xml.dom</code> — El API del Modelo de Objetos del Documento	1161
20.7	<code>xml.dom.minidom</code> — Implementación mínima del DOM	1171
20.8	<code>xml.dom.pulldom</code> — Soporte para la construcción parcial de árboles DOM	1175
20.9	<code>XML.sax</code> — Soporte para analizadores SAX2	1177
20.10	<code>xml.sax.handler</code> — Base clases for SAX handlers	1179
20.11	<code>xml.sax.saxutils</code> — Utilidades SAX	1184
20.12	<code>xml.sax.xmlreader</code> — Interfaz para analizadores XML	1185
20.13	<code>xml.parsers.expat</code> — Análisis rápido XML usando Expat	1189
21	Protocolos y soporte de Internet	1199
21.1	<code>webbrowser</code> — Cómodo controlador de navegador web	1199
21.2	<code>cgi</code> — Soporte de Interfaz de Entrada Común (CGI)	1202
21.3	<code>cgitb</code> — Administrador <i>traceback</i> para scripts CGI	1208
21.4	<code>wsgiref</code> — Utilidades WSGI e implementación de referencia	1209
21.5	<code>urllib</code> — URL módulos de manipulación	1219
21.6	<code>urllib.request</code> — Biblioteca extensible para abrir URLs	1220
21.7	<code>urllib.response</code> — Clases de respuesta usadas por <code>urllib</code>	1238
21.8	<code>urllib.parse</code> — Analiza URL en componentes	1238
21.9	<code>urllib.error</code> — Clases de excepción lanzadas por <code>urllib.request</code>	1246
21.10	<code>urllib.robotparser</code> — Analizador para robots.txt	1247
21.11	<code>http</code> — Módulos HTTP	1248
21.12	<code>http.client</code> — Cliente de protocolo HTTP	1250
21.13	<code>ftplib</code> — cliente de protocolo FTP	1257
21.14	<code>poplib</code> — Cliente de protocolo POP3	1262
21.15	<code>imaplib</code> — Protocolo del cliente IMAP4	1265
21.16	<code>nntplib</code> — Protocolo de cliente NNTP	1271
21.17	<code>smtplib</code> — Cliente de protocolo SMTP	1278
21.18	<code>smtpd</code> — Servidor SMTP	1285
21.19	<code>telnetlib</code> — cliente Telnet	1288
21.20	<code>uuid</code> — objetos UUID según RFC 4122	1291
21.21	<code>socketserver</code> — Un framework para servidores de red	1294
21.22	<code>http.server</code> — Servidores HTTP	1302
21.23	<code>http.cookies</code> — Gestión del estado HTTP	1308
21.24	<code>http.cookiejar</code> — Cookie handling for HTTP clients	1311
21.25	<code>xmlrpc</code> — Módulos XMLRPC para cliente y servidor	1319
21.26	<code>xmlrpc.client</code> — acceso cliente XML-RPC	1320
21.27	<code>xmlrpc.server</code> — Servidores básicos XML-RPC	1327
21.28	<code>ipaddress</code> — Biblioteca de manipulación IPv4/IPv6	1333
22	Servicios Multimedia	1347
22.1	<code>audioop</code> — Manipula datos de audio sin procesar	1347
22.2	<code>aifc</code> — Lee y escribe archivos AIFF y AIFC	1350
22.3	<code>sunau</code> — Lectura y escritura de ficheros Sun AU	1353

22.4	<code>wave</code> — Leer y escribir archivos WAV	1355
22.5	<code>chunk</code> — Lee los datos de los trozos de IFF	1358
22.6	<code>colorsys</code> — Conversiones entre sistemas de color	1359
22.7	<code>:mod:"imghdr"</code> — Determinar el tipo de imagen	1360
22.8	<code>sndhdr</code> — Determinar el tipo de archivo de sonido	1361
22.9	<code>ossaudiodev</code> — Acceso a dispositivos de audio compatibles con OSS	1361
23	Internacionalización	1367
23.1	<code>gettext</code> — Servicios de internacionalización multilingües	1367
23.2	<code>locale</code> — Servicios de internacionalización	1376
24	Frameworks de programa	1385
24.1	<code>turtle</code> — Gráficos con <i>Turtle</i>	1385
24.2	<code>cmd</code> — Soporte para intérpretes orientados a línea de comandos	1418
24.3	<code>shlex</code> — Análisis léxico simple	1423
25	Interfaces gráficas de usuario con Tk	1429
25.1	<code>tkinter</code> — Interface de Python para Tcl/Tk	1429
25.2	<code>tkinter.ttk</code> — Tk widgets temáticos	1440
25.3	<code>tkinter.tix</code> — Ampliación de widgets para Tk	1459
25.4	<code>tkinter.scrolledtext</code> — Widget de texto desplazado	1463
25.5	<code>IDLE</code>	1464
25.6	Otros paquetes de interfaz gráfica de usuario	1475
26	Herramientas de desarrollo	1477
26.1	<code>typing</code> — Soporte para <i>type hints</i>	1477
26.2	<code>pydoc</code> — Generador de documentación y Sistema de ayuda en línea	1496
26.3	<code>doctest</code> — Test interactive Python examples	1497
26.4	<code>unittest</code> — Unit testing framework	1519
26.5	<code>unittest.mock</code> — Biblioteca de objetos simulados	1548
26.6	<code>unittest.mock</code> — getting started	1587
26.7	<code>2to3</code> - Traducción de código Python 2 a 3	1606
26.8	<code>test</code> — Paquete de pruebas de regresión para Python	1612
26.9	<code>test.support</code> — Utilidades para el conjunto de pruebas de Python	1614
26.10	<code>test.support.script_helper</code> — Utilidades para las pruebas de ejecución de Python	1627
27	Depuración y perfilado	1629
27.1	Tabla de auditoría de eventos	1629
27.2	<code>bdb</code> — Framework de depuración	1633
27.3	<code>faulthandler</code> — Volcar el rastreo de Python	1638
27.4	<code>pdb</code> — El Depurador de Python	1640
27.5	Los perfiladores de Python	1646
27.6	<code>timeit</code> — Mide el tiempo de ejecución de pequeños fragmentos de código	1654
27.7	<code>trace</code> — Rastrear la ejecución de la declaración de Python	1659
27.8	<code>tracemalloc</code> — Rastrea la asignación de memoria	1662
28	Software Packaging and Distribution	1673
28.1	<code>distutils</code> — Creación e instalación de módulos Python	1673
28.2	<code>ensurepip</code> — Ejecutando el instalador <code>pip</code>	1674
28.3	<code>venv</code> — Creación de entornos virtuales	1675
28.4	<code>zipapp</code> — Gestiona archivadores zip ejecutables de Python	1684
29	Servicios en tiempo de ejecución de Python	1691
29.1	<code>sys</code> — Parámetros y funciones específicos del sistema	1691
29.2	<code>sysconfig</code> — Proporciona acceso a la información de configuración de Python	1710
29.3	<code>builtins</code> — Objetos incorporados	1714
29.4	<code>__main__</code> — Entorno de script del nivel superior	1714
29.5	<code>warnings</code> — Control de advertencias	1715
29.6	<code>dataclasses</code> — Clases de datos	1721

29.7	<code>contextlib</code> — Utilidades para declaraciones de contexto <code>with</code>	1729
29.8	<code>abc</code> — Clases de Base Abstracta	1741
29.9	<code>atexit</code> — Gestores de Salida	1746
29.10	<code>traceback</code> — Imprimir o recuperar un seguimiento de pila	1747
29.11	<code>__future__</code> — Definiciones de declaraciones futuras	1753
29.12	<code>gc</code> — Interfaz del recolector de basura	1754
29.13	<code>inspect</code> — Inspeccionar objetos vivos	1758
29.14	<code>site</code> — Enlace (<i>hook</i>) de configuración específico del sitio	1773
30	Intérpretes de Python personalizados	1777
30.1	<code>code</code> — Clases básicas de intérpretes	1777
30.2	<code>codeop</code> — Compile Python code	1779
31	Importando módulos	1781
31.1	<code>zipimport</code> — Importar módulos desde archivos zip	1781
31.2	<code>pkgutil</code> — Utilidad de extensión de paquete	1783
31.3	<code>modulefinder</code> — Buscar módulos usados por un script	1786
31.4	<code>runpy</code> — Localización y ejecución de módulos <i>Python</i>	1787
31.5	<code>importlib</code> — The implementation of <code>import</code>	1789
31.6	Using <code>importlib.metadata</code>	1808
32	Servicios del lenguaje Python	1813
32.1	<code>parser</code> — Acceder a árboles de análisis sintáctico de Python	1813
32.2	<code>ast</code> — Árboles de sintaxis abstracta	1817
32.3	<code>symtable</code> — Acceso a la tabla de símbolos del compilador	1824
32.4	<code>symbol</code> — Constantes utilizadas con árboles de análisis de Python	1826
32.5	<code>token</code> — Constantes usadas con árboles de sintaxis de Python	1826
32.6	<code>keyword</code> — Pruebas para palabras clave en Python	1830
32.7	<code>tokenize</code> — Conversor a tokens para código Python	1830
32.8	<code>tabnanny</code> — Detección de indentación ambigua	1834
32.9	<code>pyclbr</code> — Soporte para navegador de módulos Python	1835
32.10	<code>py_compile</code> — Compila archivos fuente Python	1836
32.11	<code>compileall</code> — Bibliotecas de Python de compilación de bytes	1838
32.12	<code>dis</code> — Desensamblador para bytecode de Python	1841
32.13	<code>pickletools</code> — Herramientas para desarrolladores pickle	1855
33	Servicios varios	1857
33.1	<code>formatter</code> — Formateo de salida genérica	1857
34	Servicios Específicos para MS Windows	1861
34.1	<code>msilib</code> — Leer y escribir archivos <i>Microsoft Installer</i>	1861
34.2	<code>msvcrt</code> — Rutinas útiles del entorno de ejecución MS VC++	1867
34.3	<code>winreg</code> — Acceso al registro de Windows	1868
34.4	<code>:mod:"winsound"</code> — Interfaz de reproducción de sonido para Windows	1877
35	Servicios específicos de Unix	1881
35.1	<code>posix</code> — Las llamadas más comunes al sistema POSIX	1881
35.2	<code>pwd</code> — La base de datos de contraseñas	1882
35.3	<code>spwd</code> — La base de datos de contraseñas ocultas	1883
35.4	<code>grp</code> — La base de datos de grupo	1884
35.5	<code>crypt</code> — Función para verificar contraseñas Unix	1884
35.6	<code>termios</code> — Control tty estilo POSIX	1886
35.7	<code>tty</code> — Funciones de control de terminal	1888
35.8	<code>pty</code> — Utilidades para Pseudo-terminal	1888
35.9	<code>fcntl</code> — Las llamadas a sistema <code>fcntl</code> y <code>ioctl</code>	1889
35.10	<code>pipes</code> — Interfaz para tuberías de shell	1892
35.11	<code>resource</code> — Información sobre el uso de recursos	1893
35.12	<code>nis</code> — Interfaz a Sun's NIS (Páginas amarillas)	1897
35.13	<code>syslog</code> — Rutinas de la biblioteca <code>syslog</code> de Unix	1897

36 Módulos Reemplazados	1901
36.1 <code>optparse</code> — Analizador sintáctico (parser) para opciones de línea de comandos	1901
36.2 <code>imp</code> — Acceda a <code>import</code> internamente	1928
37 Módulos no documentados	1933
37.1 Módulos específicos por plataforma	1933
A Glosario	1935
B Acerca de estos documentos	1949
B.1 Contribuidores de la documentación de Python	1949
C History and License	1951
C.1 History of the software	1951
C.2 Terms and conditions for accessing or otherwise using Python	1952
C.3 Licenses and Acknowledgements for Incorporated Software	1955
D Copyright	1967
Bibliografía	1969
Índice de Módulos Python	1971
Índice	1975

Aunque `reference-index` describe la sintaxis y semántica precisa del lenguaje Python, este manual de referencia de la biblioteca describe la biblioteca estándar que se distribuye con Python. También describe algunos componentes opcionales que son usualmente incluidos en las distribuciones de Python.

La biblioteca estándar de Python es muy amplia, y ofrece una gran cantidad de producciones como puede verse en la larga lista de contenidos. La biblioteca contiene módulos incorporados (escritos en C) que brindan acceso a las funcionalidades del sistema como entrada y salida de archivos que serían de otra forma inaccesibles para los programadores en Python, así como módulos escritos en Python que proveen soluciones estandarizadas para los diversos problemas que pueden ocurrir en el día a día en la programación. Algunos de éstos módulos están diseñados explícitamente para alentar y reforzar la portabilidad de los programas en Python abstrayendo especificidades de las plataformas para lograr APIs neutrales a la plataforma.

Los instaladores de Python para la plataforma Windows frecuentemente incluyen la biblioteca estándar completa y suelen también incluir muchos componentes adicionales. Para los sistemas operativos tipo Unix Python suele ser provisto como una colección de paquetes, así que puede requerirse usar las herramientas de empaquetado disponibles en los sistemas operativos para obtener algunos o todos los componentes opcionales.

Además de la biblioteca estándar, existe una colección creciente de varios miles de componentes (abarcando módulos o programas individuales, paquetes o *frameworks* completos de desarrollo de aplicaciones), disponibles en el [Python Package Index](#).

Introducción

La «biblioteca Python» contiene varios tipos de componentes diferentes.

Contiene tipos de datos que normalmente se considerarían parte del «núcleo» de un lenguaje, como números y listas. Para estos tipos, el núcleo del lenguaje Python define la forma de los literales y coloca algunas restricciones a su semántica, pero no define completamente la semántica. (Por otro lado, el núcleo del lenguaje sí define propiedades sintácticas como la ortografía y las prioridades de los operadores.)

La biblioteca también contiene funciones y excepciones incorporadas — objetos que pueden ser utilizados por todo código Python sin la necesidad de una declaración `import`. Algunas de ellos están definidas por el núcleo del lenguaje, pero muchos no son esenciales para la semántica del núcleo y sólo se describen aquí.

La mayor parte de la biblioteca, sin embargo, consiste en una colección de módulos. Hay muchas maneras de diseccionar esta colección. Algunos módulos están escritos en C y fueron incorporados en el intérprete de Python; otros están escritos en Python y se importan en código fuente. Algunos módulos proporcionan interfaces muy específicas de Python, como la impresión de un *stack trace*; otros proporcionan interfaces que son específicas para determinados sistemas operativos, como el acceso a hardware específico; otros proveen interfaces específicas para un dominio de aplicación concreto, como la World Wide Web. Algunos módulos están disponibles en todas las versiones y plataformas de Python; otros sólo están disponibles cuando el sistema subyacente los soporta o los requiere; otros solo están disponibles cuando se ha elegido una opción de configuración particular en el momento en que compiló e instaló Python.

Este manual está organizado «desde adentro hacia afuera:» primero describe las funciones integradas, los tipos de datos y las excepciones, y finalmente describe los módulos, agrupados en capítulos de módulos relacionados.

Esto significa que si comienza a leer este manual desde el principio, y salta al siguiente capítulo cuando se aburra, obtendrá una visión general razonable de los módulos y áreas de aplicación disponibles que son soportados por la biblioteca de Python. Por supuesto, no *ienes* que leerlo necesariamente como una novela — sino que también puedes navegar por la tabla de contenidos (al principio del manual), o buscar una función, módulo o término específico en el glosario (en la parte final del manual). Y por último, si disfruta aprender sobre diferentes temas de manera aleatoria, puede escoger un número de página al azar (ver módulo *random*) y leer una o dos secciones. Independientemente del orden en que lea las secciones de este manual, es útil comenzar con el capítulo *Funciones Built-in*, ya que el resto del manual asume la familiaridad con este material.

¡Que comience el espectáculo!

1.1 Notas sobre la disponibilidad

- Una nota de «Disponibilidad: Unix» significa que esta función se encuentra comúnmente en los sistemas Unix. Pero no hace ninguna afirmación sobre su existencia en un sistema operativo específico.
- Si no se indica por separado, todas las funciones con la leyenda «Disponibilidad: Unix» son compatibles con Mac OS X, que se basa en un núcleo de Unix.

Funciones Built-in

El intérprete de Python tiene una serie de funciones y tipos incluidos en él que están siempre disponibles. Están listados aquí en orden alfabético.

		Funciones Built-in		
<i>abs()</i>	<i>delattr()</i>	<i>hash()</i>	<i>memoryview()</i>	<i>set()</i>
<i>all()</i>	<i>dict()</i>	<i>help()</i>	<i>min()</i>	<i>setattr()</i>
<i>any()</i>	<i>dir()</i>	<i>hex()</i>	<i>next()</i>	<i>slice()</i>
<i>ascii()</i>	<i>divmod()</i>	<i>id()</i>	<i>object()</i>	<i>sorted()</i>
<i>bin()</i>	<i>enumerate()</i>	<i>input()</i>	<i>oct()</i>	<i>staticmethod()</i>
<i>bool()</i>	<i>eval()</i>	<i>int()</i>	<i>open()</i>	<i>str()</i>
<i>breakpoint()</i>	<i>exec()</i>	<i>isinstance()</i>	<i>ord()</i>	<i>sum()</i>
<i>bytearray()</i>	<i>filter()</i>	<i>issubclass()</i>	<i>pow()</i>	<i>super()</i>
<i>bytes()</i>	<i>float()</i>	<i>iter()</i>	<i>print()</i>	<i>tuple()</i>
<i>callable()</i>	<i>format()</i>	<i>len()</i>	<i>property()</i>	<i>type()</i>
<i>chr()</i>	<i>frozenset()</i>	<i>list()</i>	<i>range()</i>	<i>vars()</i>
<i>classmethod()</i>	<i>getattr()</i>	<i>locals()</i>	<i>repr()</i>	<i>zip()</i>
<i>compile()</i>	<i>globals()</i>	<i>map()</i>	<i>reversed()</i>	<i>__import__()</i>
<i>complex()</i>	<i>hasattr()</i>	<i>max()</i>	<i>round()</i>	

abs (*x*)

Retorna el valor absoluto de un número. El argumento puede ser un número entero o de punto flotante. Si el argumento es un número complejo, retorna su magnitud. Si *x* define un método `__abs__()`, `abs(x)` retorna `x.__abs__()`.

all (*iterable*)

Retorna `True` si todos los elementos del *iterable* son verdaderos (o si el iterable está vacío). Equivalente a:

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

any (*iterable*)

Retorna `True` si un elemento cualquiera del *iterable* es verdadero. Si el iterable está vacío, retorna `False`. Equivalente a:

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

ascii (*object*)

Como `repr()`, retorna una cadena que contiene una representación imprimible de un objeto, pero que escapa los caracteres no-ASCII que `repr()` retorna usando `\x`, `\u` or `\U`. Esto genera una cadena similar a la retornada por `repr()` en Python 2.

bin (*x*)

Convierte un número entero a una cadena binaria con prefijo «0b». El resultado es una expresión de Python válida. Si *x* no es un objeto de clase `int` en Python, tiene que definir un método `__index__()` que retorne un entero. Algunos ejemplos:

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

Según se desee o no el prefijo «0b», se puede usar uno u otro de las siguientes maneras.

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

Véase también `format()` para más información.

class bool (`[x]`)

Retorna un booleano, es decir, o bien `True` o `False`. *x* es convertido usando el estándar *truth testing procedure*. Si *x* es falso u omitido, retorna `False`; en caso contrario retorna `True`. La clase `bool` es una subclase de `int` (véase *Tipos numéricos — int, float, complex*). De ella no pueden derivarse más subclases. Sus únicas instancias son `False` y `True` (véase *Valores booleanos*).

Distinto en la versión 3.7: *x* es ahora un argumento solo de posición.

breakpoint (**args, **kws*)

Esta función te lleva al depurador en el sitio donde se produce la llamada. Específicamente, llama a `sys.breakpointhook()`, pasando *args* y *kws* directamente. Por defecto, `sys.breakpointhook()` llama a `pdb.set_trace()` sin esperar argumentos. En este caso, es puramente una función de conveniencia para evitar el importe explícito de `pdb` o tener que escribir tanto código para entrar al depurador. Sin embargo, `sys.breakpointhook()` puede ser configurado a otra función y `breakpoint()` llamará automáticamente a esta, permitiendo entrar al depurador elegido.

Lanza un *evento de auditoría* `builtins.breakpoint` con `breakpointhook` como argumento.

Nuevo en la versión 3.7.

class bytearray (`[source[, encoding[, errors]]]`)

Retorna un nuevo array de bytes. La clase `bytearray` es una secuencia mutable de enteros en el rango $0 \leq x < 256$. Tiene la mayoría de los métodos comunes en las secuencias mutables, descritos en *Tipos de secuencia mutables*, así como la mayoría de los métodos que la clase `bytes` tiene, véase *Operaciones de bytes y bytearray*.

El parámetro opcional *source* puede ser empleado para inicializar el vector (*array*) de varias maneras distintas:

- Si es una *string*, debes proporcionar también el parámetro *encoding* (y opcionalmente, *errors*; entonces `bytearray()` convierte la cadena a bytes empleando `str.encode()`.
- Si es un *integer*, el array tendrá ese tamaño y será inicializado con bytes nulos.
- 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.

- Si es un *iterable*, debe ser un iterable de enteros en el rango $0 \leq x < 256$, que son usados como los contenidos iniciales del array.

Sin argumento, se crea un array de tamaño 0.

Ver también: *Tipos de secuencias binarias — bytes, bytearray y memoryview* y *Objetos de tipo Bytearray*.

class bytes ([*source* [, *encoding* [, *errors*]]])

Retorna un nuevo objeto *bytes*, que es una secuencia inmutable de enteros en el rango $0 \leq x < 256$. *bytes* es una versión inmutable de *bytearray* — tiene los mismos métodos no-mutables y el mismo comportamiento en términos de indexación y segmentación.

En consecuencia, los argumentos del constructor son interpretados como los de *bytearray* ().

Los objetos de bytes también pueden ser creados con literales, ver strings.

Ver también *Tipos de secuencias binarias — bytes, bytearray y memoryview*, *Objetos de tipo Bytes*, y *Operaciones de bytes y bytearray*.

callable (*object*)

Retorna *True* si el argumento *object* parece invocable, y *False* sino. Si retorna *True*, aun es posible que la invocación falle, pero si es *False*, invocar el *object* no funcionará nunca. Hay que tener en cuenta que las clases son invocables (ya que llamarlas retorna una instancia nueva); y que las instancias lo serán si su clase tiene un método `__call__()`.

Nuevo en la versión 3.2: Esta función fue eliminada en Python 3.0 pero traída de vuelta en Python 3.2.

chr (*i*)

Retorna la cadena que representa un carácter cuyo código Unicode es el entero *i*. Por ejemplo, `chr(97)` retorna la cadena `'a'`, mientras que `chr(8364)` retorna la cadena `'€'`. Esta función es la inversa de *ord* ().

El rango válido para el argumento *i* es desde 0 a 1,114,111 (0x10FFFF en base 16). Se lanzará una excepción *ValueError* si *i* está fuera de ese rango.

@classmethod

Transforma un método en un método de clase.

Un método de clase recibe la clase como primer argumento implícito, de la misma forma que un método de instancia recibe la instancia. Para declarar un método de clase, emplea la siguiente expresión:

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

La forma `@classmethod` es una función *decorator* — ver *function* para más detalles.

Un método de clase puede ser llamado en la clase (como `C.f()`) o en la instancia (como `C().f()`). La instancia es ignorada salvo por su clase. Si un método de clase es llamado desde una clase derivada, entonces el objeto de la clase derivada se pasa como primer argumento implícito.

Los métodos de clase son diferentes a los métodos estáticos de C++ o Java. Si quieres ese tipo de métodos, revisa *staticmethod* ().

Para más información sobre métodos de clase, ver *types*.

compile (*source*, *filename*, *mode*, *flags*=0, *dont_inherit*=False, *optimize*=-1)

Compila el *source* en código o en un objeto AST (*Abstract Syntax Tree*, árbol de sintaxis abstracta). Los objetos código pueden ser ejecutados por las funciones *exec* () or *eval* (). *source* puede ser una cadena normal, una cadena de bytes o un objeto AST. Para más información sobre cómo trabajar con objetos AST, revisa la documentación del módulo *ast*.

El argumento *filename* debería dar el fichero desde el que el código fue leído; pasará un valor reconocible en caso este no fuera leído de un fichero (`'<string>'` es usado comúnmente para esto).

El argumento *mode* especifica que tipo de código debe ser compilado; puede ser `'exec'` si *source* consiste en una secuencia de declaraciones, `'eval'` si consiste de una expresión sencilla, o `'single'` si consiste en una declaración única interactiva (en este último caso, las declaraciones de expresiones que evalúen a algo distinto de `None` serán impresas).

Los argumentos opcionales *flags* y *dont_inherit* controlan que future statements afectan a la compilación de *source*. Si ninguno de los dos está presente (o los dos son cero), el código se compila con las declaraciones futuras que están en el código que está llamando a `compile()`. Si el argumento *flags* está presente, y *dont_inherit* no (o es cero), entonces se usan las declaraciones futuras especificadas en el argumento *flags*, además de aquellas que iban a ser utilizadas en cualquier caso. Si *dont_inherit* es un entero distinto de cero entonces el argumento *flags* es ignorado y no se consideran las declaraciones futuras en efecto durante la llamada a `compile`.

Las declaraciones futuras son especificadas por bits que pueden, bit a bit, ser unidos por OR para especificar múltiples declaraciones. El campo de bit requerido para especificar una característica específica puede ser encontrado como el atributo `compiler_flag` de la instancia de la clase `_Feature` en el módulo `__future__`.

El argumento opcional *flags* también controla si el código compilado puede contener `await`, `async for` y `async with` de nivel superior. Cuando se establece el bit `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT`, el objeto código retornado tiene `CO_COROUTINE` establecido en `co_code`, y puede ser ejecutado de forma interactiva a través de `await eval(code_object)`.

El argumento *optimize* especifica el nivel de optimización del compilador; el valor por defecto de `-1` selecciona el nivel de optimización del intérprete de la misma forma que las opciones `-O`. Los niveles explícitos son `0` (sin optimización; `__debug__` es `true`), `1` (se eliminan los `asserts`, `__debug__` es `false`) or `2` (las *docstrings* también son eliminadas).

Esta función lanza un `SyntaxError` si el código compilado es inválido, y un `ValueError` si contiene bytes nulos.

Si quieres transformar código Python a su representación AST, revisa `ast.parse()`.

Lanza un *evento de auditoría* `compile` con argumentos *source* y *filename*. Este evento también puede ser lanzado por la compilación implícita.

Nota: Cuando se compile una cadena multilínea de código en los modos `'single'` o `'eval'`, la entrada debe terminar con un carácter de nueva línea como mínimo. Esto facilita la detección de declaraciones completas e incompletas en el módulo `code`.

Advertencia: Con una cadena lo suficientemente larga o compleja, al compilar a un objeto AST es posible que el intérprete de Python pare inesperadamente debido a las limitaciones de la profundidad de la pila en el compilador del AST de Python.

Distinto en la versión 3.2: Permite el uso de caracteres de nueva línea de Windows y Mac. También, la entrada en el modo `'exec'` no tiene porque terminar en una nueva línea. Se añade el parámetro *optimize*.

Distinto en la versión 3.5: Anteriormente, un `TypeError` era lanzado cuando se encontraban bytes nulos en *source*.

Nuevo en la versión 3.8: `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` puede ahora ser pasado en *flags* para permitir el soporte de `await`, `async for`, y `async with` en niveles superiores.

class `complex` (`[real[, imag]]`)

Retorna el número complejo con el valor `real + imag*1j` o convierte una cadena o un número a un número complejo. Si el primer parámetro es una cadena, será interpretada como un número complejo y la función debe ser llamada sin un segundo parámetro. El segundo parámetro nunca puede ser una cadena. Cada argumento puede ser de cualquier tipo numérico (incluyendo *complex*). Si se omite *imag*, su valor por defecto es cero y el constructor sirve como un conversor numérico como `int` y `float`. Si ambos argumentos son omitidos, retorna `0j`.

Para un objeto general de Python *x*, `complex(x)` delega a `x.__complex__()`. Si `__complex__()` no está definida, entonces llama a `__float__()`. Si `__float__()` no está definida, entonces llama a `__index__()`.

Nota: Cuando se convierte desde una cadena, la cadena no debe contener espacios en blanco alrededor de los operadores centrales + or -. Por ejemplo, `complex('1+2j')` es correcto, pero `complex('1 + 2j')` lanza `ValueError`.

El tipo complejo está descrito en *Tipos numéricos — int, float, complex*.

Distinto en la versión 3.6: Agrupar dígitos con guiones bajos como en los literales de código está permitido.

Distinto en la versión 3.8: Recurre a `__index__()` si `__complex__()` y `__float__()` no están definidos.

delattr (*object*, *name*)

Esta función está emparentada con `setattr()`. Los argumentos son un objeto y una cadena. La cadena debe ser el mismo nombre que alguno de los atributos del objeto. La función elimina el atributo nombrado, si es que el objeto lo permite. Por ejemplo `delattr(x, 'foobar')` es equivalente a `del x.foobar`.

class dict (***kwarg*)

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

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

Crea un nuevo diccionario. El objeto `dict` es la clase diccionario. Véase `dict` y *Tipos Mapa — dict* para más información sobre esta clase.

Para otros contenedores véase las clases incorporadas (*built-in*) `list`, `set`, y `tuple`, así como el módulo `collections`.

dir ([*object*])

Sin argumentos, retorna la lista de nombres en el ámbito local. Con un argumento, intenta retornar una lista de atributos válidos para ese objeto.

Si el objeto tiene un método llamado `__dir__()`, éste será llamado y debe retornar la lista de atributos. Esto permite que los objetos que implementan una función personalizada `__getattr__()` o `__getattribute__()` puedan decidir la manera en la que `dir()` reporta sus atributos.

Si el objeto no provee de un método `__dir__()`, la función intenta obtener la información del atributo `__dict__` del objeto, si está definido, y de su objeto tipo. La lista resultante no está necesariamente completa, y puede ser inexacta cuando el objeto tiene una función `__getattr__()` implementada.

El mecanismo por defecto de `dir()` se comporta de forma distinta con diferentes tipos de objeto, ya que intenta producir la información más relevante en vez de la más completa:

- Si el objeto es un módulo, la lista contiene los nombres de los atributos del módulo.
- Si el objeto es un tipo o una clase, la lista contiene los nombres de sus atributos, y recursivamente la de los atributos de sus clases base.
- En cualquier otro caso, la lista contiene los nombres de los atributos del objeto, los nombres de los atributos de su clase, y recursivamente los atributos de sus clases base.

La lista resultante está ordenada alfabéticamente. Por ejemplo:

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

Nota: Ya que `dir()` se ofrece como una herramienta para uso en una terminal de forma interactiva, intenta ofrecer un grupo interesante de nombres más que un uno rigurosamente definido, y su comportamiento detallado puede cambiar entre versiones. Por ejemplo, los atributos de metaclasses no están en la lista resultante cuando el argumento es una clase.

divmod(*a*, *b*)

Toma dos números (no complejos) como argumentos y retorna un par de números consistentes en su cociente y su resto al emplear división de enteros. Con operandos de tipos diferentes, se aplican las reglas de los operadores aritméticos binarios. Para enteros, el resultado es el mismo que $(a // b, a \% b)$. Para números de punto flotante el resultado es $(q, a \% b)$, donde q normalmente es `math.floor(a / b)` pero puede ser 1 menos que eso. En cualquier caso $q * b + a \% b$ es muy cercano a a , si $a \% b$ es distinto de cero y tiene el mismo signo que b , y $0 \leq \text{abs}(a \% b) < \text{abs}(b)$.

enumerate(*iterable*, *start=0*)

Retorna un objeto enumerador. *iterable* tiene que ser una secuencia, un *iterator*, o algún otro objeto que soporte la iteración. El método `__next__()` del iterador retornado por la función `enumerate()` retorna una tupla que contiene un contador (desde *start*, cuyo valor por defecto es 0) y los valores obtenidos al iterar sobre *iterable*.

```
>>> 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')]
```

Equivalente a:

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

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

Los argumentos son una cadena y opcionalmente, globales y locales. Si se introduce, *globals* tiene que ser un diccionario, y *locals* puede ser cualquier objeto de mapeo.

El argumento *expression* se parsea y evalúa como una expresión de Python (técnicamente, una lista de condiciones), usando los diccionarios *globals* y *locals* como espacios de nombres globales y locales. Si el diccionario *globals* está presente y no contiene un valor para la clave `__builtins__`, se inserta bajo esa clave una referencia al diccionario del módulo incorporado `builtins` antes de que la *expression* sea parseada. Esto significa que en condiciones normales *expression* tiene acceso total al módulo estándar `builtins` y los entornos restringidos son propagados. Si el diccionario *locals* es omitido entonces su valor por defecto es el diccionario *globals*. Si ambos diccionarios son omitidos, la expresión es ejecutada con las *globals* y *locals* del entorno en el que `eval()` es llamada. Tenga en cuenta que `eval()` no tiene acceso al *alcances anidados* (no-locales) en el entorno que lo contiene.

El valor que retorna es el resultado de la expresión evaluada. Los errores de sintaxis son reportados como excepciones. Por ejemplo:

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

La función también puede ser utilizada para ejecutar objetos código arbitrario (como los que crea la función `compile()`). En este caso, se pasa un objeto código en vez de una cadena. Si el objeto código ha sido compilado usando 'exec' como el argumento *mode*, el valor que retornará `eval()` será `None`.

Pista: la ejecución dinámica de declaraciones está soportada por la función `exec()`. Las funciones `globals()` y `locals()` retornan los diccionarios global y local en ese momento, lo cual puede ser útil para su uso en `eval()` o `exec()`.

Véase `ast.literal_eval()`, una función que puede evaluar de forma segura cadenas con expresiones que contienen solo literales.

Lanza un *evento de auditoría* `exec` con un argumento `code_object`.

exec (*object* [, *globals* [, *locals*]])

This function supports dynamic execution of Python code. *object* must be either a string or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs).¹ If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file input (see the section «File input» in the Reference Manual). Be aware that the `nonlocal`, `yield`, and `return` statements may not be used outside of function definitions even within the context of code passed to the `exec()` function. The return value is `None`.

En todos los casos, si las partes opcionales son omitidas, el código es ejecutado en el ámbito actual. Si solo se indica *globals*, debe ser un diccionario (y no una subclase de diccionario), que será usada tanto para las variables locales como las globales. Si se indican tanto *globals* como *locals*, son usadas para las variables globales y locales respectivamente. Si se indican, *locals* puede ser cualquier objeto de mapeo. Recuerda que a nivel de módulo, *globals* y *locals* son el mismo diccionario. Si `exec` recibe dos objetos separados como *globals* y *locals*, el código será ejecutado como si estuviera incorporado en una definición de clase.

Si el diccionario *globals* no contiene un valor para la clave `__builtins__`, una referencia al diccionario del módulo built-in `builtins` se inserta bajo esa clave. De esta forma puedes controlar que `builtins` están disponibles para el código ejecutado insertando tu propio diccionario `__builtins__` en *globals* antes de pasárselo a `exec()`.

Lanza un *evento de auditoría* `exec` con un argumento `code_object`.

Nota: Las funciones built-in `globals()` y `locals()` Retornan el diccionario local y global actual, respectivamente, lo que puede ser útil para pasarlo y emplearlo como el segundo y el tercer argumento de `exec()`.

Nota: El *locals* por defecto actúa de la forma descrita para la función `locals()` más abajo: no se deben intentar modificaciones sobre el diccionario *locals* por defecto. Pasa un diccionario explícito *locals* si necesitas ver los efectos del código en *locals* después de que la función `exec()` retorne.

filter (*function*, *iterable*)

Construye un iterador a partir de aquellos elementos de *iterable* para los cuales *function* retorna `true`. *iterable* puede ser una secuencia, un contenedor que soporta iteración, o un iterador. Si *function* es `None`, se asume la función identidad, es decir, todos los elementos de *iterable* que son `false` son eliminados.

Ten en cuenta que `filter(function, iterable)` es equivalente a la expresión de un generador `(item for item in iterable if function(item))` si *function* no es `None` y a `(item for item in iterable if item)` si *function* es `None`.

Ver `itertools.filterfalse()` para la función complementaria que retorna los elementos de *iterable* para los cuales *function* retorna `false`.

class float ([*x*])

Retorna un número de punto flotante construido a partir de un número o una cadena *x*.

Si el argumento es una cadena, debe contener un número decimal, opcionalmente precedido de un signo, y opcionalmente entre espacios en blanco. El signo opcional puede ser `'+'` o `'-'`; un signo `'+'` no tiene efecto en el valor producido. El argumento puede ser también una cadena representando un NaN (no es un número), o un infinito positivo o negativo. Más concretamente, el argumento debe adecuarse a la siguiente gramática una vez eliminados de la cadena los caracteres en blanco por delante o detrás:

```
sign      ::=      "+" | "-"
```

¹ Ten en cuenta que el *parser* sólo acepta la convención de final de línea de tipo Unix. Si estás leyendo el código de un fichero, asegúrate de usar la el modo de conversión de nueva línea para convertir las líneas de tipo Windows o Mac.

```
infinity      ::= "Infinity" | "inf"
nan           ::= "nan"
numeric_value ::= floatnumber | infinity | nan
numeric_string ::= [sign] numeric_value
```

Aquí `floatnumber` es el formato de un literal de punto flotante de Python, tal y como está descrito en [floating](#). No es relevante si los caracteres son mayúsculas o minúsculas, de forma que «inf», «Inf», «INFINITY» e «iNfINity» son todas formas aceptables de escribir el infinito positivo.

Sino, en caso de que el argumento sea un entero o un decimal de punto flotante, se retorna un número de punto flotante del mismo valor (dentro de la precisión de punto flotante de Python). Si el argumento está fuera del rango de un punto flotante de Python, se generará una excepción `OverflowError`.

Para el objeto general de Python `x`, `float(x)` delega a `x.__float__()`. Si `__float__()` no está definido entonces recurre a `__index__()`.

Si no se le da un argumento, retorna `0.0`.

Ejemplos:

```
>>> float('+1.23')
1.23
>>> float('  -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

El tipo `float` está descrito en [Tipos numéricos — int, float, complex](#).

Distinto en la versión 3.6: Agrupar dígitos con guiones bajos como en los literales de código está permitido.

Distinto en la versión 3.7: `x` es ahora un argumento solo de posición.

Distinto en la versión 3.8: Recurre a `__index__()` si `__float__()` no está definido.

format (*value* [, *format_spec*])

Convierte *value* a su representación «con formato», de forma controlada por *format_spec*. La interpretación de *format_spec* dependerá del tipo del argumento *value*. Sin embargo, hay una sintaxis estándar de formato que emplean la mayoría de los tipos built-in: [Especificación de formato Mini-Lenguaje](#).

El *format_spec* por defecto es una cadena vacía que normalmente produce el mismo efecto que llamar a `str(value)`.

Una llamada a `format(value, format_spec)` se traduce a `type(value).__format__(value, format_spec)`, que sortea el diccionario de la instancia cuando busca por el método `__format__()` del valor. Una excepción `TypeError` será lanzada si la búsqueda del método llega a *object* y *format_spec* no está vacío, o si *format_spec* o el valor de retorno no son cadenas.

Distinto en la versión 3.4: `object().__format__(format_spec)` lanza `TypeError` si *format_spec* no es una cadena vacía.

class frozenset ([*iterable*])

Retorna un nuevo objeto *frozenset*, opcionalmente con elementos tomados de *iterable*. *frozenset* es una clase built-in. Ver [frozenset](#) y [Conjuntos — set, frozenset](#) para documentación sobre esta clase.

Para otro tipo de contenedores, ver las clases built-in *set*, *list*, *tuple*, y *dict*, así como el módulo *collections*.

getattr (*object*, *name* [, *default*])

Retorna el valor del atributo nombrado de *object*. *name* debe ser una cadena. Si la cadena es el nombre de uno de los atributos del objeto, el resultado es el valor de ese atributo. Por ejemplo, `getattr(x, 'foobar')`

es equivalente a `x.foo`. Si el atributo nombrado no existe, se retorna *default* si ha sido proporcionado como argumento, y sino se lanza una excepción `AttributeError`.

globals()

Retorna un diccionario que representa la tabla global de símbolos. Es siempre el diccionario del módulo actual (dentro de una función o método, este es el módulo donde está definida, no el módulo desde el que es llamada).

hasattr(object, name)

Los argumentos son un objeto y una cadena. El resultado es `True` si la cadena es el nombre de uno de los atributos del objeto, y `False` en caso contrario. (Está implementado mediante una llamada a `getattr(object, name)` que comprueba si se lanza una excepción `AttributeError` o no).

hash(object)

Retorna el valor hash del objeto (si tiene uno). Los valores hash son enteros. Se usan para comparar de forma rápida claves de diccionarios durante las operaciones de búsqueda. Valores numéricos que son iguales tienen el mismo valor hash (incluso si son de tipos diferentes, como es el caso para 1 y 1.0).

Nota: Para objetos que implementan métodos `__hash__()`, ten en cuenta que `hash()` trunca el valor de retorno en base a la tasa de bits de la máquina host. Ver `__hash__()` para más detalles.

help([object])

Invoca el sistema de ayuda integrado (built-in). (Esta función está indicada para su uso interactivo). Si no se le da argumento, el sistema interactivo de ayuda se inicia en la consola del intérprete. Si el argumento es una cadena, entonces es buscada como nombre de un módulo, función, clase, método, palabra clave o tema de documentación, y una página de ayuda es impresa en la consola. Si el argumento es cualquier otro tipo de objeto, una página de ayuda sobre el objeto es generada.

Ten en cuenta que si una barra(/) aparece en la lista de parámetros de una función, al invocar `help()` significa que los parámetros anteriores a la barra son solo posicionales. Para más información, puedes ver the FAQ entry on positional-only parameters.

Esta función se añade al espacio de nombres built-in a través del módulo `site`.

Distinto en la versión 3.4: Cambios a los módulos `pydoc` y `inspect` implican que las firmas reportadas para objetos invocables son más completas y consistentes.

hex(x)

Convierte un número entero a una cadena hexadecimal de minúsculas con el prefijo «0x». Si `x` no es un objeto de la clase Python `int`, tiene que definir un método `__index__()` que retorne un entero. Algunos ejemplos:

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

Si quieres convertir un número entero a una cadena hexadecimal de mayúsculas o minúsculas con prefijo o sin el, puedes usar cualquiera de las siguientes formas:

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')
```

Véase también `format()` para más información.

Ver también `int()` para convertir una cadena hexadecimal a un entero usando una base de 16.

Nota: Para obtener una cadena hexadecimal que represente un punto flotante, utiliza el método `float.hex()`.

id (*object*)

Retorna la «identidad» de un objeto. Esto es un entero que está garantizado que es único y constante para este objeto durante toda su existencia. Dos objetos con existencias en el tiempo que no coincidan pueden tener el mismo valor de `id()`.

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

Raises an [auditing event](#) `builtins.id` with argument `id`.

input ([*prompt*])

Si el argumento *prompt* está presente, se escribe a la salida estándar sin una nueva línea a continuación. La función lee entonces una línea de la entrada, la convierte en una cadena (eliminando la nueva línea), y retorna eso. Cuando se lee EOF, se lanza una excepción [EOFError](#). Ejemplo:

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

Si el módulo [readline](#) estaba cargado, entonces `input()` lo usará para proporcionar características más elaboradas de edición de líneas e historiales.

Lanza un [evento de auditoría](#) `builtins.input` con el argumento `prompt` antes de leer entrada

Lanza un evento de auditoría `builtins.input/result` con el resultado justo después de haber leído con éxito la entrada.

class int ([*x*])**class int** (*x*, *base*=10)

Retorna un objeto entero construido desde un número o cadena *x*, o retorna 0 si no se le proporcionan argumentos. Si *x* define `__int__()`, `int(x)` retorna `x.__int__()`. Si *x* define `__index__()`, retorna `x.__index__()`. Si *x* define `__trunc__()`, retorna `x.__trunc__()`. Para números de punto flotante, los valores serán truncados hacia cero.

Si *x* no es un número o si se indica *base*, entonces *x* debe ser una cadena, una instancia de [bytes](#), o una de [bytearray](#) que representa un integer literal de base *base*. Opcionalmente, el literal puede ser precedido de + or - (sin espacios entre el número y el signo) y rodeados por espacio en blanco. Un literal de base-*n* consiste en los dígitos de 0 a *n*-1, con valores entre 10 y 35 para los caracteres de a a z (o de A a Z). La *base* por defecto es 10. Los valores permitidos son 0 y 2-36. Los literales de base-2, -8 y -16 pueden incluir opcionalmente un prefijo `0b/0B`, `0o/0O`, o `0x/0X`, de igual forma que los literales enteros en el código. Base-0 indica que se debe interpretar exactamente como un literal de código, de forma que la base real es 2, 8, 10 o 16, y que `int('010', 0)` no sea legal, mientras que `int('010')` sí lo es, así como `int('010', 8)`.

El tipo entero se describe en [Tipos numéricos — int, float, complex](#).

Distinto en la versión 3.4: Si *base* no es una instancia de [int](#) y el objeto *base* tiene un método `base.__index__`, ese método es llamado para obtener un entero para esa base. En versiones anteriores se empleaba `base.__int__` en vez de `base.__index__`.

Distinto en la versión 3.6: Agrupar dígitos con guiones bajos como en los literales de código está permitido.

Distinto en la versión 3.7: *x* es ahora un argumento solo de posición.

Distinto en la versión 3.8: Recurre a `__index__()` si no está definido `__int__()`.

Distinto en la versión 3.8.14: [int](#) string inputs and string representations can be limited to help avoid denial of service attacks. A [ValueError](#) is raised when the limit is exceeded while converting a string *x* to an [int](#) or when converting an [int](#) into a string would exceed the limit. See the [integer string conversion length limitation](#) documentation.

isinstance (*object*, *classinfo*)

Retorna `True` si el argumento *object* es una instancia del argumento *classinfo*, o de una subclase (directa, indirecta o [virtual](#)) del mismo. Si *object* no es un objeto del tipo indicado, esta función siempre retorna `False`. Si *classinfo* es una tupla de objetos de tipo (o recursivamente, otras tuplas lo son), retorna `True` si *object* es una instancia de alguno de los tipos. Si *classinfo* no es un tipo, una tupla de tipos, ó una tupla de tuplas de tipos, una excepción [TypeError](#) es lanzada.

issubclass (*class*, *classinfo*)

Retorna `True` si *class* es una subclase (directa, indirecta o *virtual*) de *classinfo*. Una clase es considerada una subclase de sí misma. *classinfo* puede ser una tupla de objetos de clase, en cuyo caso cada elemento en *classinfo* será comprobado. En cualquier otro caso, se lanzará una excepción `TypeError`.

iter (*object* [, *sentinel*])

Retorna un objeto *iterator*. El primer argumento se interpreta de forma muy diferente en función de la presencia del segundo. Sin un segundo argumento, *object* debe ser un objeto *collection* que soporte el protocolo de iteración (el método `__iter__()`), o debe soportar el protocolo de secuencia (el método `__getitem__()` con argumentos enteros comenzando en 0). Si no soporta ninguno de esos protocolos, se lanza una excepción `TypeError`. Si el segundo argumento, *sentinel*, es indicado, entonces *object* debe ser un objeto invocable. El iterador creado en ese caso llamará a *object* sin argumentos para cada invocación a su método `__next__()`; si el valor retornado es igual a *sentinel*, una `StopIteration` será lanzada, de lo contrario el valor será retornado.

Ver también *Tipos de iteradores*.

Una aplicación muy útil de la segunda forma de `iter()` es la construcción de un lector de bloques. Por ejemplo, leer bloques de ancho fijo de una base de datos binaria hasta que el fin del fichero sea alcanzado:

```
from functools import partial
with open('mydata.db', 'rb') as f:
    for block in iter(partial(f.read, 64), b''):
```

len (*s*)

Retorna el tamaño (el número de elementos) de un objeto. El argumento puede ser una secuencia (como una cadena, un objeto `byte`, una tupla, lista o rango) o una colección (como un diccionario, un `set` o un `frozen set`).

class list ([*iterable*])

Más que una función, *list* es realmente un tipo de secuencia mutable, como está documentado en *Listas* y *Tipos secuencia — list, tuple, range*.

locals ()

Actualiza y retorna un diccionario representando la tabla de símbolos locales actual. Las variables libres son retornadas por `locals()` cuando ésta es llamada en bloques de funciones, pero no en bloques de clases. Nótese que a nivel de módulo, `locals()` y `globals()` son el mismo diccionario.

Nota: Los contenidos de este diccionario no deben ser modificados; sus cambios no afectarán los valores de las variables locales y libres utilizadas por el intérprete.

map (*function*, *iterable*, ...)

Retorna un iterador que aplica *function* a cada elemento de *iterable*, retornando el resultado. Si se le pasan argumentos adicionales de tipo *iterable*, *function* debe tomar la misma cantidad de argumentos y es aplicado a los elementos de todos ellos en paralelo. Con iterables múltiples, el iterador se detiene cuando el iterable más corto se agota. Para casos donde las entradas de la función ya están organizadas como tuplas de argumentos, ver `itertools.starmap()`.

max (*iterable*, *[, *key*, *default*])

max (*arg1*, *arg2*, **args* [, *key*])

Retorna el elemento mayor en un iterable o el mayor de dos o más argumentos.

Si un argumento posicional es dado, debe ser un *iterable*. El elemento mayor en el iterable es retornado. Si dos o más argumentos posicionales son indicados, el mayor de los argumentos posicionales será retornado.

Hay dos argumentos de solo palabra clave que son opcionales. El argumento *key* especifica una función de ordenación de un sólo argumento, como la usada para `list.sort()`. El argumento *default* especifica un objeto a retornar si el iterable proporcionado está vacío. Si el iterable está vacío y *default* no ha sido indicado, se lanza un `ValueError`.

Si hay múltiples elementos con el valor máximo, la función retorna el primero que ha encontrado. Esto es consistente con otras herramientas para preservar la estabilidad de la ordenación como `sorted(iterable,`

`key=keyfunc, reverse=True)[0]` y `heapq.nlargest(1, iterable, key=keyfunc)`.

Nuevo en la versión 3.4: El argumento *default* sólo por palabra clave.

Distinto en la versión 3.8: *key* puede ser `None`.

class `memoryview` (*obj*)

Retorna un objeto «*memory view*» creado a partir del argumento indicado. Para más información ver [Vistas de memoria](#).

min (*iterable*, *[, *key*, *default*])

min (*arg1*, *arg2*, **args*[, *key*])

Retorna el menor elemento en un iterable o el menor de dos o más argumentos.

Si se le indica un argumento posicional, debe ser un *iterable*. El menor elemento del iterable es retornado. Si dos o más argumentos posicionales son indicados, el menor de los argumentos posicionales es retornado.

Hay dos argumentos de solo palabra clave que son opcionales. El argumento *key* especifica una función de ordenación de un sólo argumento, como la usada para `list.sort()`. El argumento *default* especifica un objeto a retornar si el iterable proporcionado está vacío. Si el iterable está vacío y *default* no ha sido indicado, se lanza un `ValueError`.

Si hay múltiples elementos con el valor mínimo, la función retorna el primero que encuentra. Esto es consistente con otras herramientas que preservan la estabilidad de la ordenación como `sorted(iterable, key=keyfunc)[0]` y `heapq.nsmallest(1, iterable, key=keyfunc)`.

Nuevo en la versión 3.4: El argumento *default* sólo por palabra clave.

Distinto en la versión 3.8: *key* puede ser `None`.

next (*iterator*[, *default*])

Extrae el siguiente elemento de *iterator* llamando a su método `__next__()`. Si se le indica *default*, éste será retornado si se agota el iterador, de lo contrario, se lanza un `StopIteration`.

class `object`

Retorna un nuevo objeto indiferenciado. *object* es la base de todas las clases. Tiene todos los métodos que son comunes a todas las instancias de clases de Python. Esta función no acepta ningún argumento.

Nota: *object* no tiene un `__dict__`, así que no puedes asignar atributos arbitrarios a una instancia de la clase *object*.

oct (*x*)

Convierte un número entero a una cadena octal con prefijo «0o». El resultado es una expresión válida de Python. Si *x* no es un objeto de la clase Python *int*, tiene que tener definido un método `__index__()` que retorne un entero. Por ejemplo:

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

Si quieres convertir un número entero a una cadena octal, tanto con prefijo «0o» como sin el, puedes usar cualquiera de las siguientes formas.

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')
```

Véase también `format()` para más información.

open (*file*, *mode*='r', *buffering*=-1, *encoding*=None, *errors*=None, *newline*=None, *closefd*=True, *opener*=None)

Open *file* and return a corresponding *file object*. If the file cannot be opened, an *OSError* is raised. See *tut-files* for more examples of how to use this function.

file es un *path-like object* que da la ruta (absoluta o relativa al directorio de trabajo actual) del fichero a ser abierto o un descriptor de fichero entero del fichero a ser envuelto. (Si un descriptor de fichero es dado, será cerrado cuando el objeto de entrada-salida sea cerrado, a menos que *closefd* esté puesto a *False*.)

mode es una cadena opcional que especifica el modo en el cual el fichero es abierto. Su valor por defecto es 'r' que significa que está abierto para lectura en modo texto. Otros valores comunes son 'w' para escritura (truncando el fichero si ya existe), 'x' para creación en exclusiva y 'a' para añadir (lo que en algunos sistemas Unix implica que *toda* la escritura añade al final del fichero independientemente de la posición de búsqueda actual). En modo texto, si no se especifica *encoding* entonces la codificación empleada es dependiente de plataforma: se invoca a `locale.getpreferredencoding(False)` para obtener la codificación regional actual. (Para lectura y escritura de bytes en crudo usa el modo binario y deja *encoding* sin especificar). Los modos disponibles son:

Carácter	Significado
'r'	abierto para lectura (por defecto)
'w'	abierto para escritura, truncando primero el fichero
'x'	abierto para creación en exclusiva, falla si el fichero ya existe
'a'	abierto para escritura, añadiendo al final del fichero si este existe
'b'	modo binario
't'	modo texto (por defecto)
'+'	abierto para actualizar (lectura y escritura)

El modo por defecto es 'r' (abierto para lectura de texto, sinónimo de 'rt'. Los modos 'w+' y 'wb+' abren y truncan el fichero. Los modos 'r+' y 'rb+' abren el fichero sin truncarlo.

Como se menciona en *Resumen*, Python distingue entre I/O binario y de texto. Los ficheros abiertos en modo binario (incluyendo 'b' en el argumento *mode*) retornan su contenido como objetos de *bytes* sin ninguna decodificación. En modo de texto (por defecto, o cuando se incluye 't' en el argumento *mode*), los contenidos del fichero se retornan como *str*, tras decodificar los *bytes* usando una codificación dependiente de plataforma o usando el *encoding* especificado como argumento.

Hay un carácter adicional permitido para indicar un modo, 'U', que ya no tiene ningún efecto y que se considera obsoleto. Permitía anteriormente en modo texto *universal newlines*, lo que se convirtió en el comportamiento por defecto en Python 3.0. Para más detalles, referirse a la documentación del parámetro *newline*.

Nota: Python no depende de la noción de ficheros de texto del sistema operativo subyacente; todo el procesado lo hace Python, y es por tanto independiente de plataforma.

buffering es un entero opcional que configura la política de buffering. Indica 0 para desactivar el buffering (sólo permitido en modo binario), 1 para seleccionar buffering por línea (sólo para modo texto), y un entero >1 para indicar el tamaño en bytes de un buffer de tamaño fijo. Cuando no se indica el argumento *buffering*, la norma por defecto de buffering funciona de la siguiente manera:

- Los ficheros binarios son transmitidos por búferes con tamaños fijos de bloque; el tamaño del búfer es escogido usando un intento heurístico para determinar el tamaño de bloque del dispositivo y recurriendo sino a `io.DEFAULT_BUFFER_SIZE`. En muchos sistemas, el búfer tendrá normalmente un tamaño de 4096 o 8192 bytes.
- Los ficheros de texto «interactivos» (ficheros para los cuales `isatty()` retorna *True*) usan buffering por líneas. Otros ficheros de texto emplean la norma descrita anteriormente para ficheros binarios.

encoding es el nombre de la codificación empleada con el fichero. Esto solo debe ser usado en el modo texto. La codificación por defecto es dependiente de plataforma (aquello que retorna `locale.getpreferredencoding()`), pero puede emplearse cualquier *text encoding* soportado por Python. Véase el módulo *codecs* para la lista de codificaciones soportadas.

errors es una cadena opcional que especifica como deben manejarse los errores de codificación y decodificación – esto no puede ser usado en modo binario. Están disponibles varios gestores de error (listados en *Manejadores de errores*), pero también es válido cualquier nombre de gestión de error registrado con *codecs.register_error()*. Los nombres estándar incluyen:

- 'strict' para lanzar una excepción *ValueError* si hay un error de codificación. El valor por defecto, *None*, produce el mismo efecto.
- 'ignore' ignora los errores. Nótese que ignorar errores de codificación puede conllevar la pérdida de datos.
- 'replace' provoca que se inserte un marcador de reemplazo (como '?') en aquellos sitios donde hay datos malformados.
- 'surrogateescape' representa bytes incorrectos como puntos código del *Unicode Private Use Area* que van desde U+DC80 a U+DCFF. Estos puntos código privados volverán a convertirse en los mismos bytes cuando el gestor de errores *surrogateescape* sea usando al escribir datos. Esto es útil para el procesamiento de ficheros con una codificación desconocida.
- 'xmlcharrefreplace' está soportado solamente cuando se escribe a un fichero. Los caracteres que no estén soportados por la codificación son reemplazados por la referencia al carácter XML apropiado `&#nnn;`.
- 'backslashreplace' reemplaza datos malformados con las secuencias de escapes de barra invertida de Python.
- 'namereplace' reemplaza caracteres no soportados con secuencias de escape `\N{...}` (y también está sólo soportado en escritura).

newline controla cómo funciona el modo *universal newlines* (solo aplica a modo texto). Puede ser *None*, '\n', '\r', y '\r\n'. Funciona de la siguiente manera:

- Cuando se está leyendo entrada desde el flujo, si *newline* es *None*, el modo *universal newlines* es activado. Las líneas en la entrada pueden terminar en '\n', '\r', o '\r\n', y serán traducidas a '\n' antes de ser retornadas a la entidad que llama. Si es '\n', el modo *universal newlines* estará activado pero los finales de línea se retornan sin traducir a la entidad que llama. Si tiene cualquiera de los otros valores válidos, las líneas de entrada estarán terminadas sólo por la cadena dada, y los finales de línea serán retornados sin traducir a la entidad que llama.
- Cuando se escribe salida al flujo, si *newline* es *None*, cualquier carácter '\n' escrito es traducido al separador de línea por defecto en el sistema, *os.linesep*. Si *newline* es '\n' o '\r\n', entonces no se produce ninguna traducción. Si *newline* toma cualquiera de los otros valores válidos, entonces cualquier carácter '\n' escrito es traducido a la cadena indicada.

Si *closefd* es *False* y se indica un descriptor de fichero en vez de un nombre de fichero, el descriptor se mantendrá abierto cuando se cierre el fichero. Si se indica un nombre de fichero, *closefd* debe ser *True* (lo es por defecto), ya que de otra forma se lanzará un error.

Un abridor personalizado puede emplearse pasando un invocable como *opener*. El descriptor de fichero del objeto se obtiene llamando a *opener* con (*file*, *flags*). *opener* debe retornar un descriptor de fichero abierto (pasando *os.open* como *opener* resulta en una funcionalidad similar a *None*).

El nuevo fichero creado es *no-heredable*.

El siguiente ejemplo emplea el parámetro *dir_fd* de la función *os.open()* para abrir un fichero relativo a un directorio dado:

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt', file=f)
...
>>> os.close(dir_fd) # don't leak a file descriptor
```


El tipo de *file object* retornado por la función `open()` depende del modo. Cuando se emplea `open()` para abrir un fichero en modo texto ('w', 'r', 'wt', 'rt', etc.), retorna una subclase de `io.TextIOBase` (específicamente `io.TextIOWrapper`). Cuando se emplea para abrir un fichero en modo binario con buffering, la clase retornada es una subclase de `io.BufferedIOBase`. La clase exacta varía: en modo binario de lectura, retorna `io.BufferedReader`; en modo de escritura y adición en binario, retorna `io.BufferedWriter`, y en modo de escritura/lectura, retorna `io.BufferedRandom`. Si el buffering está desactivado, el flujo en crudo, una subclase de `io.RawIOBase`, `io.FileIO`, es retornada.

Véase también los módulos para manejo de ficheros, como `fileinput`, `io` (donde es declarada `open()`), `os`, `os.path`, `tempfile`, y `shutil`.

Lanza un *evento de auditoría* `open` con los argumentos `file`, `mode`, `flags`.

Los argumentos `mode` y `flags` pueden haber sido modificados o inferidos de la llamada original.

Distinto en la versión 3.3:

- El parámetro `opener` fue añadido.
- El modo 'x' fue añadido.
- `IOError` era la excepción lanzada anteriormente, ahora es un alias de `OSError`.
- Se lanza `FileExistsError` si ya existe el fichero abierto en modo de creación exclusiva ('x').

Distinto en la versión 3.4:

- El fichero ahora es no-heredable.

Deprecated since version 3.4, will be removed in version 3.9: El modo 'U'.

Distinto en la versión 3.5:

- Si la llamada al sistema es interrumpida y el gestor de señales no lanza una excepción, ahora la función reintenta la llamada de sistema en vez de lanzar una excepción `InterruptedError` (véase [PEP 475](#) para la justificación).
- El gestor de errores 'namereplace' fue añadido.

Distinto en la versión 3.6:

- Añadido el soporte para aceptar objetos que implementan `os.PathLike`.
- En Windows, abrir un búfer en la consola puede retornar una subclase de `io.RawIOBase` en vez de `io.FileIO`.

ord(c)

Al proporcionarle una cadena representando un carácter Unicode, retorna un entero que representa el código Unicode de ese carácter. Por ejemplo, `ord('a')` retorna el entero 97 y `ord('€')` (símbolo del Euro) retorna 8364. Esta es la función inversa de `chr()`.

pow(base, exp[, mod])

Retorna *base* elevado a *exp*; si *mod* está presente, retorna *base* elevado a *exp*, módulo *mod* (calculado de manera más eficiente que `pow(base, exp) % mod`). La forma con dos argumentos `pow(base, exp)` es equivalente a usar el operador potencia: `base**exp`.

Los argumentos deben ser de tipo numérico. Si hay tipos mixtos de operandos, las reglas de coerción para operadores binarios aritméticos aplican. Para operandos de la clase `int`, el resultado tiene el mismo tipo que los operandos (después de la coerción) a menos que el segundo argumento sea negativo; en tal caso, todos los argumentos son convertidos a punto flotante y un resultado de punto flotante es retornado. Por ejemplo, `10**2` retorna 100, pero `10**-2` retorna 0.01.

Para operandos `int` como *base* y *exp*, si *mod* está presente, *mod* debe ser también de tipo entero y distinto de cero. Si *mod* está presente y *exp* es negativo, *base* debe ser un número primo relativo a *mod*. En ese caso, se retorna `pow(inv_base, -exp, mod)`, donde *inv_base* es la inversa al módulo *mod* de *base*.

Aquí tienes un ejemplo de cómo calcular la inversa de 38 módulo 97:

```
>>> pow(38, -1, mod=97)
23
>>> 23 * 38 % 97 == 1
True
```

Distinto en la versión 3.8: Para operandos `int`, la forma de `pow` con tres argumentos acepta ahora que el segundo argumento sea negativo, lo que permite el cálculo de inversos modulares.

Distinto en la versión 3.8: Permite argumentos de palabra clave. Anteriormente, solo se soportaba el uso de argumentos posicionales.

print (*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Imprime *objects* al flujo de texto *file*, separándolos por *sep* y seguidos por *end*. *sep*, *end*, *file* y *flush*, si están presentes, deben ser dados como argumentos por palabra clave.

Todos los argumentos que no son por palabra clave se convierten a cadenas tal y como `str()` hace y se escriben al flujo, separados por *sep* y seguidos por *end*. Tanto *sep* como *end* deben ser cadenas; también pueden ser `None`, lo cual significa que se empleen los valores por defecto. Si no se indica *objects*, `print()` escribirá *end*.

El argumento *file* debe ser un objeto que implemente un método `write(string)`; si éste no está presente o es `None`, se usará `sys.stdout`. Dado que los argumentos mostrados son convertidos a cadenas de texto, `print()` no puede ser utilizada con objetos fichero en modo binario. Para esos, utiliza en cambio `file.write(...)`.

Que la salida sea en búfer o no suele estar determinado por *file*, pero si el argumento por palabra clave *flush* se emplea, el flujo se descarga forzosamente.

Distinto en la versión 3.3: Añadido el argumento por palabra clave *flush*.

class property (fget=None, fset=None, fdel=None, doc=None)

Retorna un atributo propiedad.

fget es una función para obtener el valor de un atributo. *fset* es una función para asignar el valor de un atributo. *fdel* es una función para eliminar el valor de un atributo. Y *doc* crea un *docstring* para el atributo.

Un caso de uso típico es la definición de un atributo gestionado x:

```
class C:
    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.")
```

Si *c* es una instancia de *C*, *c.x* invocará el obtenedor (*getter*), *c.x = value* invocará el asignador (*setter*) y *del c.x* el suprimidor (*deleter*).

Si está indicada, *doc* será la *docstring* del atributo propiedad. En caso contrario, la propiedad copiará la *docstring* de *fget* si ésta existe. Esto permite crear propiedades de sólo lectura de forma fácil empleando `property()` como *decorator*:

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

(continué en la próxima página)

(proviene de la página anterior)

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

El decorador `@property` convierte el método `voltage()` en un «getter» para un atributo de sólo lectura con el mismo nombre, y asigna «Get the current voltage.» como la *docstring* de `voltage`.

Un objeto propiedad tiene métodos `getter`, `setter`, y `deleter` que pueden usarse como decoradores que crean una copia de la propiedad con su correspondiente función de acceso asignada a la función decorada. Esto se explica mejor con un ejemplo:

```
class C:
    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
```

Este código equivale exactamente al primer ejemplo. Asegúrese de otorgarle a las funciones adicionales el mismo nombre que la propiedad original (`x` en este caso.)

El objeto propiedad retornado tiene también los atributos `fget`, `fset`, y `fdel` correspondientes a los argumentos del constructor.

Distinto en la versión 3.5: Las *docstrings* de los objetos propiedad son escribibles.

class `range` (*stop*)

class `range` (*start*, *stop* [, *step*])

range, más que una función, es en realidad un tipo de secuencia inmutable, tal y como está documentado en *Rangos* y *Tipos secuencia — list, tuple, range*.

repr (*object*)

Retorna una cadena que contiene una representación imprimible de un objeto. Para muchos tipos, esta función intenta retornar la cadena que retornaría el objeto con el mismo valor cuando es pasado a `eval()`, de lo contrario la representación es una cadena entre corchetes angulares («<>») que contiene el nombre del tipo del objeto junto con información adicional que incluye a menudo el nombre y la dirección del objeto. Una clase puede controlar lo que esta función retorna definiendo un método `__repr__()`.

reversed (*seq*)

Retorna un *iterator* reverso. *seq* debe ser un objeto que tenga un método `__reversed__()` o que soporte el protocolo de secuencia (el método `__len__()` y el método `__getitem__()` con argumentos enteros comenzando en 0).

round (*number* [, *ndigits*])

Retorna *number* redondeado a *ndigits* de precisión después del punto decimal. Si *ndigits* es omitido o es `None`, retorna el entero más cercano a su entrada.

Para los tipos integrados (*built-in*) que soportan `round()`, los valores son redondeados al múltiplo de 10 más cercano a la potencia menos *ndigits*; si dos múltiplos están igual de cerca, el redondeo se hace hacia la opción par (así que por ejemplo tanto `round(0.5)` como `round(-0.5)` son 0, y `round(1.5)` es 2).

Para un objeto `number` general de Python, `round` delega a `number.__round__`.

Nota: El comportamiento de `round()` para flotantes puede ser sorprendente: por ejemplo, `round(2.675, 2)` da `2.67` en vez de los `2.68` esperados. Esto no es un error: es el resultado del hecho de que la mayoría de fracciones decimales no se puede representar de forma exacta como flotantes. Véase [tut-fp-issues](#) para más información.

class `set` (`[iterable]`)

Retorna un nuevo objeto `set`, opcionalmente con elementos tomados de `iterable`. `set` es una clase integrada (*built-in*). Véase [set](#) y [Conjuntos — set, frozenset](#) para documentación sobre esta clase.

Para otros contenedores ver las clases integradas (*built-in*) `frozenset`, `list`, `tuple`, y `dict`, así como el módulo `collections`.

setattr (`object`, `name`, `value`)

Es la función complementaria a `getattr()`. Los argumentos son un objeto, una cadena, y un valor arbitrario. La cadena puede nombrar un atributo existente o uno nuevo. La función asigna el valor al atributo si el objeto lo permite. Por ejemplo, `setattr(x, 'foobar', 123)` es equivalente a `x.foobar = 123`.

class `slice` (`stop`)

class `slice` (`start`, `stop``[, step]`)

Retorna un objeto `slice` que representa el conjunto de índices especificados por `range(start, stop, step)`. Los argumentos `start` y `step` son por defecto `None`. Los objetos `slice` tienen atributos de sólo lectura `start`, `stop` y `step` que simplemente retornan los valores de los argumentos (o sus valores por defecto). Éstos no tienen otra funcionalidad explícita; sin embargo son usados por *Numerical Python* y otras extensiones de terceros. Estos objetos `slices` pueden ser generados también empleando la sintaxis extendida de indexación. Por ejemplo: `a[start:stop:step]` o `a[start:stop, i]`. Véase `itertools.islice()` para la versión alternativa que retorna un iterador.

sorted (`iterable`, `*`, `key=None`, `reverse=False`)

Retorna una nueva lista ordenada a partir de los elementos en `iterable`.

Tiene dos argumentos opcionales que deben ser especificados como argumentos de palabra clave.

`key` especifica una función de un argumento que es empleada para extraer una clave de comparación de cada elemento en `iterable` (por ejemplo, `key=str.lower`). El valor por defecto es `None` (compara los elementos directamente).

`reverse` es un valor boleado. Si está puesto a `True`, entonces la lista de elementos se ordena como si cada comparación fuera reversa.

Puedes usar `functools.cmp_to_key()` para convertir las funciones `cmp` a la antigua usanza en funciones `key`.

La función *built-in* `sorted()` está garantizada en cuanto a su estabilidad. Un ordenamiento es estable si garantiza que no cambia el orden relativo de elementos que resultan igual en la comparación — esto es de gran ayuda para ordenar en múltiples pases (por ejemplo, ordenar por departamento, después por el escalafón de salario).

Para ejemplos de ordenamiento y para un breve tutorial sobre ello, ver [sortinghowto](#).

@staticmethod

Transforma un método en un método estático.

Un método estático no recibe un primer argumento implícito. Para declarar un método estático, utiliza esta expresión:

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

La forma `@staticmethod` es una función *decorator* — ver [function](#) para más detalles.

Un método estático puede ser llamado sobre la clase (como `C.f()`) o sobre una instancia (como `C().f()`).

Los métodos estáticos en Python son similares a los que se encuentran en Java o C++. Ver también `classmethod()` para una variante que es útil para crear constructores de clase alternativos.

Como todos los decoradores, es también posible llamar a `staticmethod` como una función normal y hacer algo con su resultado. Esto es necesario a veces cuando necesitas una referencia a una función desde el cuerpo de una clase y quieres evitar la transformación automática a un método de la instancia. Para dichos casos, emplea esta expresión:

```
class C:
    builtin_open = staticmethod(open)
```

Para más información sobre métodos estáticos, ver `types`.

class str (*object*="")

class str (*object*=`b"`, *encoding*=`'utf-8'`, *errors*=`'strict'`)

Retorna una versión `str` del *object*. Ver `str()` para más detalles.

`str` es la *class* cadena built-in. Para información general sobre strings, ver *Cadenas de caracteres — str*.

sum (*iterable*, */*, *start*=0)

Suma *start* y los elementos de un *iterable* de izquierda a derecha y Retorna el total. Los elementos del *iterable* son normalmente números, y el valor *start* no puede ser una cadena.

Para algunos casos de uso, hay buenas alternativas a `sum()`. La manera preferente y más rápida de concatenar secuencias de cadenas es llamado a `' '.join(sequence)`. Para añadir valores de punto flotante con precisión extendida, ver `math.fsum()`. Para concatenar series de iterables, considera usar `itertools.chain()`.

Distinto en la versión 3.8: El parámetro *start* puede ser especificado como un argumento de palabra clave.

super ([*type*], *object-or-type*)]

Retorna un objeto proxy que delega las llamadas de métodos a clases padre o hermanas de *type*. Esto es útil para acceder métodos heredados que han sido invalidados en una clase.

object-or-type determina el *method resolution order* a ser buscado. La búsqueda empieza desde la clase justo después de *type*.

Por ejemplo `__mro__` de *object-or-type* es `D -> B -> C -> A -> object` y el valor de *type* es `B`, entonces `super()` busca `C -> A -> object`.

El atributo `__mro__` de *object-or-type* lista el orden de búsqueda del método de solución empleado por `getattr()` y `super()`. El atributo es dinámico y puede cambiar en cuanto la jerarquía de herencia se actualiza.

Si se omite el segundo argumento, el objeto `super` retornado no está vinculado. Si el segundo argumento es un objeto, `isinstance(obj, type)` debe ser verdadero. Si el segundo argumento es un tipo, `issubclass(type2, type)` debe ser verdadero (esto es útil para `classmethods`).

Hay dos casos de uso típicos para `super`. En una jerarquía de clases con herencia única, `super` puede ser utilizado para referirse a las clases padre sin llamarlas explícitamente, haciendo el código más sencillo de mantener. Este uso es muy similar al de `super` en otros lenguajes de programación.

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 such implementations 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).

Para ambos casos, la llamada típica de una superclase se parece a:

```
class C(B):
    def method(self, arg):
        super().method(arg)      # This does the same thing as:
                                # super(C, self).method(arg)
```

Además de para los métodos de búsqueda, `super()` también funciona para búsquedas de atributos. Un caso de uso posible para esto es llamar a *descriptores* en una clase padre o hermana.

Nótese que `super()` se implementa como parte del proceso vinculante para búsquedas de atributos explícitos puntuados tales como `super().__getitem__(name)`. Lo hace implementando su propio método `__getattr__()` para buscar clases en un orden predecible que soporte herencia múltiple cooperativa. De la misma manera, `super()` no está definida para búsquedas implícitas usando declaraciones o operadores como `super()[name]`.

Nótese también, que aparte de en su forma sin argumentos, `super()` no está limitada a su uso dentro de métodos. La forma con dos argumentos especifica de forma exacta los argumentos y hace las referencias apropiadas. La forma sin argumentos solo funciona dentro de una definición de clase, ya que el compilador completa los detalles necesarios para obtener correctamente la clase que está siendo definida, así como accediendo a la instancia actual para métodos ordinarios.

Para sugerencias prácticas sobre como diseñar clases cooperativas usando `super()`, véase *guía de uso de super()*.

class tuple ([*iterable*])

Más que una función, `tuple` es en realidad un tipo de secuencia inmutable, tal y como está documentado en *Tuplas y Tipos secuencia — list, tuple, range*.

class type (*object*)

class type (*name, bases, dict, **kwargs*)

Con un argumento, retorna el tipo de un *object*. El valor de retorno es un objeto tipo y generalmente el mismo objeto que el retornado por `object.__class__`.

La función integrada `isinstance()` es la recomendada para testear el tipo de un objeto, ya que tiene en cuenta las subclases.

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 contains the base classes and becomes the `__bases__` attribute; if empty, *object*, the ultimate base of all classes, is added. The *dict* dictionary contains attribute and method definitions for the class body; it may be copied or wrapped before becoming the `__dict__` attribute. The following two statements create identical *type* objects:

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

Ver también *Objetos Tipo*.

Keyword arguments provided to the three argument form are passed to the appropriate metaclass machinery (usually `__init_subclass__()`) in the same way that keywords in a class definition (besides *metaclass*) would.

See also class-customization.

Distinto en la versión 3.6: Subclases de `type` que no sobrecarguen `type.__new__` ya no pueden usar la forma con un argumento para obtener el tipo de un objeto.

vars ([*object*])

Retorna el atributo `__dict__` para un módulo, clase, instancia o cualquier otro objeto con un atributo `__dict__`.

Los objetos como módulos o instancias tienen un atributo actualizable `__dict__`; sin embargo, otros objetos pueden tener restricciones de escritura en sus atributos `__dict__` (por ejemplo, hay clases que usan `types.MappingProxyType` para evitar actualizaciones directas del diccionario).

Sin un argumento, `vars()` funciona como `locals()`. Ten en cuenta que el diccionario de *locals* solo es útil para lecturas ya que las actualizaciones del diccionario de *locals* son ignoradas.

A `TypeError` exception is raised if an object is specified but it doesn't have a `__dict__` attribute (for example, if its class defines the `__slots__` attribute).

zip (*iterables)

Produce un iterador que agrega elementos de cada uno de los iterables.

Retorna un iterador de tuplas, donde el i -ésimo elemento de la tupla contiene el i -ésimo elemento de cada una de las secuencias o iterables en los argumentos. El iterador para cuando se agota el iterable de entrada más corto. Con un sólo argumento iterable, retorna un iterador de 1 tupla. Sin argumentos, retorna un iterador vacío. Es equivalente a:

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

La evaluación de izquierda a derecha de los iterables está garantizada. Esto permite emplear una expresión idiomática para agregar una serie de datos en grupos de tamaño n usando `zip(*[iter(s)]*n)`. Esto repite el *mismo* iterador n veces de forma que cada tupla de salida tiene el resultado de n llamadas al iterador. Esto tiene el efecto de dividir la entrada en trozos de longitud n .

`zip()` solo debería utilizarse con tamaños de entrada diferentes en el caso de que no te importe que haya valores sin agrupar de los iterables más largos. Si en cambio esos valores son importantes, utiliza en cambio `itertools.zip_longest()`.

`zip()` en conjunción con el operador `*` puede usar para descomprimir (*unzip*) una lista:

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

__import__ (name, globals=None, locals=None, fromlist=(), level=0)

Nota: Ésta es una función avanzada que no se necesita en el uso cotidiano de programación en Python, a diferencia de `importlib.import_module()`.

Esta función es invocada por la declaración `import`. Puede ser reemplazada para cambiar la semántica de la declaración `import` (mediante la importación del módulo `builtins` y su asignación a `builtins.__import__`), pero esto está **fuertemente** no recomendado ya que es normalmente mucho más simple usar ganchos (*hooks*) de importación (ver [PEP 302](#)) para obtener los mismos objetivos y sin causar problemas en código que asume que la implementación por defecto de importaciones está siendo utilizada. El uso directo de `__import__()` tampoco está recomendado y se prefiere `importlib.import_module()`.

La función importa el módulo `name`, usando potencialmente las `globals` y `locals` indicadas para determinar como interpretar el nombre en el contexto de un paquete. `fromlist` indica los nombres de objetos o submódulos que deberían ser importados del módulo indicado por `name`. La implementación estándar no utiliza su argumento `locals` para nada, y usa `globals` solo para determinar el contexto en un paquete de la declaración `import`.

`level` especifica si usar importaciones absolutas o relativas. 0 (por defecto) significa sólo realizar importaciones absolutas. Valores positivos de `level` indican el número de directorios progenitores relativos al del módulo para buscar llamando a `__import__()` (ver [PEP 328](#) para los detalles).

Cuando la variable *name* tiene la forma `package.module`, normalmente el paquete del nivel más alto (el nombre hasta el primer punto) se retorna, *no* el modulo llamado por *name*. Sin embargo, cuando un argumento *fromlist* no vacío es indicado, el módulo llamado por *name* es retornado.

Por ejemplo, la declaración `import spam` resulta en un bytecode similar al siguiente código:

```
spam = __import__('spam', globals(), locals(), [], 0)
```

La declaración `import spam.ham` resulta en esta llamada:

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

Nótese cómo `__import__()` retorna el módulo del nivel superior en este caso porque este es el objeto que está enlazado a un nombre por la declaración `import`.

Por otra parte, la declaración `from spam.ham import eggs, sausage as saus` resulta en

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0)
eggs = _temp.eggs
saus = _temp.sausage
```

Aquí, el módulo `spam.ham` es retornado desde `__import__()`. Desde este objeto, los nombres a importar son obtenidos y asignados a sus nombres respectivos.

Si simplemente quieres importar un módulo (potencialmente dentro de un paquete) por nombre, usa `importlib.import_module()`.

Distinto en la versión 3.3: Valores negativos para *level* ya no están soportados (lo que también cambia el valor por defecto a 0).

Notas al pie

Constantes incorporadas

Un pequeño número de constantes viven en el espacio de nombres incorporado. Ellas son:

False

El valor falso del tipo *bool*. Las asignaciones a `False` son ilegales y generan un *SyntaxError*.

True

El valor verdadero del tipo *bool*. Las asignaciones a `True` son ilegales y generan un *SyntaxError*.

None

El único valor del tipo *NoneType*. `None` se utiliza con frecuencia para representar la ausencia de un valor, como cuando los argumentos predeterminados no se pasan a una función. Las asignaciones a `None` son ilegales y generan un *SyntaxError*.

NotImplemented

Valor especial que deben retornar los métodos especiales binarios (por ejemplo, `__eq__()`, `__lt__()`, `__add__()`, `__rsub__()`, etc.) para indicar que la operación no se implementa con respecto al otro tipo; pueden ser retornados por los métodos especiales binarios in situ (por ejemplo: `__imul__()`, `__iand__()`, etc.) para el mismo propósito. Su valor de verdad es verdadero.

Nota: When a binary (or in-place) method returns `NotImplemented` the interpreter will try the reflected operation on the other type (or some other fallback, depending on the operator). If all attempts return `NotImplemented`, the interpreter will raise an appropriate exception. Incorrectly returning `NotImplemented` will result in a misleading error message or the `NotImplemented` value being returned to Python code.

Consulte *Implementar operaciones aritméticas* para ver ejemplos.

Nota: `NotImplementedError` y `NotImplemented` no son lo mismo, aunque tengan nombres y propósitos similares. Consulte *NotImplementedError* para obtener más información sobre cuándo usarlo.

Ellipsis

The same as the ellipsis literal `«...»`. Special value used mostly in conjunction with extended slicing syntax for user-defined container data types.

`__debug__`

Esta constante es verdadera si Python no se inició con una opción `-O`. Vea también la instrucción `assert`.

Nota: Los nombres: *None*, *False*, *True* y `__debug__` no se pueden reasignar (asignaciones a ellos, incluso como un nombre de atributo, lanza *SyntaxError*), por lo que pueden considerarse constantes «verdaderas».

3.1 Constantes agregadas por el módulo `site`

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*)

Objetos que cuando se imprimen, muestra un mensaje como «Use quit() o Ctrl-D (i.e. EOF) to exit», y cuando se llama, lanza *SystemExit* con el código de salida especificado.

copyright

credits

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

license

Objeto que cuando se imprime, muestra el mensaje «Escriba licencia () para ver el texto completo de la licencia», y cuando se le llama, muestra el texto completo de la licencia en forma de buscapersonas (una pantalla a la vez).

Tipos Integrados

Esta sección describe los tipos de datos estándar que vienen incluidos en el intérprete.

Los principales tipos de datos son: numéricos, secuencias, mapas, clases, instancias y excepciones.

Algunas clases de tipo colección son mutables. Los métodos que añaden, retiran u ordenan los contenidos lo hacen internamente, y a no ser que retornen un elemento concreto, nunca retornan la propia instancia contenedora, sino `None`.

Algunas operaciones son soportadas por varios tipos de objetos diferentes; por ejemplo, prácticamente todos los objetos pueden ser comparados por igualdad, evaluados para ser considerados como valores booleanos, o representarse en forma de cadena de caracteres (Ya sea con la función `repr()` o la ligeramente diferente `str()`). Esta última es la usada implícitamente por la función `print()`.

4.1 Evaluar como valor verdadero/falso

Cualquier objeto puede ser evaluado como si fuera un valor verdadero o falso, para ser usado directamente en sentencias `if` o `while`, o como un operador en una operación booleana como las que veremos más adelante.

Por defecto, un objeto se considera verdadero a no ser que su clase defina o bien un método `__bool__()` que retorna `False` o un método `__len__()` que retorna cero, cuando se invoque desde ese objeto.¹ Aquí están listados la mayoría de los objetos predefinidos que se evalúan como falsos:

- constantes definidas para tener valor falso: `None` y `False`.
- cero en cualquiera de los diferentes tipos numéricos: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- cualquier colección o secuencia vacía: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

Las operaciones y funciones predefinidas que retornan como resultado un booleano siempre retornan `0` o `False` para un valor falso, y `1` o `True` para un valor verdadero, a no ser que se indique otra cosa (Hay una excepción importante: Los operadores booleanos `or` y `and` siempre retornan uno de los dos operadores).

¹ Se puede consultar información adicional sobre estos métodos especiales en el Manual de Referencia de Python (customization).

4.2 Operaciones booleanas — and, or, not

Estas son las operaciones booleanas, ordenadas de menor a mayor prioridad:

Operación	Resultado	Notas
<code>x or y</code>	si <code>x</code> es falso, entonces <code>y</code> , si no, <code>x</code>	(1)
<code>x and y</code>	si <code>x</code> es falso, entonces <code>x</code> , si no, <code>y</code>	(2)
<code>not x</code>	si <code>x</code> es falso, entonces <code>True</code> , si no, <code>False</code>	(3)

Notas:

- (1) Este operador usa lógica cortocircuitada, por lo que solo evalúa el segundo argumento si el primero es falso.
- (2) Este operador usa lógica cortocircuitada, por lo que solo evalúa el segundo argumento si el primero es verdadero.
- (3) El operador `not` tiene menos prioridad que los operadores no booleanos, así que `not a == b` se interpreta como `not (a == b)`, y `y ``a == not b` es un error sintáctico.

4.3 Comparaciones

Existen ocho operadores de comparación en Python. Todos comparten el mismo nivel de prioridad (que es mayor que el nivel de las operaciones booleanas). Las comparaciones pueden encadenarse de cualquier manera; por ejemplo, `x < y <= z` equivale a `x < y and y <= z`, excepto porque `y` solo se evalúa una vez (No obstante, en ambos casos `z` no se evalúa si no es verdad que `x < y`).

Esta tabla resume las operaciones de comparación:

Operación	Significado
<code><</code>	estrictamente menor que
<code><=</code>	menor o igual que
<code>></code>	estrictamente mayor que
<code>>=</code>	mayor o igual que
<code>==</code>	igual que
<code>!=</code>	diferente que
<code>is</code>	igualdad a nivel de identidad (Son el mismo objeto)
<code>is not</code>	desigualdad a nivel de identidad (no son el mismo objeto)

Nunca se consideran iguales objetos que son de tipos diferentes, con la excepción de los tipos numéricos. El operador `==` siempre está definido, pero en algunos tipos de objetos (Como por ejemplo, las clases) es equivalente al operador `is`. Los operadores `<`, `<=`, `>` y `>=` solo están definidos cuando tienen sentido; por ejemplo, si uno de los operadores es un número complejo, la comparación elevará una excepción de tipo `TypeError`.

Instancias de una clase que no son idénticas normalmente se consideran como diferentes, a no ser que la clase defina un método `__eq__()`.

Las instancias de una clase no pueden ordenarse con respecto a otras instancias de la misma clase, ni con otro tipo de objetos, a no ser que la clase defina un subconjunto suficiente de estos métodos: `__lt__()`, `__le__()`, `__gt__()` y `__ge__()` (En general, `__lt__()` y `__eq__()` son suficientes, si solo necesitas los significados convencionales de los operadores de comparación).

El comportamiento de los operadores `is` e `is not` no se puede personalizar; además, nunca elevan una excepción, no importa que dos objetos se comparen.

Hay otras dos operaciones con la misma prioridad sintáctica: `in` y `not in`, que son soportadas por aquellos tipos de datos que son de tipo *iterable* o que implementen el método `__contains__()`.

4.4 Tipos numéricos — `int`, `float`, `complex`

Hay tres tipos numéricos distintos: *enteros*, *números en coma flotante* y *números complejos*. Además, los booleanos son un subtipo de los enteros. Los enteros tienen precisión ilimitada. Los números en coma flotante se implementan normalmente usando el tipo `double` de C; Hay más información sobre la precisión y la representación interna de los números en coma flotante usadas por la máquina sobre la que se ejecuta tu programa en `sys.float_info`. Los números complejos tienen una parte real y otra imaginaria, ambas representadas con números en coma flotante. Para extraer estas partes del número complejo `z` se usan los métodos `z.real` y `z.imag`. (La librería estándar incluye tipos numéricos adicionales: `fractions.Fraction` para números racionales y `decimal.Decimal` para números en coma flotante con precisión definida por el usuario).

Los números se crean a partir de una expresión literal, o como resultado de una combinación de funciones predefinidas y operadores. Expresiones literales de números (incluyendo números expresados en hexadecimal, octal o binario) producen enteros. Si la expresión literal contiene un punto decimal o un signo de exponente, se genera un número en coma flotante. Si se añade como sufijo una `'j'` o una `'J'` a un literal numérico, se genera un número imaginario puro (Un número complejo con la parte real a cero), que se puede sumar a un número entero o de coma flotante para obtener un número complejo con parte real e imaginaria.

Python soporta completamente una aritmética mixta: Cuando un operador binario de tipo aritmético se encuentra con que los operadores son de tipos diferentes, el operando con el tipo de dato más «estrecho» o restrictivo se convierte o amplía hasta el nivel del otro operando. Los enteros son más «estrechos» que los de coma flotante, que a su vez son más estrechos que los números complejos. Las comparaciones entre números de diferentes tipos se comportan como si se compararan los valores exactos de estos.²

Las funciones constructoras `int()`, `float()` y `complex()` se pueden usar para generar números de cada tipo determinado.

Todos los tipos numéricos (menos los complejos) soportan las siguientes operaciones (Para las prioridades de las operaciones, véase `operator-summary`):

Operación	Resultado	No-tas	Documentación completa
<code>x + y</code>	suma de <i>x</i> e <i>y</i>		
<code>x - y</code>	resta de <i>x</i> e <i>y</i>		
<code>x * y</code>	multiplicación de <i>x</i> por <i>y</i>		
<code>x / y</code>	división de <i>x</i> por <i>y</i>		
<code>x // y</code>	división entera de <i>x</i> por <i>y</i>	(1)	
<code>x % y</code>	resto o residuo de <i>x</i> por <i>y</i>	(2)	
<code>-x</code>	valor de <i>x</i> , negado		
<code>+x</code>	valor de <i>x</i> , sin cambiar		
<code>abs(x)</code>	valor absoluto de la magnitud de <i>x</i>		<code>abs()</code>
<code>int(x)</code>	valor de <i>x</i> convertido a entero	(3)(6)	<code>int()</code>
<code>float(x)</code>	valor de <i>x</i> convertido a número de punto flotante	(4)(6)	<code>float()</code>
<code>complex(re, im)</code>	un número complejo, con parte real <i>re</i> y parte imaginaria <i>im</i> . El valor de <i>im</i> por defecto vale cero.	(6)	<code>complex()</code>
<code>c.conjugate()</code>	conjugado del número complejo <i>c</i>		
<code>divmod(x, y)</code>	el par de valores (<code>x // y</code> , <code>x % y</code>)	(2)	<code>divmod()</code>
<code>pow(x, y)</code>	<i>x</i> elevado a <i>y</i>	(5)	<code>pow()</code>
<code>x ** y</code>	<i>x</i> elevado a <i>y</i>	(5)	

Notas:

- (1) También conocida como división entera. El resultado es un número entero en el sentido matemático, pero no necesariamente de tipo entero. El resultado se redondea de forma automática hacia menos infinito: `1 // 2` es 0, `(-1) // 2` es -1, `1 // (-2)` es -1 y `(-1) // (-2)` es 0.

² En consecuencia, la lista `[1, 2]` se considera igual que `[1.0, 2.0]`, y de forma similar para las tuplas.

- (2) No es apropiada para números complejos. Es preferible convertir a valores en coma flotante usando la función `abs()` si fuera apropiado.
- (3) Conversiones desde coma flotante a entero pueden redondearse o truncarse como en C; véanse las funciones `math.floor()` y `math.ceil()` para un mayor control.
- (4) float también acepta las cadenas de caracteres «nan» e «inf», con un prefijo opcional «+» o «-», para los valores *Not a Number* (NaN) e infinito positivo o negativo.
- (5) Python define `pow(0, 0)` y `0 ** 0` para que valgan 1, como es práctica habitual en los lenguajes de programación.
- (6) Los literales numéricos aceptables incluyen los dígitos desde el 0 hasta el 9, así como cualquier carácter Unicode equivalente (puntos de código con la propiedad Nd).

En <http://www.unicode.org/Public/12.1.0/ucd/extracted/DerivedNumericType.txt> se puede consultar una lista completa de los puntos de código con la propiedad Nd.

Todas las clases derivadas de `numbers.Real` (`int` y `float`) también soportan las siguientes operaciones:

Operación	Resultado
<code>math.trunc(x)</code>	x truncado a <i>Integral</i>
<code>round(x[, n])</code>	El valor de x redondeado a n dígitos, redondeando la mitad al número par más cercano (Redondeo del banquero). Si no se especifica valor para n , se asume 0.
<code>math.floor(x)</code>	el mayor número <i>Integral</i> que sea $\leq x$
<code>math.ceil(x)</code>	el menor número <i>Integral</i> que sea $\geq x$

Para más operaciones numéricas consulta los módulos `math` y `cmath`.

4.4.1 Operaciones de bits en números enteros

Las operaciones a nivel de bit solo tienen sentido con números enteros. El resultado de una de estas operaciones se calcula como si se hubiera realizado en una representación en complemento a dos que tuviera un número infinito de bits de signo.

La prioridad de todas las operaciones de bits son menores que las operaciones numéricas, pero mayores que las comparaciones; la operación unaria `~` tiene la misma prioridad que las otras operaciones unarias numéricas (`+` y `-`).

Esta tabla lista las operaciones de bits, ordenadas de menor a mayor prioridad:

Operación	Resultado	Notas
<code>x y</code>	la operación <i>or</i> entre x e y	(4)
<code>x ^ y</code>	la operación <i>exclusive or</i> entre x e y	(4)
<code>x & y</code>	la operación <i>and</i> entre x e y	(4)
<code>x << n</code>	El valor x desplazado a la izquierda n bits	(1)(2)
<code>x >> n</code>	valor de x desplazado a la derecha n bits	(1)(3)
<code>~x</code>	invierte los bits de x	

Notas:

- (1) Los desplazamientos negativos son ilegales y elevan una excepción de tipo `ValueError`.
- (2) A left shift by n bits is equivalent to multiplication by `pow(2, n)`.
- (3) A right shift by n bits is equivalent to floor division by `pow(2, n)`.
- (4) Realizar estos cálculos con al menos un bit extra de signo en una representación finita de un número en complemento a dos (Un ancho de bits de trabajo de `1 + max(x.bit_length(), y.bit_length())` o

más) es suficiente para obtener el mismo resultado que si se hubiera realizado con un número infinito de bits de signo.

4.4.2 Métodos adicionales de los enteros

El tipo `int` implementa la *clase base abstracta* `numbers.Integral`. Además, proporciona los siguientes métodos:

`int.bit_length()`

Retorna el número de bits necesarios para representar un número entero, excluyendo el bit de signo y los ceros a la izquierda:

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

De forma más precisa, si x es distinto de cero, entonces `x.bit_length()` es el único número entero positivo k tal que $2^{k-1} \leq \text{abs}(x) < 2^k$. De igual manera, cuando `abs(x)` es lo suficientemente pequeño para tener un logaritmo redondeado correctamente, entonces $k = 1 + \text{int}(\log(\text{abs}(x), 2))$. Si x es cero, entonces `x.bit_length()` retorna 0.

Equivale a:

```
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
```

Nuevo en la versión 3.1.

`int.to_bytes(length, byteorder, *, signed=False)`

Retorna un array de bytes que representan el número entero.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xffc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
b'\xe8\x03'
```

El número entero se representa usando el número de bits indicados con `length`. Se elevará la excepción `OverflowError` si no se puede representar el valor con ese número de bits.

El argumento `byteorder` determina el orden de representación del número entero. Si `byteorder` es "big", el byte más significativo ocupa la primera posición en el vector. Si `byteorder` es "little", el byte más significativo estará en la última posición. Para indicar que queremos usar el ordenamiento propio de la plataforma, podemos usar `sys.byteorder` como valor del argumento.

El parámetro `signed` determina si se usa el complemento a dos para representar los números enteros. Si `signed` es `False`, y se usa un valor entero negativo, se elevará la excepción `OverflowError`. El valor por defecto para `signed` es `False`.

Nuevo en la versión 3.2.

`classmethod int.from_bytes(bytes, byteorder, *, signed=False)`

Retorna el número entero representado por el vector de bytes.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

El argumento *bytes* debe ser o bien un *objeto tipo binario* o un iterable que produzca bytes.

El argumento *byteorder* determina el orden de representación del número entero. Si *byteorder* es "big", el byte más significativo ocupa la primera posición en el vector. Si *byteorder* es "little", el byte más significativo estará en la última posición. Para indicar que queremos usar el ordenamiento propio de la plataforma, podemos usar *sys.byteorder* como valor del argumento.

El argumento *signed* determina si se representará el número entero usando complemento a dos.

Nuevo en la versión 3.2.

`int.as_integer_ratio()`

Retorna una pareja de números enteros cuya proporción es igual a la del número entero original, y con un denominador positivo. En el caso de números enteros, la proporción siempre es el número original y 1 en el denominador.

Nuevo en la versión 3.8.

4.4.3 Métodos adicionales de Float

El tipo float implementa la clase *numbers.Real class base abstracta*. Los números float tienen además los siguientes métodos.

`float.as_integer_ratio()`

Retorna una pareja de números enteros cuya proporción es exactamente igual que la del valor en punto flotante original, con un denominador positivo. Si se llama con valores infinitos eleva una excepción de tipo *OverflowError* y si se llama con NaN (*Not A Number*) eleva una excepción de tipo *ValueError*.

`float.is_integer()`

Retorna True si el valor en coma flotante se puede representar sin pérdida con un número entero, y False si no se puede:

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

Hay dos métodos que convierten desde y hacia cadenas de caracteres en hexadecimal. Como los valores en coma flotante en Python se almacenan internamente en binario, las conversiones desde o hacia cadenas *decimales* pueden implicar un pequeño error de redondeo. Pero con cadenas de texto en hexadecimal, las cadenas se corresponden y permiten representar de forma exacta los números en coma flotante. Esto puede ser útil, ya sea a la hora de depurar errores, o en procesos numéricos.

`float.hex()`

Retorna la representación de un valor en coma flotante en forma de cadena de texto en hexadecimal. Para números finitos, la representación siempre empieza con el prefijo 0x, y con una p justo antes del exponente.

classmethod `float.fromhex(s)`

Método de clase que retorna el valor en coma flotante representado por la cadena de caracteres en hexadecimal en *s*. La cadena *s* puede tener espacios en blanco al principio o al final.

Nótese que `float.hex()` es un método de instancia, mientras que `float.fromhex()` es un método de clase.

Una cadena de caracteres en hexadecimal sigue este formato:

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

donde el componente opcional `sign` puede ser o bien `+` o `-`. Las componentes `integer` y `fraction` son cadenas de caracteres que solo usan dígitos hexadecimales, y `exponent` es un número decimal, precedido con un signo opcional. No se distingue entre mayúsculas y minúsculas, y debe haber al menos un dígito hexadecimal tanto en la parte entera como en la fracción. Esta sintaxis es similar a la sintaxis especificada en la sección 6.4.4.2 del estándar C99, y es también la sintaxis usada en Java desde la versión 1.5. En particular, la salida de `float.hex()` se puede usar como una cadena de caracteres en hexadecimal en código C o Java, y las cadenas de caracteres hexadecimal producidas por el carácter de formato `%a` en C, o por el método Java, `Double.toHexString`, son aceptadas por `float.fromhex()`.

Nótese que el valor del exponente está expresado en decimal, no en hexadecimal, e indica la potencia de 2 por la que debemos multiplicar el coeficiente. Por ejemplo, la cadena de caracteres hexadecimal `0x3.a7p10` representa el número en coma flotante $(3 + 10./16 + 7./16**2) * 2.0**10$, o `3740.0`:

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

Si aplicamos la operación inversa a `3740.0` retorna una cadena de caracteres hexadecimal diferente que, aun así, representa el mismo número:

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

4.4.4 Cálculo del *hash* de tipos numéricos

Para dos números `x` e `y`, posiblemente de tipos diferentes, se requiere que `hash(x) == hash(y)` sea verdadero siempre que `x == y` (Véase la documentación sobre el método `__hash__()` para más detalles). Por razones tanto de eficiencia como de facilidad de implementación entre los tipos numéricos diferentes (Incluyendo `int`, `float`, `decimal.Decimal` y `fractions.Fraction`), el método de *hash* de Python se basa en una función matemática sencilla que está definida para cualquier número racional, con lo cual se puede aplicar a todas las instancias de `int` y `fractions.Fraction`, y a todas las instancias finitas de `float` y `decimal.Decimal`. En esencia, lo que hace esta función es una reducción módulo `P` para un valor fijo del número primo `P`. El valor de `P` está disponible en Python como atributo de `sys.hash_info` con el nombre de `modulus`.

CPython implementation detail: Actualmente, el número primo usado es $P = 2^{31} - 1$ para máquinas de 32 bits, y $P = 2^{61} - 1$ en máquinas de 64 bits.

Aquí están las reglas en detalle:

- Si $x = m / n$ es un número racional no negativo y `n` no es divisible por `P`, se define `hash(x)` como `m * invmod(n, P) % P`, donde `invmod(n, P)` retorna la inversa de `n` módulo `P`.
- Si $x = m / n$ es un número racional no negativo y `n` es divisible por `P` (Pero no así `m`), entonces `n` no tiene módulo inverso de `P` y no se puede aplicar la regla anterior; en este caso, `hash(x)` retorna el valor constante definido en `sys.hash_info.inf`.
- Si $x = m / n$ es un número racional negativo se define `hash(x)` como `-hash(x)`. Si el resultado fuera `-1`, lo cambia por `-2`.
- Los valores concretos `sys.hash_info.inf`, `-sys.hash_info.inf` y `sys.hash_info.nan` se usan como valores *hash* de infinito positivo, infinito negativo y *NaN* (*Not a Number*), respectivamente. (Todos los valores *NaN* comparten el mismo valor de *hash*).
- Para un número complejo `z` (Una instancia de la clase `complex`), el valor de *hash* se calcula combinando los valores de *hash* de la parte real e imaginaria, usando la fórmula `hash(z.real) + sys.hash_info.imag * hash(z.imag)`, módulo reducido `2**sys.hash_info.width`, de forma que el valor ob-

tenido esté en en rango `range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1))`. De nuevo, si el resultado fuera `-1`, se reemplaza por `-2`.

Para clarificar las reglas previas, aquí mostramos un ejemplo de código Python, equivalente al cálculo realizado en la función `hash`, para calcular el `hash` de un número racional, de tipo `float`, o `complex`:

```
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # Remove common factors of P. (Unnecessary if m and n already coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_value = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        # pow(n, P-2, P) gives the inverse of n modulo P.
        hash_value = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_value = -hash_value
    if hash_value == -1:
        hash_value = -2
    return hash_value

def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return sys.hash_info.nan
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_value = hash_float(z.real) + sys.hash_info.imag * hash_float(z.imag)
    # do a signed reduction modulo 2**sys.hash_info.width
    M = 2**(sys.hash_info.width - 1)
    hash_value = (hash_value & (M - 1)) - (hash_value & M)
    if hash_value == -1:
        hash_value = -2
    return hash_value
```

4.5 Tipos de iteradores

Python soporta el concepto de iteradores sobre contenedores. Esto se implementa usando dos métodos diferentes: Estos son usados por las clases definidas por el usuario para soportar iteración. Las secuencias, que se describirán con mayor detalle, siempre soportan la iteración.

Para que un objeto contenedor soporte iteración, debe definir un método:

`container.__iter__()`

Retorna un objeto iterador. Este objeto es requerido para soportar el protocolo de iteración que se describe a continuación. Si un contenedor soporta diferentes tipos de iteración, se pueden implementar métodos adicionales para estos iteradores (por ejemplo, un tipo de contenedor que puede soportar distintas formas de iteración podría ser una estructura de tipo árbol que proporcione a la vez un recorrido en profundidad o en anchura). Este método se corresponde al *slot* `tp_iter` de la estructura usada para los objetos Python en la API Python/C.

Los objetos iteradores en si necesitan definir los siguientes dos métodos, que forma juntos el *protocolo iterador*:

`iterator.__iter__()`

Retorna el propio objeto iterador. Este método es necesario para permitir tanto a los contenedores como a los iteradores usar la palabras clave `for` o `in`. Este método se corresponde con el *slot* `tp_iter` de la estructura usada para los objetos Python en la API Python/C.

`iterator.__next__()`

Retorna el siguiente elemento del contenedor. Si no hubiera más elementos, eleva la excepción *StopIteration*. Este método se corresponde con el *slot* `tp_iternext` de la estructura usada para los objetos Python en la API Python/C.

Python define varios objetos iteradores que permiten iterar sobre las secuencias, ya sean generales o específicas, diccionarios y otras estructuras de datos especializadas. Los tipos específicos no son tan importantes como la implementación del protocolo iterador.

Una vez que la ejecución del método `__next__()` eleva la excepción *StopIteration*, debe continuar haciéndolo en subsiguientes llamadas al método. Si una implementación no cumple esto, se considera rota.

4.5.1 Tipos Generador

Los *generator* de Python proporcionan una manera cómoda de implementar el protocolo iterador. Si un objeto de tipo contenedor implementa el método `__iter__()` como un generador, de forma automática este retornará un objeto iterador (Técnicamente, un objeto generador) que implementa los métodos `__iter__()` y `__next__()`. Se puede obtener más información acerca de los generadores en la documentación de la expresión `yield`.

4.6 Tipos secuencia — list, tuple, range

Hay tres tipos básicos de secuencia: listas, tuplas y objetos de tipo rango. Existen tipos de secuencia especiales usados para el procesamiento de *datos binarios* y *cadenas de caracteres* que se describirán en secciones específicas.

4.6.1 Operaciones comunes de las secuencias

Las operaciones de la siguiente tabla están soportadas por la mayoría de los tipos secuencia, tanto mutables como inmutables. La clase `collections.abc.Sequence` se incluye para facilitar la implementación correcta de estas operaciones en nuestros propios tipos de secuencias.

La tabla lista las operaciones ordenadas de menor a mayor prioridad. En la tabla, *s* y *t* representan secuencias del mismo tipo, *n*, *i*, *j* y *k* son números enteros y *x* es un objeto arbitrario que cumple con cualquier restricción de tipo o valor impuesta por *s*.

Las operaciones `in` y `not in` tienen la misma prioridad que los operadores de comparación. Las operaciones `+` (Concatenación) y `*` (Repetición) tienen la misma prioridad que sus equivalentes numéricos³

Operación	Resultado	No-tas
<code>x in s</code>	True si un elemento de <i>s</i> es igual a <i>x</i> , False en caso contrario	(1)
<code>x not in s</code>	False si un elemento de <i>s</i> es igual a <i>x</i> , True en caso contrario	(1)
<code>s + t</code>	la concatenación de <i>s</i> y <i>t</i>	(6)(7)
<code>s * non * s</code>	equivale a concatenar <i>s</i> consigo mismo <i>n</i> veces	(2)(7)
<code>s[i]</code>	El elemento <i>i</i> -ésimo de <i>s</i> , empezando a contar en 0	(3)
<code>s[i:j]</code>	la rebanada de <i>s</i> desde <i>i</i> hasta <i>j</i>	(3)(4)
<code>s[i:j:k]</code>	la rebanada de <i>s</i> desde <i>i</i> hasta <i>j</i> , con paso <i>j</i>	(3)(5)
<code>len(s)</code>	longitud de <i>s</i>	
<code>min(s)</code>	el elemento más pequeño de <i>s</i>	
<code>max(s)</code>	el elemento más grande de <i>s</i>	
<code>s.index(x[, i[, j]])</code>	índice de la primera ocurrencia de <i>x</i> en <i>s</i> (en la posición <i>i</i> o superior, y antes de <i>j</i>)	(8)
<code>s.count(x)</code>	número total de ocurrencias de <i>x</i> en <i>s</i>	

También se pueden comparar secuencias del mismo tipo. En particular, las tuplas y las listas se comparan por orden lexicográfico, comparando los elementos en la misma posición. Esto significa que, para que se consideren iguales, todos los elementos correspondientes deben ser iguales entre sí, y las dos secuencias deben ser del mismo tipo y de la misma longitud (Para más detalles, véase `comparisons` en la referencia del lenguaje).

Notas:

- (1) Aunque las operaciones `in` y `not in` se usan generalmente para comprobar si un elemento está dentro de un contenedor, en algunas secuencias especializadas (Como `str`, `bytes` y `bytearray`) también se pueden usar para comprobar si está incluida una secuencia:

```
>>> "gg" in "eggs"
True
```

- (2) Valores de *n* menores que 0 se consideran como 0 (Que produce una secuencia vacía del mismo tipo que *s*). Nótese que los elementos de la secuencia *s* no se copian, sino que se referencian múltiples veces. Esto a menudo confunde a programadores noveles de Python; considérese:

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

Lo que ha pasado es que `[[]]` es una lista de un elemento, siendo este elemento una lista vacía, así que los tres elementos de `[[]] * 3` son referencias a la misma lista vacía. Modificar cualquiera de los elementos de `lists` modifica la lista inicial. Para crear una lista de listas independientes entre sí, se puede hacer:

³ Deben haberlo hecho, ya que el analizador no puede decir el tipo de operandos.

```

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

```

Se puede consultar una explicación más completa en esta entrada de la lista de preguntas más frecuentes [faq-multidimensional-list](#).

- (3) Si i o j es negativo, el índice es relativo al final de la secuencia s : Se realiza la sustitución $\text{len}(s) + i$ o $\text{len}(s) + j$. Nótese que -0 sigue siendo 0 .
- (4) La rebanada de s desde i a j se define como la secuencia de elementos con índice k , de forma que $i \leq k < j$. Si i o j es mayor que $\text{len}(s)$ se usa $\text{len}(s)$. Si i se omite o es `None`, se usa 0 . Si j se omite o es `None`, se usa $\text{len}(s)$. Si i es mayor o igual a j , la rebanada estaría vacía.
- (5) La rebanada de s , desde i hasta j con paso k , se define como la secuencia de elementos con índice $x = i + n*k$ tal que $0 \leq n < (j-i)/k$. En otras palabras, los índices son $i, i+k, i+2*k, i+3*k$ y así consecutivamente, hasta que se alcance el valor de j (Pero sin incluir nunca j). Cuando k es positivo, i y j se limitan al valor de $\text{len}(s)$, si fueran mayores. Si k es negativo, i y j se reducen de $\text{len}(s) - 1$. Si i o j se omiten o su valor es `None`, se convierten en valores «finales» (Donde el sentido de final depende del signo de k). Nótese que k no puede valer 0 . Si k vale `None`, se considera como 1 .
- (6) La concatenación de secuencias inmutables siempre produce un nuevo objeto. Esto significa que construir una secuencia usando la concatenación tiene un coste en ejecución cuadrático respecto al tamaño de la secuencia final. Para obtener un rendimiento lineal, se puede optar por una de las alternativas siguientes:
 - en vez de concatenar objetos de tipo `str`, se puede construir una lista y usar finalmente el método `str.join()`, o bien utilizar una instancia de la clase `io.StringIO` y recuperar el valor final completo
 - de forma similar, en vez de concatenar objetos de tipo `bytes` se puede usar el método `bytes.join()`, la clase `io.BytesIO`, o se puede realizar una modificación interna usando un objeto de la clase `bytearray`. Los objetos de tipo `bytearray` son mutables y tienen un mecanismo interno de gestión muy eficiente
 - en vez de concatenar tuplas (Instancias de `tuple`), usar una lista (`list`) y expandirla
 - para otros tipos, investiga la documentación relevante de la clase
- (7) Algunos tipos de secuencia (como la clase `range`) solo soportan elementos que siguen un patrón específico, y por tanto no soportan la concatenación ni la repetición.
- (8) El método `index` eleva la excepción `ValueError` si x no se encuentra en s . No todas las implementaciones soportan los parámetros opcionales i y j . Estos parámetros permiten una búsqueda eficiente de partes de una secuencia. Usar estos parámetros es más o menos equivalente a usar `s[i:j].index(x)`, pero sin copiar ningún dato y con el valor de índice retornado como valor relativo al inicio de la secuencia, en vez de al inicio de la rebanada.

4.6.2 Tipos de secuencia inmutables

La única operación que las secuencias inmutables implementan generalmente, y que no está definida también en las secuencias mutables, es el soporte para el cálculo de la función predefinida `hash()`.

Este soporte permite usar secuencias inmutables, como por ejemplo las instancias de la clase `tuple`, como claves para diccionarios (`dict`), así como ser almacenadas en conjuntos (`set`) o conjuntos congelados (`frozenset`).

Intentar calcular el `hash` de una secuencia inmutable que contenga objetos mutables producirá una excepción de tipo `TypeError`.

4.6.3 Tipos de secuencia mutables

Las operaciones de la siguiente tabla están definidas para todos los tipos de secuencia mutables. La clase `ABC collections.abc.MutableSequence` se incluye para facilitar la implementación correcta de un tipo de secuencia propio.

En la tabla, s es una instancia de una secuencia de tipo mutable, t es cualquier objeto iterable y x es un objeto arbitrario que cumple las restricciones de tipo y valor que vengan impuestas por s (Como ejemplo, la clase `bytearray` solo acepta enteros que cumplan la condición $0 \leq x \leq 255$).

Operación	Resultado	No-tas
<code>s[i] = x</code>	el elemento i de s es reemplazado por x	
<code>s[i:j] = t</code>	la rebanada de valores de s que van de i a j es reemplazada por el contenido del iterador t	
<code>del s[i:j]</code>	equivalente a <code>s[i:j] = []</code>	
<code>s[i:j:k] = t</code>	los elementos de <code>s[i:j:k]</code> son reemplazados por los elementos de t	(1)
<code>del s[i:j:k]</code>	borra los elementos de <code>s[i:j:k]</code> de la lista	
<code>s.append(x)</code>	añade x al final de la secuencia (Equivalente a <code>s[len(s):len(s)] = [x]</code>)	
<code>s.clear()</code>	elimina todos los elementos de s (Equivalente a <code>del s[:]</code>)	(5)
<code>s.copy()</code>	crea una copia superficial de s (Equivalente a <code>s[:]</code>)	(5)
<code>s.extend(t)</code> o <code>s += t</code>	extiende s con los contenidos de t (En la mayoría de los casos equivale a <code>s[len(s):len(s)] = t</code>)	
<code>s *= n</code>	actualiza s con su contenido repetido n veces	(6)
<code>s.insert(i, x)</code>	inserta x en s en la posición indicada por el índice i (Equivalente a <code>s[i:i] = [x]</code>)	
<code>s.pop()</code> or <code>s.pop(i)</code>	retorna el elemento en la posición indicada por i , y a la vez lo elimina de la secuencia s	(2)
<code>s.remove(x)</code>	elimina el primer elemento de s tal que <code>s[i]</code> sea igual a x	(3)
<code>s.reverse()</code>	invierte el orden de los elementos de s , a nivel interno	(4)

Notas:

- (1) La secuencia t debe tener la misma longitud que la rebanada a la que reemplaza.
- (2) El parámetro opcional i tiene un valor por defecto de -1 , así que si no se especifica se retorna el último valor y este se elimina de la secuencia.
- (3) El método `remove()` eleva la excepción `ValueError` cuando no se encuentra x en s .
- (4) El método `reverse()` modifica la secuencia internamente, por motivos de eficiencia espacial para secuencias muy grandes. Como recordatorio al usuario de que el método produce un efecto secundario, no se retorna la secuencia invertida.
- (5) Ambos métodos `clear()` y `copy()` se incluyen por consistencia con las interfaces de clases que no soportan operaciones de rebanado (Como las clases `dict` y `set`). El método `copy()` no es parte de la clase `ABC collections.abc.MutableSequence`, pero la mayoría de las clases finales de tipo secuencia mutable lo incluyen.

Nuevo en la versión 3.3: Los métodos `clear()` y `copy()`.

- (6) El valor de n es un entero, o un objeto que implemente el método `__index__()`. Los valores negativos, junto con el cero, producen una lista vacía. Los elementos de la secuencia no son copiados, sino que se referencian múltiples veces, tal y como se explicó para `s * n` en *Operaciones comunes de las secuencias*.

4.6.4 Listas

Las listas son secuencia mutables, usadas normalmente para almacenar colecciones de elementos homogéneos (Donde el grado de similitud de los mismo depende de la aplicación).

class `list` (`[iterable]`)

Las listas se pueden construir de diferentes formas:

- Usando un par de corchetes para definir una lista vacía: `[]`
- Usando corchetes, separando los elementos incluidos con comas: `[a], [a, b, c]`
- Usando una lista intensiva o por comprensión: `[x for x in iterable]`
- Usando el constructor de tipo: `list()` o `list(iterable)`

La lista se construye con los mismos elementos y en el mismo orden que *iterable*, donde *iterable* puede ser una secuencia, un contenedor que soporta iteración, o un objeto iterador. Si *iterable* es de por sí una lista, se construye y retorna una copia, como si se hubiera llamado a `iterable[:]`. Por ejemplo, `list('abc')` retorna `['a', 'b', 'c']` y `list((1, 2, 3))` retorna `[1, 2, 3]`. Si no se pasan parámetros, se construye una nueva lista vacía, `[]`.

Muchas otras operaciones también producen listas, incluyendo la función básica `sorted()`.

Las listas implementan todas las operaciones *comunes* y *mutables* propias de las secuencias. Además, las listas incorporan los siguientes métodos:

sort (`*`, `key=None`, `reverse=False`)

Este método ordena la lista *in situ* (se modifica internamente), usando únicamente comparaciones de tipo `<`. Las excepciones no son capturadas internamente: si alguna comparación falla, la operación entera de ordenación falla (Y la lista probablemente haya quedado modificada parcialmente).

El método `sort()` acepta dos parámetros, que solo pueden pasarse por nombre (*keyword-only arguments*):

El parámetro `key` especifica una función de un argumento que se usa para obtener, para cada elemento de la lista, un valor concreto o clave (*key*) a usar en las operaciones de comparación (Por ejemplo, `key=str.lower`). El elemento clave para cada elemento se calcula una única vez y se reutiliza para todo el proceso de ordenamiento. El valor por defecto, `None`, hace que la lista se ordene comparando directamente los elementos, sin obtener valores clave.

La utilidad `functools.cmp_to_key()` se puede usar para convertir una función *cmp* del estilo de la versión 2.x a una función *key*.

El valor de `reverse` es un valor booleano. Si se define como `True`, entonces los elementos de la lista se ordenan como si las operaciones de comparación se hubiesen invertido.

Este método modifica la lista *in situ*, para ahorrar espacio cuando se ordena una secuencia muy grande. Para recordar a los usuarios que funciona de esta manera, no se retorna la secuencia ordenada (Puedes usar `sorted()` para obtener de forma explícita una nueva secuencia ordenada).

El método `sort()` es estable. Un algoritmo de ordenación es estable si garantiza que no se cambia el orden relativo que mantienen inicialmente los elementos que se consideran iguales — Esto es útil para realizar ordenaciones en múltiples fases (Por ejemplo, ordenar por departamento y después por salario).

Para ver ejemplos de ordenación y un breve tutorial sobre el tema, véase `sortinghowto`.

CPython implementation detail: Mientras una lista está siendo ordenada, los efectos de intentar modificarla, o incluso examinarla, no están definidos. La implementación en C de Python hace que la lista parezca vacía durante la ordenación, y eleva una excepción del tipo `ValueError` si detecta un cambio en la lista durante el proceso de ordenación.

4.6.5 Tuplas

Las tuplas son secuencias inmutables, usadas normalmente para almacenar colecciones de datos heterogéneos (Como las duplas o tuplas de dos elementos producidas por la función básica `enumerate()`). También son usadas en aquellos casos donde se necesite una secuencia inmutable de datos heterogéneos (Como por ejemplo permitir el almacenamiento en un objeto de tipo `set` o `dict`).

class tuple ([*iterable*])

Las tuplas se pueden construir de diferentes maneras:

- Usando un par de símbolos de paréntesis, para indicar una tupla vacía: `()`
- Usando una coma al final, para crear una tupla de un único elemento: `a`, o `(a,)`
- Separando los elementos por comas: `a, b, c` o `(a, b, c)`
- Usando la función básica `tuple()` built-in: `tuple()` o `tuple(iterable)`

El constructor genera una tupla cuyos elementos son los mismos y están en el mismo orden que los elementos del *iterable*, donde *iterable* puede ser una secuencia, un contenedor que soporta iteración, o un objeto de tipo *iterator*. Si *iterable* es ya de por sí una tupla, se retorna sin cambiar. Por ejemplo, `tuple('abc')` retorna `('a', 'b', 'c')` y `tuple([1, 2, 3])` retorna `(1, 2, 3)`. Si no se indica ningún parámetro, el constructor creará una nueva tupla vacía. `()`.

Nótese que es la coma la que realmente construye la tupla, no los paréntesis. Los paréntesis son opcionales, excepto en el caso de la tupla vacía, o cuando se necesitan para evitar una ambigüedad sintáctica. Por ejemplo, `f(a, b, c)` es una llamada a una función con tres parámetros, pero `f((a, b, c))` es una llamada a una función con un único parámetro, en este caso una tupla de tres elementos.

Las tuplas implementan todas las operaciones de secuencia *common*.

Para colecciones de datos heterogéneos donde el acceso por nombre resulta más claro que por índice, quizá crear una tupla con nombres (`collections.namedtuple()`) pueden ser más apropiado.

4.6.6 Rangos

Los objetos de tipo `range` representan una secuencia inmutable de números y se usan habitualmente para ejecutar un bucle `for` un número determinado de veces.

class range (*stop*)

class range (*start*, *stop* [, *step*])

Los parámetros usados por el constructor del rango deben ser números enteros (O bien objetos de tipo `int` o instancias de una clase que implemente el método especial `__index__`). Si el parámetro *step* se omite, se asume el valor 1. Si se omite el parámetro *start*, se toma como 0. Si se intenta usar 0 como valor de *step*, se eleva una excepción de tipo `ValueError`.

Para un valor positivo de *step*, el contenido del rango *r* viene determinado por la fórmula `r[i] = start + step*i` donde `i >= 0` y `r[i] < stop`.

Para un valor negativo de *step*, el contenido del rango sigue estando determinado por la fórmula `r[i] = start + step*i`, pero las restricciones ahora son `i >= 0` y `r[i] > stop`.

Un objeto de tipo rango se considera vacío si `r[0]` no cumple con las restricciones de valor. Los rangos soportan índices negativos, pero estos son interpretados como índices considerados desde el final de la secuencia determinada por los índices positivos.

Los rangos que contengan valores mayores que `sys.maxsize` se permiten, pero algunas capacidades (como la función `len()`) pueden elevar una excepción de tipo `OverflowError`.

Ejemplos de rangos:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
```

(continúe en la próxima página)

(proviene de la página anterior)

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

Los rangos implementan todas las operaciones *comunes* de las secuencias, excepto la concatenación y la repetición (Esto es porque los objetos de tipo rango solamente pueden representar secuencias que siguen un patrón estricto, y tanto la repetición como la concatenación pueden romperlo).

start

El valor del parámetro `start` (0 si no se utiliza el parámetro)

stop

El valor del parámetro `stop`

step

El valor del parámetro `step` (1 si no se utiliza el parámetro)

La ventaja de usar un objeto de tipo `range` en vez de uno de tipo `list` o `tuple` es que con `range` siempre se usa una cantidad fija (y pequeña) de memoria, independientemente del rango que represente (Ya que solamente necesita almacenar los valores para `start`, `stop` y `step`, y calcula los valores intermedios a medida que los va necesitando).

Los objetos rango implementan la clase ABC `collections.abc.Sequence`, y proporcionan capacidades como comprobación de inclusión, búsqueda de elementos por índice, operaciones de rebanadas y soporte de índices negativos (Véase *Tipos secuencia — list, tuple, range*):

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

La comparación entre rangos usando los operadores `==` y `!=` se realiza como con las secuencias. Esto es, dos rangos se consideran iguales si representan exactamente la misma secuencia de elementos. (Fíjate que, según esta definición, dos rangos pueden considerarse iguales aunque tengan diferentes valores para `start`, `stop` y `step`, por ejemplo `range(0) == range(2, 1, 3)` y `range(0, 3, 2) == range(0, 4, 2)`).

Distinto en la versión 3.2: Implementa la clase abstracta `Sequence`. Soportan operaciones de rebanado e índices negativos. Comprobar si un entero de tipo `int` está incluido en un rango se realiza en un tiempo constante, no se realiza una iteración a través de todos los elementos.

Distinto en la versión 3.3: Define los operadores `==` y `!=` para comparar rangos en base a la secuencia de valores que definen (En vez de compararse en base a la identidad).

Nuevo en la versión 3.3: Los atributos `start`, `stop` y `step`.

Ver también:

- En [linspace recipe](#) se muestra como implementar una versión *lazy* o perezosa de una función para `range` que funciona con valores en coma flotante.

4.7 Cadenas de caracteres — `str`

La información textual se representa en Python con objetos de tipo `str`, normalmente llamados cadenas de caracteres o simplemente *cadenas*. Las cadenas de caracteres son *secuencias* inmutables de puntos de código Unicode. Las cadenas se pueden definir de diferentes maneras:

- Comillas simples: `'permite incluir comillas "dobles"'`
- Comillas dobles: `"permite incluir comillas 'simples'"`.
- Triples comillas: ya sea con comillas simples `'''Triples comillas simples'''` o dobles `"""Triples comillas dobles"""`

Las cadenas definidas con comillas triples pueden incluir varias líneas. Todos los espacios en blancos incluidos se incorporan a la cadena de forma literal.

Cadenas literales que forman parte de una expresión y que solo estén separados por espacios en blanco, se convertirán implícitamente a una única cadena. Esto es, `("spam " "eggs") == "spam eggs"`.

Véase `strings` para más información acerca de las diferentes formas de expresar cadenas de forma literal, incluidos los caracteres de escape, y del prefijo `r` (*raw*) que deshabilita el procesamiento de la mayoría de dichas secuencias de escape.

Las cadenas de caracteres también se pueden crear usando el constructor `str`.

Como no hay un tipo separado para los caracteres, indexar una cadena produce una cadena de longitud 1. Esto es, para una cadena de caracteres no vacía `s`, `s[0] == s[0:1]`.

Tampoco hay una versión mutable de las cadenas de caracteres, pero el método `str.join()` o la clase `io.StringIO` pueden usarse para construir de forma eficiente una cadena de caracteres a partir de fragmentos.

Distinto en la versión 3.3: Para facilitar la compatibilidad hacia atrás con la versión 2, el prefijo `u` se permite en las cadenas de caracteres. No tiene ningún efecto en la interpretación del literal y no se puede combinar con el prefijo `r`.

class `str` (*object*=")

class `str` (*object*=`b"`, *encoding*=`'utf-8'`, *errors*=`'strict'`)

Retorna una representación en forma de *cadena de caracteres* de *object*. Si no se proporciona ningún valor, retorna una cadena vacía. Si se proporciona, el comportamiento de `str()` depende de los valores pasados en los parámetros *encoding* y *errors*, como veremos.

Si no se especifica ni *encoding* ni *errors*, `str(object)` retorna `object.__str__()`, que es la representación «informal» o mas cómoda de usar, en forma de cadena de caracteres, del valor de *object*. Para una cadena de caracteres, es la cadena en sí. Si *object* no tiene un método `__str__()`, entonces `str()` usará como reemplazo el método `repr(object)`.

Si se indica alguno de los dos parámetros *encoding* o *errors*, entonces *object* debe ser un objeto binario o similar (*bytes-like object*, es decir, una instancia de `bytes` o `bytearray`). En este caso, si *object* es de tipo `bytes` o `bytearray`, la llamada a `str(bytes, encoding, errors)` es equivalente a `bytes.decode(encoding, errors)`. Si no, el objeto de tipo `bytes` que esta subyacente en el objeto *buffer* se obtiene mediante una llamada a `bytes.decode()`. Véase *Tipos de secuencias binarias — bytes, bytearray y memoryview* y *bufferobjects* para más información sobre los objetos *buffer*.

Si se pasa un objeto de tipo `bytes` a la función `str()` sin especificar o bien el parámetro *encoding* o bien el *errors*, se vuelve al caso normal donde se retorna la representación informal (Véase también la `-b` de las opciones de línea de órdenes de Python). Por ejemplo:

```
>>> str(b'Zoot!')
"b'Zoot!'"
```

Para más información sobre la clase `str` y sus métodos, consulta [Cadenas de caracteres — `str`](#) y la sección [Métodos de las cadenas de caracteres](#) a continuación. Para las opciones de formateo de cadenas, lee las secciones [f-strings](#) y [Sintaxis de formateo de cadena](#). También puedes consultar la sección [Servicios de procesamiento de texto](#).

4.7.1 Métodos de las cadenas de caracteres

Todas las cadenas de caracteres implementan las operaciones [comunes](#) de las secuencias, junto con los métodos descritos a continuación.

Las cadenas soportan dos estilos de formateo, uno proporciona un grado muy completo de flexibilidad y personalización (Véase `str.format()`, [Sintaxis de formateo de cadena](#) y [Formato de cadena de caracteres personalizado](#)) mientras que el otro se basa en la función C `printf`, que soporta un menor número de tipos y es ligeramente más complicada de usar, pero es a menudo más rápida para los casos que puede manejar ([Formateo de cadenas al estilo `*printf*`](#)).

La sección [Servicios de procesamiento de texto](#) de la librería estándar cubre una serie de módulos que proporcionan varias utilidades para trabajar con textos (Incluyendo las expresiones regulares en el módulo `re`).

`str.capitalize()`

Retorna una copia de la cadena con el primer carácter en mayúsculas y el resto en minúsculas.

Distinto en la versión 3.8: El primer carácter se pasa ahora a título, más que a mayúsculas. Esto significa que caracteres como dígrafos solo tendrán la primera letra en mayúsculas, en vez de todo el carácter.

`str.casefold()`

Retorna el texto de la cadena, normalizado a minúsculas. Los textos así normalizados pueden usarse para realizar búsquedas textuales independientes de mayúsculas y minúsculas.

El texto normalizado a minúsculas es más agresivo que el texto en minúsculas normal, porque se intenta unificar todas las grafías distintas de la letras. Por ejemplo, En Alemán la letra minúscula 'ß' equivale a "ss". Como ya está en minúsculas, el método `lower()` no modifica 'ß', pero el método `casefold()` lo convertirá a "ss".

El algoritmo de normalización a minúsculas se describe en la sección 3.13 del estándar Unicode.

Nuevo en la versión 3.3.

`str.center(width[, fillchar])`

Retorna el texto de la cadena, centrado en una cadena de longitud `width`. El relleno a izquierda y derecha se realiza usando el carácter definido por el parámetro `fillchar` (Por defecto se usa el carácter espacio ASCII). Si la cadena original tiene una longitud `len(s)` igual o superior a `width`, se retorna el texto sin modificar.

`str.count(sub[, start[, end]])`

Retorna el número de ocurrencias no solapadas de la cadena `sub` en el rango `[start, end]`. Los parámetros opcionales `start` y `end` Se interpretan como en una expresión de rebanada.

`str.encode(encoding="utf-8", errors="strict")`

Retorna una versión codificada en forma de bytes. La codificación por defecto es 'utf-8'. El parámetro `errors` permite especificar diferentes esquemas de gestión de errores. El valor por defecto de `errors` es 'strict', que significa que cualquier error en la codificación eleva una excepción de tipo `UnicodeError`. Otros valores posibles son 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' y cualquier otro nombre que se haya registrado mediante la función `codecs.register_error()`, véase la sección [Manejadores de errores](#). Para una lista de los posibles sistemas de codificación, véase la sección [Codificaciones estándar](#).

Distinto en la versión 3.1: Añade soporte para el uso de parámetros por nombre.

`str.endswith(suffix[, start[, end]])`

Retorna `True` si la cadena termina con el sufijo especificado con el parámetro `prefix`, y `False` en caso contrario. También podemos usar `suffix` para pasar una tupla de sufijos a buscar. Si especificamos el parámetro opcional `start`, la comprobación empieza en esa posición. Con el parámetro opcional `stop`, la comprobación termina en esa posición.

`str.expandtabs (tabsize=8)`

Retorna una copia de la cadena, con todos los caracteres de tipo tabulador reemplazados por uno o más espacios, dependiendo de la columna actual y del tamaño definido para el tabulador. Las posiciones de tabulación ocurren cada *tabsize* caracteres (Siendo el valor por defecto de *tabsize* 8, lo que produce las posiciones de tabulación 0, 8, 16,...). Para expandir la cadena, la columna actual se pone a cero y se va examinando el texto carácter a carácter. Si se encuentra un tabulador, (`\t`), se insertan uno o más espacios hasta que sea igual a la siguiente posición de tabulación (El carácter tabulador en sí es descartado). Si el carácter es un indicador de salto de línea (`\n`) o de retorno (`\r`), se copia y el valor de columna actual se vuelve a poner a cero. Cualquier otro carácter es copiado sin cambios y hace que el contador de columna se incremente en 1, sin tener en cuenta como se representa gráficamente el carácter.

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

`str.find (sub[, start[, end]])`

Retorna el menor índice de la cadena *s* donde se puede encontrar la cadena *sub*, considerando solo el intervalo *s[start:end]*. Los parámetros opcionales *start* y *end* se interpretan como si fueran “índices de una rebanada. retorna -1 si no se encuentra la cadena.

Nota: El método `find()` se debe usar solo si se necesita saber la posición de la cadena *sub*. Si solo se necesita comprobar si *sub* es una parte de *s*, es mejor usar el operador `in`:

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

`str.format (*args, **kwargs)`

Realiza una operación de formateo. La cadena de caracteres sobre la que se está ejecutando este método puede contener texto literal y también marcas de reemplazo de texto definidas mediante llaves `{}`. Cada sección a reemplazar contiene o bien un índice numérico que hace referencia a un parámetro por posición, o el nombre de un parámetro por nombre. retorna una copia de la cadena donde se han sustituido las marcas de reemplazo por los valores correspondientes pasados como parámetros.

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

Véase *Sintaxis de formateo de cadena* para una descripción de las distintas opciones de formateo que se pueden usar.

Nota: Cuando se formatea un número (*int*, *float*, *complex*, *decimal.Decimal* y clases derivadas) usando la *n* (Por ejemplo, `'{:n}'.format(1234)`), la función ajusta temporalmente el valor de la variable de entorno local `LC_TYPE` a `LC_NUMERIC` para decodificar los campos `*decimal_point*` y `*thousands_sep*` de la función `localeconv()`, si usan caracteres que no son ASCII o si ocupan más de un byte, y el valor definido en `LC_NUMERIC` es diferente del definido en `LC_CTYPE`. Estos cambios temporales pueden afectar a otros *threads*.

Distinto en la versión 3.7: Cuando se formatea un número usando la *n*, la función puede asignar de forma temporal la variable `LC_CTYPE`.

`str.format_map (mapping)`

Similar a `str.format(**mapping)`, pero se usa `**mapping` de forma directa y no se copia a una diccionario. Esto es útil si `*mapping*` es, por ejemplo, una instancia de una subclase de *dict*:

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
```

(continué en la próxima página)

(proviene de la página anterior)

```
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

Nuevo en la versión 3.2.

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

Como `find()`, pero eleva una excepción de tipo `ValueError` si no se encuentra la cadena a buscar.

`str.isalnum()`

Retorna `True` si todos los caracteres de la cadena son alfanuméricos y hay, al menos, un carácter. En caso contrario, retorna `False`. Un carácter `c` se considera alfanumérico si alguna de las siguientes funciones retorna `True`: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()` o `c.isnumeric()`.

`str.isalpha()`

Retorna `True` si todos los caracteres de la cadena son alfabéticos y hay, al menos, un carácter. En caso contrario, retorna `False`. Los caracteres alfabéticos son aquellos definidos en la base de datos de Unicode como «*Letter*», es decir, aquellos cuya propiedad categoría general es «*Lm*», «*Lt*», «*Lu*», «*Ll*» o «*Lo*». Nótese que esta definición de «Alfabético» es diferente de la que usa el estándar Unicode.

`str.isascii()`

Retorna `True` si la cadena de caracteres está vacía, o si todos los caracteres de la cadena son ASCII. En caso contrario, retorna `False`. Los caracteres ASCII son aquellos cuyos puntos de código Unicode están en el rango U+0000-U+007F.

Nuevo en la versión 3.7.

`str.isdecimal()`

Retorna `True` si todos los caracteres de la cadena son caracteres decimales y hay, al menos, un carácter. En caso contrario, retorna `False`. Los caracteres decimales son aquellos que se pueden usar para formar números en base 10, por ejemplo, U+0660, ARABIC-INDIC DIGIT ZERO. Formalmente, un carácter decimal es un carácter en la categoría general «*Nd*» de Unicode.

`str.isdigit()`

Retorna `True` si todos los caracteres de la cadena son dígitos y hay, al menos, un carácter. En caso contrario, retorna `False`. Los dígitos incluyen los caracteres decimales y dígitos que requieren un tratamiento especial, como por ejemplo los usados para superíndices. Esto incluye dígitos que no pueden ser usados para formar números en base 10, como los números *Kharosthi*. Formalmente, un dígito es un carácter que tiene la propiedad «*Numeric_Type*» definida como «*Digit*» o «*Decimal*».

`str.isidentifier()`

Retorna `True` si la cadena de caracteres es un identificador válido de acuerdo a la especificación del lenguaje, véase `identifiers`.

Se puede usar la función `keyword.iskeyword()` para comprobar si la cadena `s` es una palabra reservada, como `def` o `class`.

Ejemplo:

```
>>> from keyword import iskeyword

>>> 'hello'.isidentifier(), iskeyword('hello')
True, False
>>> 'def'.isidentifier(), iskeyword('def')
True, True
```

`str.islower()`

Retorna `True` si todos los caracteres de la cadena que tengan formas en mayúsculas y minúsculas⁴ están en minúsculas y hay, al menos, un carácter de ese tipo. En caso contrario, retorna `False`.

⁴ Los caracteres con versiones mayúsculas/minúsculas son aquellos cuya categoría general corresponde con «*Lu*» (Letra, Mayúscula), «*Ll*» (Letra, minúscula) o «*Lt*» (Letra, *tilde*case).

`str.isnumeric()`

Retorna `True` si todos los caracteres de la cadena son caracteres numéricos y hay, al menos, un carácter. En caso contrario, retorna `False`. Los caracteres numéricos incluyen los dígitos, y todos los caracteres Unicode que tiene definida la propiedad valor numérico, por ejemplo U+2155, VULGAR FRACTION ONE FIFTH. Formalmente, los caracteres numéricos son aquellos que la propiedad `Numeric_Type` definida como `Digit`, `Decimal` o `Numeric`.

`str.isprintable()`

Retorna `True` si todos los caracteres de la cadena son imprimibles o si la cadena está vacía. En caso contrario, retorna `False`. Los caracteres no imprimibles son aquellos definidos en la base de datos de Unicode como «other» o «Separator», con la excepción del carácter ASCII «espacio» (0x20), que se considera imprimible (Nótese que en este contexto, imprimible son aquellos caracteres que no necesitan ser escapados cuando se imprimen con la función `repr()`). No tiene relevancia en cadenas escritas a `sys.stdout` o `sys.stderr`.

`str.isspace()`

Retorna `True` si todos los caracteres de la cadena son espacios en blanco y hay, al menos, un carácter. En caso contrario, retorna `False`.

Un carácter se considera espacio en blanco si, en la base de datos de Unicode (Véase `unicodedata`), está clasificado en la categoría general `Zs` («Espacio, separador») o la clase bidireccional es `WS`, `B`, or `S`.

`str.istitle()`

Retorna `True` si las palabras en la cadena tiene forma de título y hay, al menos, un carácter, por ejemplo una mayúscula solo puede aparecer al principio o después de un carácter que no tenga formas alternativas mayúsculas-minúsculas, y las minúsculas solo después de carácter que si tiene formas alternativas mayúsculas-minúsculas. En caso contrario, retorna `False`.

`str.isupper()`

Retorna `True` si todos los caracteres de la cadena que tengan formas en mayúsculas y minúsculas⁴ están en mayúsculas y hay, al menos, un carácter de ese tipo. En caso contrario, retorna `False`.

`str.join(iterable)`

Retorna una cadena de caracteres formada por la concatenación de las cadenas en el *iterable*. Se eleva una excepción de tipo `TypeError` si alguno de los elementos en el *iterable* no es una cadena, incluyendo objetos de tipo `bytes`. Se usa como separador entre los elementos la cadena de caracteres pasada como parámetro.

`str.ljust(width[, fillchar])`

Retorna el texto de la cadena, justificado a la izquierda en una cadena de longitud *width*. El carácter de relleno a usar viene definido por el parámetro *fillchar* (Por defecto se usa el carácter espacio ASCII). Si la cadena original tiene una longitud `len(s)` igual o superior a *width*, se Retorna el texto sin modificar.

`str.lower()`

Retorna una copia de la cadena de caracteres con todas las letras en minúsculas⁴.

El algoritmo usado para la conversión a minúsculas está descrito en la sección 3.13 del estándar Unicode.

`str.lstrip([chars])`

Retorna una copia de la cadena, eliminado determinados caracteres si se encuentran al principio. El parámetro *chars* especifica el conjunto de caracteres a eliminar. Si se omite o si se especifica `None`, se eliminan todos los espacios en blanco. No debe entenderse el valor de *chars* como un prefijo, sino que se elimina cualquier combinación de sus caracteres:

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

static `str.maketrans(x[, y[, z]])`

Este método estático retorna una tabla de traducción apta para ser usada por el método `str.translate()`.

Si solo se usa un parámetro, este debe ser un diccionario que mapea valores de punto Unicode (enteros) o caracteres (Cadenas de longitud 1) a valores Unicode, cadenas (De cualquier longitud) o `None`. Las claves se

convertirán a ordinales.

Si se pasan dos parámetros, deben ser cadenas de la misma longitud, y en la tabla resultante, cada carácter en *x* se mapea al carácter en la misma posición en *y*. Si se añade un tercer parámetro, debe ser una cadena de caracteres, todos los cuales se mapearán a `None` en la tabla resultante.

`str.partition (sep)`

Divide la cadena en la primera ocurrencia de *sep*, y retorna una tupla de tres elementos, conteniendo la parte anterior al separador, el separador en sí y la parte posterior al separador. Si no se encuentra el separador, Retorna una tupla de tres elementos, el primero la cadena original y los dos siguientes son cadenas vacías.

`str.replace (old, new[, count])`

Retorna una copia de la cadena con todas las ocurrencias de la cadena *old* sustituidas por *new*. Si se utiliza el parámetro *count*, solo se cambian las primeras *count* ocurrencias.

`str.rfind (sub[, start[, end]])`

Retorna el mayor índice dentro de la cadena *s* donde se puede encontrar la cadena *sub*, estando *sub* incluida en *s*[*start*:*end*]. Los parámetros opcionales *start* y *end* se interpretan igual que en las operaciones de rebanado. retorna -1 si no se encuentra *sub*.

`str.rindex (sub[, start[, end]])`

Como el método `rfind()`, pero eleva la excepción `ValueError` si no se encuentra la cadena *sub*.

`str.rjust (width[, fillchar])`

Retorna el texto de la cadena, justificado a la derecha en una cadena de longitud *width*. El carácter de relleno a usar viene definido por el parámetro *fillchar* (Por defecto se usa el carácter espacio ASCII). Si *width* es menor o igual que `len(s)`, se retorna el texto sin modificar.

`str.rpartition (sep)`

Divide la cadena en la última ocurrencia de *sep*, y retorna una tupla de tres elementos, conteniendo la parte anterior al separador, el separador en sí y la parte posterior al separador. Si no se encuentra el separador, Retorna una tupla de tres elementos, los dos primeras posiciones con cadenas vacías y en la tercera la cadena original.

`str.rsplit (sep=None, maxsplit=-1)`

Retorna una lista con las palabras que componen la cadena de caracteres original, usando como separador el valor de *sep*. Si se utiliza el parámetro *maxsplit*, se realizan como máximo *maxsplit* divisiones, retornando los que están más a la derecha. Si no se especifica *sep* o se pasa con valor `None`, se usa como separador cualquier carácter de espacio en blanco. Si no contamos la diferencia de empezar las divisiones desde la derecha, el comportamiento de este método `rsplit()` es equivalente al de `split()`, que se describe con detalle más adelante.

`str.rstrip ([chars])`

Retorna una copia de la cadena, eliminado determinados caracteres si se encuentran al final. El parámetro *chars* especifica el conjunto de caracteres a eliminar. Si se omite o si se especifica `None`, se eliminan todos los espacios en blanco. No debe entenderse el valor de *chars* como un prefijo, sino que se elimina cualquier combinación de sus caracteres:

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

`str.split (sep=None, maxsplit=-1)`

Retorna una lista con las palabras que componen la cadena de caracteres original, usando como separador el valor de *sep*. Si se utiliza el parámetro *maxsplit*, se realizan como máximo *maxsplit* divisiones, (Por tanto, la lista resultante tendrá *maxsplit*+1 elementos). Si no se especifica *maxsplit* o se pasa con valor -1, entonces no hay límite al número de divisiones a realizar (Se harán todas las que se puedan).

Si se especifica *sep*, las repeticiones de caracteres delimitadores no se agrupan juntos, sino que se considera que están delimitando cadenas vacías (Por ejemplo, `'1,,2'.split(',')` retorna `['1', '', '2']`). El parámetro *sep* puede contener más de un carácter (Por ejemplo, `'1<>2<>3'.split('<>')` retorna `['1', '2', '3']`). Dividir una cadena vacía con un separador determinado retornará `['']`.

Por ejemplo:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,.'.split(',')
['1', '2', '', '3', '']
```

Si no se especifica *sep* o es `None`, se usa un algoritmo de división diferente: Secuencias consecutivas de caracteres de espacio en blanco se consideran como un único separador, y el resultado no contendrá cadenas vacías ni al principio ni al final de la lista, aunque la cadena original tuviera espacios en blanco al principio o al final. En consecuencia, dividir una cadena vacía o una cadena que solo contenga espacios en blanco usando `None` como separador siempre retornará una lista vacía `[]`.

Por ejemplo:

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> ' 1 2 3 '.split()
['1', '2', '3']
```

`str.splitlines([keepends])`

Retorna una lista con las líneas en la cadena, dividiendo por los saltos de línea. Los caracteres de salto de línea en sí no se incluyen a no ser que se especifique lo contrario pasando el valor `True` en al parámetro *keepends*.

Este método considera como saltos de línea los siguientes caracteres. En concreto, estos son un superconjunto de los *saltos de líneas universales*.

Representación	Descripción
<code>\n</code>	Salto de línea
<code>\r</code>	Retorno de carro
<code>\r\n</code>	Retorno de carro + salto de línea
<code>\v o \x0b</code>	Tabulación de línea
<code>\f o \x0c</code>	Avance de página
<code>\x1c</code>	Separador de archivo
<code>\x1d</code>	Separador de grupo
<code>\x1e</code>	Separador de registro
<code>\x85</code>	Siguiente línea (Código de control <i>CI</i>)
<code>\u2028</code>	Separador de línea
<code>\u2029</code>	Separador de párrafo

Distinto en la versión 3.2: Se añaden `\v` y `\f` a la lista de separadores.

Por ejemplo:

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

Al contrario que con `split()`, cuando se especifica una cadena con *sep*, el método retorna una lista vacía para la cadena vacía, y un salto de línea al final del texto no produce una línea extra:

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


Por comparación, `split('\n')` entrega:

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

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

Retorna `True` si la cadena empieza por *prefix*, en caso contrario Retorna `False`. El valor de *prefix* puede ser también una tupla de prefijos por los que buscar. Con el parámetro opcional *start*, la comprobación empieza en esa posición de la cadena.

`str.strip([chars])`

Retorna una copia de la cadena con los caracteres indicados eliminados, tanto si están al principio como al final de la cadena. El parámetro opcional *chars* es una cadena que especifica el conjunto de caracteres a eliminar. Si se omite o se usa `None`, se eliminan los caracteres de espacio en blanco. No debe entenderse el valor de *chars* como un prefijo, sino que se elimina cualquier combinación de sus caracteres:

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

Los caracteres indicados por *chars* se eliminan de los extremos al principio y al final de la cadena. Se elimina los caracteres del inicio hasta que se encuentra un carácter que no esté incluido en el conjunto definido por *chars*. Se procede de manera similar para los caracteres al final. Por ejemplo:

```
>>> comment_string = '#..... Section 3.2.1 Issue #32 .....'
>>> comment_string.strip('#! ')
'Section 3.2.1 Issue #32'
```

`str.swapcase()`

Retorna una copia de la cadena con los caracteres en mayúsculas convertidos a minúsculas, y viceversa. Nótese que no es necesariamente cierto que `s.swapcase().swapcase() == s`.

`str.title()`

Retorna una versión en forma de título de la cadena, con la primera letra de cada palabra en mayúsculas y el resto en minúsculas.

Por ejemplo:

```
>>> 'Hello world'.title()
'Hello World'
```

El algoritmo usa una definición sencilla e independiente del lenguaje, consistente en considerar una palabra como un grupo de letras consecutivas. Esta definición funciona en varios entornos, pero implica que, por ejemplo en inglés, los apóstrofes en las contracciones y en los posesivos constituyen una separación entre palabras, que puede que no sea el efecto deseado:

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

Se puede solucionar parcialmente el problema de los apóstrofes usando expresiones regulares:

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

`str.translate(table)`

Retorna una copia de la cadena en la que cada carácter ha sido sustituido por su equivalente definido en la tabla de traducción dada. La tabla puede ser cualquier objeto que soporta el acceso mediante índices implementado en método `__getitem__()`, normalmente un objeto de tipo *mapa* o *secuencia*. Cuando se accede como índice con un código Unicode (Un entero), el objeto tabla puede hacer una de las siguientes cosas: retornar otro código Unicode o retornar una cadena de caracteres, de forma que se usaran uno u otro como reemplazo en la cadena de salida; retornar `None` para eliminar el carácter en la cadena de salida, o elevar una excepción de tipo `LookupError`, que hará que el carácter se copie igual en la cadena de salida.

Se puede usar `str.maketrans()` para crear un mapa de traducción carácter a carácter de diferentes formas.

Véase también el módulo `codecs` para una aproximación más flexible al mapeo de caracteres.

`str.upper()`

Retorna una copia de la cadena, con todos los caracteres con formas mayúsculas/minúsculas⁴ pasados a minúsculas. Nótese que `s.upper().isupper()` puede retornar falso si `s` contiene caracteres que no tengan las dos formas, o si la categoría Unicode del carácter o caracteres resultantes no es «*Lu*» (Letra, mayúsculas), sino, por ejemplo, «*Lt*» (Letra, Título).

El algoritmo de paso a mayúsculas es el descrito en la sección 3.13 del estándar Unicode.

`str.zfill(width)`

Retorna una copia de la cadena, rellena por la izquierda con los carácter ASCII '0' necesarios para conseguir una cadena de longitud *width*. El carácter prefijo de signo ('+'/'-') se gestiona insertando el relleno *después* del carácter de signo en vez de antes. Si *width* es menor o igual que `len(s)`, se retorna la cadena original.

Por ejemplo:

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

4.7.2 Formateo de cadenas al estilo `*printf*`

Nota: Las operaciones de formateo explicadas aquí tienen una serie de peculiaridades que conducen a ciertos errores comunes (Como fallar al representar tuplas y diccionarios correctamente). Se pueden evitar estos errores usando las nuevas cadenas de caracteres con formato, el método `str.format()`, o *plantillas de cadenas de caracteres*. Cada una de estas alternativas proporcionan sus propios compromisos entre facilidad de uso, flexibilidad y capacidad de extensión.

Las cadenas de caracteres tienen una operación básica: El operador `%` (módulo). Esta operación se conoce también como *formateo* de cadenas y operador de interpolación. Dada la expresión `formato valores` (Donde *formato* es una cadena), las especificaciones de conversión indicadas en la cadena con el símbolo `%` son reemplazadas por cero o más elementos de *valores*. El efecto es similar a usar la función del lenguaje C `sprintf()`.

Si *formato* tiene un único marcador, *valores* puede ser un objeto sencillo, no una tupla.⁵ En caso contrario, *valores* debe ser una tupla con exactamente el mismo número de elementos que marcadores usados en la cadena de formato, o un único objeto de tipo mapa (Por ejemplo, un diccionario).

Un especificador de conversión consiste en dos o más caracteres y tiene los siguientes componentes, que deben aparecer en el siguiente orden:

1. El carácter `'%'`, que identifica el inicio del marcador.
2. Una clave de mapeo (opcional), consistente en una secuencia de caracteres entre paréntesis, como por ejemplo, `(somename)`.
3. Indicador de conversión (opcional), que afecta el resultado de ciertas conversiones de tipos.

⁵ Para formatear solo una tupla se debe, por tanto, usar una tupla conteniendo un único elemento, que sería la tupla a ser formateada.

4. Valor de ancho mínimo (opcional). Si se especifica un '*' (asterisco), el ancho real se lee del siguiente elemento de la tupla *valores*, y el objeto a convertir viene después del ancho mínimo, con un indicador de precisión opcional.
5. Precisión (Opcional), en la forma '.' (punto) seguido de la precisión. Si se especifica un '*' (Asterisco), el valor de precisión real se lee del siguiente elemento de la tupla *valores*, y el valor a convertir viene después de la precisión.
6. Modificador de longitud (Opcional).
7. Tipo de conversión.

Cuando el operador derecho es un diccionario (o cualquier otro objeto de tipo mapa), los marcadores en la cadena *deben* incluir un valor de clave entre paréntesis, inmediatamente después del carácter '%'. El valor de la clave se usa para seleccionar el valor a formatear desde el mapa. Por ejemplo:

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

En este caso, no se pueden usar el especificador * en la cadena de formato (Dado que requiere una lista secuencial de parámetros).

Los indicadores de conversión son:

Flag	Significado
'#'	El valor a convertir usará la «forma alternativa» (Que se definirá más adelante)
'0'	La conversión rellena con ceros por la izquierda para valores numéricos.
'-'	El valor convertido se ajusta a la izquierda (Sobreescribe la conversión '0' si se especifican los dos)
' '	(Un espacio) Se deba añadir un espacio en blanco antes de un número positivo (O una cadena vacía) si se usa una conversión con signo.
'+'	Un carácter signo ('+' o '-') precede a la conversión (Sobreescribe el indicador de «espacio»)

Puede estar presente un modificador de longitud (h, l o L), pero se ignora y no es necesario para Python – por lo que, por ejemplo, la salida de %ld es idéntica a %d.

Los tipos de conversión son:

Con- ver- sión	Significado	No- tas
'd'	Entero decimal con signo.	
'i'	Entero decimal con signo.	
'o'	Valor octal con signo.	(1)
'u'	Obsoleto – es idéntico a 'd'.	(6)
'x'	Hexadecimal con signo (En minúsculas)	(2)
'X'	Hexadecimal con signo (En mayúsculas)	(2)
'e'	Formato en coma flotante exponencial (En minúsculas).	(3)
'E'	Formato en coma flotante exponencial (En mayúsculas).	(3)
'f'	Formato en coma flotante decimal.	(3)
'F'	Formato en coma flotante decimal.	(3)
'g'	Formato en coma flotante. Usa formato exponencial con minúsculas si el exponente es menor que -4 o no es menor que la precisión, en caso contrario usa el formato decimal.	(4)
'G'	Formato en coma flotante. Usa formato exponencial con mayúsculas si el exponente es menor que -4 o no es menor que la precisión, en caso contrario usa el formato decimal.	(4)
'c'	Un único carácter (Acepta números enteros o cadenas de caracteres de longitud 1)	
'r'	Cadena de texto (Representará cualquier objeto usando la función <i>repr()</i>).	(5)
's'	Cadena de texto (Representará cualquier objeto usando la función <i>str()</i>).	(5)
'a'	Cadena de texto (Representará cualquier objeto usando la función <i>ascii()</i>).	(5)
'%'	No se representa ningún argumento, obteniéndose en el resultado la cadena '%'	

Notas:

- (1) La forma alternativa hace que se inserte antes del primer dígito un prefijo indicativo del formato octal ('0o')
- (2) La forma alternativa hace que se inserte antes del primer dígito uno de los dos prefijos indicativos del formato hexadecimal '0x' or '0X' (Que se use uno u otro depende de que indicador de formato se haya usado, 'x' or 'X').
- (3) La forma alternativa hace que se incluya siempre el símbolo del punto o coma decimal, incluso si no hubiera dígitos después.
La precisión determina el número de dígitos que vienen después del punto decimal, y por defecto es 6.
- (4) La forma alternativa hace que se incluya siempre el símbolo del punto o coma decimal, y los ceros a su derecha no se eliminan, como sería el caso en la forma normal.
La precisión determina el número de dígitos significativos que vienen antes y después del punto decimal, y por defecto es 6.
- (5) Si la precisión es N, la salida se trunca a N caracteres.
- (6) Véase [PEP 237](#).

Como en Python las cadenas de caracteres tiene una longitud explícita, la conversión de %s no requiere que la cadena termine con '\0'.

Distinto en la versión 3.1: Las conversiones %f para números con valores absolutos mayores que 1e50 ya no son reemplazadas por conversiones %g.

4.8 Tipos de secuencias binarias — bytes, bytearray y memoryview

Los tipos básicos para trabajar con datos binarios son las clases `bytes` y `bytearray`. Ambas pueden ser usadas por la clase `memoryview`, que usa el protocolo buffer para acceder a la memoria de otros objetos binarios sin necesidad de hacer una copia.

El módulo `array` soporta un almacenamiento eficiente de tipos de datos básicos como enteros de 32 bits o números en formato de doble precisión en coma flotante IEEE754.

4.8.1 Objetos de tipo Bytes

Los objetos `bytes` son secuencias inmutables de bytes. Como muchos de los protocolos binarios más usados se basan en la codificación ASCII para texto, los objetos `bytes` ofrecen varios métodos que solo son válidos cuando se trabaja con datos compatibles ASCII y son, en varios aspectos, muy cercanos a los cadenas de texto.

class bytes ([*source*[, *encoding*[, *errors*]]])

Para empezar, la sintaxis de los valores literales de `bytes` son prácticamente iguales que para las cadenas de texto, con la diferencia de que se añade el carácter `b` como prefijo:

- Comillas sencillas: `b'Se siguen aceptando comillas "dobles" embebidas'`
- Comillas dobles: `b'Se siguen aceptando comillas 'simples' embebidas'`.
- Comillas triples: `b'''3 comillas simples''',b"""3 comillas dobles"""`

Solo se admiten caracteres ASCII en representaciones literales de `bytes` (Con independencia del tipo de codificación declarado). Cualquier valor por encima de 127 debe ser definido usando su secuencia de escape.

Al igual que con las cadenas, los literales de `bytes` pueden usar el prefijo `r` para deshabilitar el procesado de las secuencias de escape. Véase `strings` para más información acerca de los diferentes formas de expresar `bytes` de forma literal, incluyendo el soporte de secuencias de escape.

Aunque las secuencias de bytes y sus representaciones se basen en texto ASCII, los objetos `bytes` se comportan más como secuencias inmutables de números enteros, donde cada elemento de la secuencia está restringido

a los valores de x tal que $0 \leq x < 256$ (Si se intenta violar esta restricción se elevará una excepción de tipo `ValueError`). Esto se ha hecho de forma intencionada para enfatizar que, aunque muchos formatos binarios incluyen elementos basados en caracteres ASCII y pueden ser manipulados mediante algunas técnicas de procesamiento de textos, este no es el caso general para los datos binarios (Aplicar algoritmos pensados para proceso de textos a datos binarios que no se compatibles con ASCII normalmente corromperán dichos datos).

Además de con literales, se pueden crear objetos de tipo `byte` de las siguientes maneras:

- Un secuencia de una longitud especificada rellena con ceros: `bytes(10)`
- A partir de un iterable de números enteros: `bytes(range(20))`
- Copiando datos binarios ya existentes mediante el protocolo *buffer*: `bytes(obj)`

Véase además la función básica `bytes`.

Como dos dígitos hexadecimales se corresponden exactamente con un byte, suelen usarse números hexadecimales para describir datos binarios. Por ello, los objetos de tipo `byte` disponen de un método adicional para leer datos en ese formato:

classmethod `fromhex(string)`

Este método de clase de `bytes` retorna un objeto binario, decodificado a partir de la cadena suministrada como parámetro. La cadena de texto debe consistir en dos dígitos hexadecimales por cada byte, ignorándose los caracteres ASCII de espacio en blanco, si los hubiera.

```
>>> bytes.fromhex('2Ef0 F1f2 ')
b'\xf0\xf1\xf2'
```

Distinto en la versión 3.7: El método `bytes.fromhex()` ignora ahora todos los caracteres ASCII de espacio en blanco, no solo el carácter espacio.

Existe una función que realiza la operación inversa, es decir, transforma un objeto binario en una representación textual usando hexadecimal.

hex `([sep[, bytes_per_sep]])`

Retorna una cadena de texto que contiene dos dígitos hexadecimales por cada byte de la instancia.

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

Si quieres que la cadena en hexadecimal sea más fácil de leer, se puede especificar un único carácter separador con el parámetro `sep` para que se añada a la salida. Un segundo parámetro opcional, `bytes_per_sep`, controla los espacios. Valores positivos calculan la posición del separador desde la derecha, los negativos lo hacen desde la izquierda.

```
>>> value = b'\xf0\xf1\xf2'
>>> value.hex('-')
'f0-f1-f2'
>>> value.hex('_', 2)
'f0_f1f2'
>>> b'UDDLRRLRAB'.hex(' ', -4)
'55554444 4c524c52 4142'
```

Nuevo en la versión 3.5.

Distinto en la versión 3.8: El método `bytes.hex()` ahora soporta los parámetros opcionales `sep` y `bytes_per_sep`, que permiten insertar separadores entre los bytes de la cadena de salida.

Como los objetos de tipo `bytes` son secuencias de números enteros (Similares a tuplas), para un objeto binario `b`, `b[0]` retorna un entero, mientras que `b[0:1]` retorna un objeto de tipo `bytes` de longitud 1 (Mientras que las cadenas de texto siempre retornan una cadena de longitud 1, ya sea accediendo por índice o mediante una operación de rebanada).

La representación de los objetos tipo `bytes` usa el formato literal (`b'...'`) ya que es, por lo general, más útil que, digamos, `bytes([46, 46, 46])`. Siempre se puede convertir un objeto binario en una lista de enteros usando `list(b)`.

Nota: Para usuarios de Python 2.x: En la serie Python 2.x, se permitía una variedad de conversiones implícitas entre cadenas de caracteres de 8 bits (El tipo de datos más cercano que se podía usar en estas versiones) y cadenas de caracteres Unicode. Esto se implementó como una forma de compatibilidad hacia atrás para reflejar el hecho de que originalmente Python solo soportaba textos de 8 bits, siendo el texto Unicode un añadido posterior. En Python 3.x, estas conversiones implícitas se han eliminado: Todas las conversiones entre datos binarios y textos Unicode deben ser explícitas, y objetos de tipo *bytes* y objetos de tipo cadena de caracteres siempre serán considerados diferentes.

4.8.2 Objetos de tipo *Bytearray*

Los objetos de tipo *bytearray* son versiones mutables de los objetos de tipo *bytes*.

class *bytearray* ([*source* [, *encoding* [, *errors*]]])

No existe una sintaxis específica para crear objetos de tipo *bytearray*, hay que crearlos siempre llamando a su constructor:

- Creando una secuencia vacía: `bytearray()`
- Creando una instancia de una longitud determinada, rellena con ceros: `bytearray(10)`
- A partir de un iterable de números enteros: `bytearray(range(20))`
- Copiando datos binarios ya existentes mediante el protocolo *buffer*: `bytearray(b'Hi!')`

Como los objetos *bytearray* son mutables, soportan todas las operaciones aplicables a tipos *mutables*, además de las operaciones propias de los *bytearrays* descritas en *Operaciones de bytes y bytearray*.

Véase también la función básica *bytearray*.

Como dos dígitos hexadecimales se corresponden exactamente con un byte, suelen usarse números hexadecimales para describir datos binarios. Por ello, los objetos de tipo *bytearray* disponen de un método de clase adicional para leer datos en ese formato:

classmethod *fromhex* (*string*)

Este método de clase de *bytes* retorna un objeto *bytearray*, decodificado a partir de la cadena suministrada como parámetro. La cadena de texto debe consistir en dos dígitos hexadecimales por cada byte, ignorándose los caracteres ASCII de espacio en blanco, si los hubiera.

```
>>> bytearray.fromhex('2Ef0 F1f2 ')
bytearray(b'\xf0\xf1\xf2')
```

Distinto en la versión 3.7: El método *bytearray.fromhex()* ignora ahora todos los caracteres ASCII de espacio en blanco, no solo el carácter espacio.

Existe una función que realiza la operación inversa, es decir, transforma un objeto *bytearray* en una representación textual usando hexadecimal.

hex ([*sep* [, *bytes_per_sep*]])

Retorna una cadena de texto que contiene dos dígitos hexadecimales por cada byte de la instancia.

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

Nuevo en la versión 3.5.

Distinto en la versión 3.8: De forma similar a *bytes.hex()*, *bytearray.hex()* soporta ahora los parámetros opcionales *sep* y *bytes_per_sep* para insertar separadores entre los bytes en la cadena hexadecimal de salida.

Como los objetos de tipo *bytearray* son secuencias de números enteros (Similares a listas), para un objeto *bytearray* *b*, *b[0]* retorna un entero, mientras que *b[0:1]* retorna un objeto de tipo *bytearray* de longitud 1 (Mientras que las cadenas de texto siempre retornan una cadena de longitud 1, ya sea accediendo por índice o mediante una operación de rebanada).

La representación de los objetos tipo *bytearray* usa el formato literal (`bytearray(b'...')`) ya que es, por lo general, más útil que, digamos, `bytearray([46, 46, 46])`. Siempre se puede convertir un objeto *bytearray* en una lista de enteros usando `list(b)`.

4.8.3 Operaciones de *bytes* y *bytearray*

Ambos tipos, *bytes* y *bytearray* soportan las operaciones *comunes* de las secuencias. Los operadores no funcionan solo con operandos del mismo tipo, sino con cualquier *objeto tipo binario*. Gracias a esta flexibilidad, estos tipos pueden combinarse libremente en expresiones sin que se produzcan errores. Sin embargo, el tipo del valor resultante puede depender del orden de los operandos.

Nota: Los métodos de objetos de tipo *bytes* y *bytesarray* no aceptan cadenas de caracteres como parámetros, de la misma manera que los métodos de las cadenas tampoco aceptan *bytes* como parámetros. Por ejemplo, debes escribir:

```
a = "abc"
b = a.replace("a", "f")
```

y:

```
a = b"abc"
b = a.replace(b"a", b"f")
```

Algunas operaciones de *bytes* y *bytearrays* asumen el uso de formatos binarios compatibles ASCII, y por tanto deben ser evitadas cuando trabajamos con datos binarios arbitrarios. Estas restricciones se explican a continuación.

Nota: Usar estas operaciones basadas en ASCII para manipular datos binarios que no se almacenan en un formato basado en ASCII pueden producir corrupción de datos.

Los siguientes métodos de *bytes* y *bytearrays* pueden ser usados con datos en formatos binarios arbitrarios.

```
bytes.count(sub[, start[, end]])
bytearray.count(sub[, start[, end]])
```

Retorna el número de secuencias no solapadas de la subsecuencia *sub* en el rango *[start, end]*. Los parámetros opcionales *start* y *end* se interpretan como en las operaciones de rebanado.

La subsecuencia a buscar puede ser cualquier *objeto tipo binario* o un número entero entre 0 y 255.

Distinto en la versión 3.3: También acepta como subsecuencia un número entero entre 0 y 255.

```
bytes.decode(encoding="utf-8", errors="strict")
bytearray.decode(encoding="utf-8", errors="strict")
```

Retorna una cadena de caracteres decodificada a partir de la secuencia de bytes. La codificación por defecto es 'utf-8'. El parámetro *errors* puede definir diferentes estrategias de gestión de errores. El valor por defecto de *errors* es 'strict', que hace que cualquier error de la decodificación eleve una excepción de tipo *UnicodeError*. Otros valores posibles son "ignore", 'replace' y cualquier otro nombre definido mediante la función `codecs.register_error()`, véase la sección *Manejadores de errores*. Para un listado de todos los valores de codificación posibles, véase *Codificaciones estándar*.

Nota: Pasando el parámetro *encoding* a la clase *str* permite decodificar cualquier *objeto tipo binario* directamente, sin necesidad de crear una objeto temporal de tipo *bytes* o *bytearray*.

Distinto en la versión 3.1: Añadido soporte para poder usar parámetros por nombre.

```
bytes.endswith(suffix[, start[, end]])
bytearray.endswith(suffix[, start[, end]])
```

Retorna True si los datos binarios acaban con el valor indicado por *suffix*, en caso contrario retorna False. El valor de *suffix* puede ser también una tupla de sufijos para buscar. Con el parámetro opcional *start*, la

comparación empieza a partir de esa posición. Si se especifica el parámetro opcional *end*, la comparación termina en esa posición.

El sufijo (o sufijos) a buscar puede ser cualquier *objeto tipo binario*.

```
bytes.find(sub[, start[, end]])  
bytearray.find(sub[, start[, end]])
```

Retorna el mínimo índice dentro de los datos donde se ha encontrado la subsecuencia *sub*, de forma que *sub* está contenida en la rebanada *s[start:end]*. Los parámetros opcionales *start* y *end* se interpretan como en las operaciones de rebanadas. retorna *-1* si no se puede encontrar *sub*.

La subsecuencia a buscar puede ser cualquier *objeto tipo binario* o un número entero entre 0 y 255.

Nota: El método *find()* se debe usar solo si se necesita saber la posición de *sub*. Si solo se necesita comprobar si *sub* es una parte de *s*, es mejor usar el operador *in*:

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

Distinto en la versión 3.3: También acepta como subsecuencia un número entero entre 0 y 255.

```
bytes.index(sub[, start[, end]])  
bytearray.index(sub[, start[, end]])
```

Como *find()*, pero eleva una excepción de tipo *ValueError* si no se encuentra la subsecuencia a buscar.

La subsecuencia a buscar puede ser cualquier *objeto tipo binario* o un número entero entre 0 y 255.

Distinto en la versión 3.3: También acepta como subsecuencia un número entero entre 0 y 255.

```
bytes.join(iterable)  
bytearray.join(iterable)
```

Retorna un objeto de tipo *bytes* o *bytearray* que es la concatenación de las secuencias binarias en *iterable*. Si alguno de los objetos de la secuencia no es un *objeto tipo binario* se eleva la excepción *TypeError*, incluso si son cadenas de caracteres (objetos *str*). El separador entre los distintos elementos es el contenido del objeto *bytes* o *bytearray* usando para invocar el método.

```
static bytes.maketrans(from, to)  
static bytearray.maketrans(from, to)
```

Este método estático retorna una tabla de traducción apta para ser usada por el método *bytes.translate()*, que mapea cada carácter en *from* en la misma posición en *to*; tanto *from* como *to* debe ser *objetos tipo binario* y deben tener la misma longitud.

Nuevo en la versión 3.1.

```
bytes.partition(sep)  
bytearray.partition(sep)
```

Retorna la secuencia en la primera ocurrencia de *sep*, y retorna una tupla de tres elementos que contiene la parte antes del separador, el separador en sí o una copia de tipo *bytearray* y la parte después del separador. Si no se encuentra el separador, retorna una tupla de tres elementos, con la primera posición ocupada por la secuencia original, y las dos posiciones siguientes rellenas con objetos *bytes* o *bytearray* vacíos.

El separador a buscar puede ser cualquier *objeto tipo binario*.

```
bytes.replace(old, new[, count])  
bytearray.replace(old, new[, count])
```

Retorna una copia de la secuencia con todas las ocurrencias de *old* sustituidas por *new*. Si se utiliza el parámetro *count*, solo se cambian las primeras *count* ocurrencias.

La subsecuencia a buscar y su reemplazo puede ser cualquier *objeto tipo binario*.

Nota: La versión *bytearray* de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

```
bytes.rfind(sub[, start[, end]])
bytearray.rfind(sub[, start[, end]])
```

Retorna el mayor índice dentro de la secuencia *s* donde se puede encontrar *sub*, estando *sub* incluida en *s*[start:end]. Los parámetros opcionales *start* y *end* se interpretan igual que en las operaciones de rebando. Retorna -1 si no se encuentra *sub*.

La subsecuencia a buscar puede ser cualquier *objeto tipo binario* o un número entero entre 0 y 255.

Distinto en la versión 3.3: También acepta como subsecuencia un número entero entre 0 y 255.

```
bytes.rindex(sub[, start[, end]])
bytearray.rindex(sub[, start[, end]])
```

Como el método *rfind()*, pero eleva la excepción *ValueError* si no se encuentra *sub*.

La subsecuencia a buscar puede ser cualquier *objeto tipo binario* o un número entero entre 0 y 255.

Distinto en la versión 3.3: También acepta como subsecuencia un número entero entre 0 y 255.

```
bytes.rpartition(sep)
bytearray.rpartition(sep)
```

Divide la secuencia en la primera ocurrencia de *sep*, y retorna una tupla de tres elementos que contiene la parte antes del separador, el separador en sí o una copia de tipo *bytearray* y la parte después del separador. Si no se encuentra el separador, retorna una tupla de tres elementos, con las dos primeras posiciones rellenas con objetos *bytes* o *bytearray* vacíos, y la tercera posición ocupada por la secuencia original.

El separador a buscar puede ser cualquier *objeto tipo binario*.

```
bytes.startswith(prefix[, start[, end]])
bytearray.startswith(prefix[, start[, end]])
```

Retorna True si los datos binarios empiezan con el valor indicado por *prefix*, en caso contrario retorna False. El valor de *prefix* puede ser también una tupla de prefijos para buscar. Con el parámetro opcional *start*, la comparación empieza a partir de esa posición. Si se especifica el parámetro opcional *end*, la comparación termina en esa posición.

El prefijo (o prefijos) a buscar puede ser cualquier *objeto tipo binario*.

```
bytes.translate(table, /, delete=b'')
bytearray.translate(table, /, delete=b'')
```

Retorna una copia del objeto *bytes* o *bytearray* donde todas las ocurrencias de bytes especificados en el parámetro *delete* han sido borrados, y el resto han sido mapeados a través de la tabla de traducción indicada, que debe ser un objeto de tipo *bytes* con una longitud de 256 elementos.

Puedes usar el método *bytes.maketrans()* para crear la tabla de traducción.

Se puede ajustar el parámetro *table* a None para conseguir una traducción que solo borra caracteres:

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

Distinto en la versión 3.6: El parámetro *delete* se puede ahora especificar por nombre.

Los siguientes métodos de los objetos *bytes* y *bytearray* presentan un comportamiento por defecto que asume el uso de formatos binarios compatibles con ASCII, pero aun así pueden ser usados con datos binarios arbitrarios usando los parámetros apropiados. Nótese que todos los métodos de *bytearray* en esta sección nunca modifican los datos internamente, sino que siempre retornan objetos nuevos.

```
bytes.center(width[, fillbyte])
bytearray.center(width[, fillbyte])
```

Retorna una copia del objeto centrado en una secuencia de longitud *width*. El relleno se realiza usando el valor definido en el parámetro *fillbyte* (Por defecto, el carácter espacio en ASCII). Para los objetos de tipo *bytes*, se retorna la secuencia original intacta si *width* es menor o igual que *len(s)*.

Nota: La versión *bytearray* de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

```
bytes.ljust (width[, fillbyte])  
bytearray.ljust (width[, fillbyte])
```

Retorna una copia del objeto justificado por la izquierda en una secuencia de longitud *width*. El relleno se realiza usando el valor definido en el parámetro *fillbyte* (Por defecto, el carácter espacio en ASCII). Para los objetos de tipo *bytes*, se retorna la secuencia original intacta si *width* es menor o igual que `len(s)`.

Nota: La versión *bytearray* de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

```
bytes.lstrip ([chars])  
bytearray.lstrip ([chars])
```

Retorna una copia de la secuencia con los caracteres iniciales especificados eliminados. El parámetro *chars* es una secuencia binaria que especifica el conjunto bytes a ser eliminados; el nombre hace referencia a que este método se usa normalmente con secuencias de caracteres ASCII. Si no se indica o si se especifica *None*, el comportamiento por defecto será eliminar los caracteres de espacio ASCII. No debe entenderse el valor de *chars* como un prefijo, sino que se elimina cualquier combinación de sus caracteres:

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

La secuencia binaria de bytes a eliminar deber ser un *objeto tipo binario*.

Nota: La versión *bytearray* de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

```
bytes.rjust (width[, fillbyte])  
bytearray.rjust (width[, fillbyte])
```

Retorna una copia del objeto justificado por la derecha en una secuencia de longitud *width*. El relleno se realiza usando el valor definido en el parámetro *fillbyte* (Por defecto, el carácter espacio en ASCII). Para los objetos de tipo *bytes*, se retorna la secuencia original intacta si *width* es menor o igual que `len(s)`.

Nota: La versión *bytearray* de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

```
bytes.rsplit (sep=None, maxsplit=-1)  
bytearray.rsplit (sep=None, maxsplit=-1)
```

Divide una secuencia binaria en subsecuencias del mismo tipo, usando como separador el valor de *sep*. Si se utiliza el parámetro *maxsplit*, se realizan como máximo *maxsplit* divisiones, retornando los que están más a la derecha. Si no se especifica *sep* o se pasa con valor *None*, se usa como separador el carácter espacio en ASCII. Si no contamos la diferencia de empezar las divisiones desde la derecha, el comportamiento de este método *rsplit()* es equivalente al de *split()*, que se describe con detalle más adelante.

```
bytes.rstrip ([chars])  
bytearray.rstrip ([chars])
```

Retorna una copia de la cadena, eliminado determinados bytes si se encuentran al final. El parámetro *chars* es una secuencia binaria que especifica el conjunto de bytes a eliminar; el nombre hace referencia a que este método se usa normalmente con secuencias de caracteres ASCII. Si se omite o si se especifica *None*, se eliminan los caracteres espacio en ASCII. No debe entenderse el valor de *chars* como un prefijo, sino que se elimina cualquier combinación de sus caracteres:

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

La secuencia binaria de bytes a eliminar deber ser un *objeto tipo binario*.

Nota: La versión *bytearray* de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

`bytes.split (sep=None, maxsplit=-1)`
`bytearray.split (sep=None, maxsplit=-1)`

Divide una secuencia binaria en subsecuencias del mismo tipo, usando como separador el valor de *sep*. Si se utiliza el parámetro *maxsplit* y es un número positivo, se realizan como máximo *maxsplit* divisiones (Resultando en una secuencia de como mucho *maxsplit*+1 elementos). Si no se especifica *sep* o se pasa '1', no hay límite al número de divisiones.

Si se especifica *sep*, las repeticiones de caracteres delimitadores no se agrupan juntos, sino que se considera que están delimitando cadenas vacías (Por ejemplo, `b'1,,2'.split(b',')` retorna `[b'1', b'', b'2']`). El parámetro *sep* puede contener más de un carácter (Por ejemplo, `b'1<>2<>3'.split(b'<>')` retorna `[b'1', b'2', b'3']`). Dividir una cadena vacía con un separador determinado retornará `[b'']` o `[bytearray(b'')]` dependiendo del tipo de objeto dividido. El parámetro *sep* puede ser cualquier *objeto tipo binario*.

Por ejemplo:

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3,.'.split(b',')
[b'1', b'2', b'', b'3', b'']
```

Si no se especifica *sep* o es *None*, se usa un algoritmo de división diferente: Secuencias consecutivas de caracteres de espacio en ASCII se consideran como un único separador, y el resultado no contendrá cadenas vacías ni al principio ni al final de la lista, aunque la cadena original tuviera espacios en blanco al principio o al final. En consecuencia, dividir una secuencia vacía o que solo contenga espacios en blanco usando *None* como separador siempre retornará una lista vacía `[]`.

Por ejemplo:

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b' 1 2 3 '.split()
[b'1', b'2', b'3']
```

`bytes.strip ([chars])`
`bytearray.strip ([chars])`

Retorna una copia de la secuencia con los bytes indicados eliminados, tanto si están al principio como al final de la cadena. El parámetro opcional *chars* es una secuencia de bytes que especifica el conjunto de caracteres a eliminar; el nombre hace referencia a que este método se usa normalmente con secuencias de caracteres ASCII. Si se omite o se usa *None*, se eliminan los caracteres de espacio ASCII. No debe entenderse el valor de *chars* como un prefijo o sufijo, sino que se elimina cualquier combinación de sus valores:

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

La secuencia binaria de bytes a eliminar deber ser un *objeto tipo binario*.

Nota: La versión *bytearray* de este método *no* modifica los valores internamente (no opera *in place*): siempre

produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

Los siguientes métodos de los objetos *bytes* y *bytearray* asumen el uso de formatos binarios compatibles con ASCII, y no deben ser usados con datos binarios arbitrarios. Nótese que todos los métodos de *bytearray* en esta sección nunca modifican los datos internamente, sino que siempre retornan objetos nuevos.

`bytes.capitalize()`
`bytearray.capitalize()`

Retorna una copia de la secuencia con cada byte interpretado como un carácter ASCII, y el primer byte en mayúsculas y el resto en minúsculas. Los valores que no sean ASCII no se ven modificados.

Nota: La versión *bytearray* de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

`bytes.expandtabs(tabsize=8)`
`bytearray.expandtabs(tabsize=8)`

Retorna una copia de la secuencia, con todos los caracteres ASCII *tab*/ reemplazados por uno o más espacios ASCII, dependiendo de la columna actual y del tamaño definido para el tabulador. Las posiciones de tabulación ocurren cada *tabsize* caracteres (Siendo el valor por defecto de *tabsize* 8, lo que produce las posiciones de tabulación 0, 8, 16,...). Para expandir la secuencia, la columna actual se pone a cero y se va examinando byte a byte. Si se encuentra un tabulador, (b'\t'), se insertan uno o más espacios hasta que sea igual a la siguiente posición de tabulación (El carácter tabulador en sí es descartado). Si el byte es un indicador de salto de línea (b'\n') o de retorno (b'\r'), se copia y el valor de columna actual se vuelve a poner a cero. Cualquier otro carácter es copiado sin cambios y hace que el contador de columna se incremente en 1, sin tener en cuenta como se representa impreso el byte:

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

Nota: La versión *bytearray* de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

`bytes.isalnum()`
`bytearray.isalnum()`

Retorna True si todos los bytes de la secuencia son caracteres alfabéticos ASCII o caracteres decimales ASCII y la secuencia no está vacía. En cualquier otro caso retorna False. Los caracteres alfabéticos ASCII son los bytes incluidos en la secuencia b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'. Los caracteres decimales ASCII son los bytes incluidos en la secuencia b'0123456789'.

Por ejemplo:

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

`bytes.isalpha()`
`bytearray.isalpha()`

Retorna True si todos los bytes de la secuencia son caracteres alfabéticos ASCII y la secuencia no está vacía. En cualquier otro caso Retorna False. Los caracteres alfabéticos ASCII son los bytes incluidos en la secuencia b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'.

Por ejemplo:

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

`bytes.isascii()`

`bytearray.isascii()`

Retorna True si la secuencia está vacía o si todos los bytes de la secuencia son caracteres ASCII. En cualquier otro caso retorna False. Los caracteres ASCII son los bytes incluidos en el rango 0-0x7F.

Nuevo en la versión 3.7.

`bytes.isdigit()`

`bytearray.isdigit()`

Retorna True si todos los bytes de la secuencia son caracteres decimales ASCII y la secuencia no está vacía. En cualquier otro caso retorna False. Los caracteres decimales ASCII son los bytes incluidos en la secuencia `b'0123456789'`.

Por ejemplo:

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

`bytes.islower()`

`bytearray.islower()`

Retorna True si hay al menos un carácter ASCII en minúsculas, y no hay ningún carácter ASCII en mayúsculas. En cualquier otro caso retorna False.

Por ejemplo:

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

Los caracteres ASCII en minúsculas son los bytes incluidos en la secuencia `b'abcdefghijklmnopqrstuvwxyz'`. los caracteres ASCII en mayúsculas son los bytes en la secuencia `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.isspace()`

`bytearray.isspace()`

Retorna True si todos los bytes de la secuencia son caracteres ASCII de espacio en blanco y la secuencia no está vacía. En cualquier otro caso Retorna False. Los caracteres de espacio en blanco ASCII son los bytes incluidos en la secuencia `b' \t\n\r\x0b\f'` (Espacio, tabulador, nueva línea, retorno de carro, tabulador vertical y avance de página).

`bytes.istitle()`

`bytearray.istitle()`

Retorna True si la secuencia ASCII está en forma de título, y la secuencia no está vacía. En cualquier otro caso retorna False. Véase el método `bytes.title()` para más detalles en la definición de «En forma de título».

Por ejemplo:

```
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

`bytes.isupper()`

`bytearray.isupper()`

Retorna `True` si hay al menos un carácter ASCII en mayúsculas, y no hay ningún carácter ASCII en minúsculas. En cualquier otro caso retorna `False`.

Por ejemplo:

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

Los caracteres ASCII en minúsculas son los bytes incluidos en la secuencia `b'abcdefghijklmnopqrstuvwxyz'`. los caracteres ASCII en mayúsculas son los bytes en la secuencia `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.lower()`

`bytearray.lower()`

Retorna una copia de la secuencia con todos los caracteres ASCII en mayúsculas sustituidos por su versión correspondiente en minúsculas.

Por ejemplo:

```
>>> b'Hello World'.lower()
b'hello world'
```

Los caracteres ASCII en minúsculas son los bytes incluidos en la secuencia `b'abcdefghijklmnopqrstuvwxyz'`. los caracteres ASCII en mayúsculas son los bytes en la secuencia `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Nota: La versión `bytearray` de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

`bytes.splitlines(keepends=False)`

`bytearray.splitlines(keepends=False)`

Retorna una lista de las líneas en la secuencia binaria, usando como separadores los *saltos de líneas universales*. Los caracteres usados como separadores no se incluyen en la lista de resultados a no ser que se pase el parámetro *keepends* a `True`.

Por ejemplo:

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

Al contrario que el método `split()`, cuando se especifica una cadena delimitadora con el parámetro *sep*, este método retorna una lista vacía para la cadena vacía, y un carácter de salto de línea al final de la secuencia no resulta en una línea extra:

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

`bytes.swapcase()`

`bytearray.swapcase()`

Retorna una copia de la secuencia con todos los caracteres ASCII en minúsculas sustituidos por su versión correspondiente en mayúsculas, y viceversa.

Por ejemplo:

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

Los caracteres ASCII en minúsculas son los bytes incluidos en la secuencia `b'abcdefghijklmnopqrstuvwxyz'`. los caracteres ASCII en mayúsculas son los bytes en la secuencia `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Al contrario que la función `str.swapcase()`, en este caso siempre se cumple que `bin.swapcase() . swapcase() == bin` para las versiones binarias. La conversión de mayúsculas a minúsculas son simétricas en ASCII, aunque esto no es el caso general para códigos de punto Unicode.

Nota: La versión `bytearray` de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

`bytes.title()`

`bytearray.title()`

Retorna una versión en forma de título de la secuencia binaria, con la primera letra de cada palabra en mayúsculas y el resto en minúsculas.

Por ejemplo:

```
>>> b'Hello world'.title()
b'Hello World'
```

Los caracteres ASCII en minúsculas son los bytes incluidos en la secuencia `b'abcdefghijklmnopqrstuvwxyz'`. los caracteres ASCII en mayúsculas son los bytes en la secuencia `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. El resto de los caracteres no presentan diferencias entre mayúsculas y minúsculas.

El algoritmo usa una definición sencilla e independiente del lenguaje, consistente en considerar una palabra como un grupo de letras consecutivas. Esta definición funciona en varios entornos, pero implica que, por ejemplo en inglés, los apóstrofes en las contracciones y en los posesivos constituyen una separación entre palabras, que puede que no sea el efecto deseado:

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

Se puede solucionar parcialmente el problema de los apóstrofes usando expresiones regulares:

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

Nota: La versión `bytearray` de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

`bytes.upper()`

`bytearray.upper()`

Retorna una copia de la secuencia con todos los caracteres ASCII en minúsculas sustituidos por su versión correspondiente en mayúsculas.

Por ejemplo:

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

Los caracteres ASCII en minúsculas son los bytes incluidos en la secuencia `b'abcdefghijklmnopqrstuvwxyz'`. los caracteres ASCII en mayúsculas son los bytes en la secuencia `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Nota: La versión *bytearray* de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

`bytes.zfill(width)`

`bytearray.zfill(width)`

Retorna una copia de la secuencia rellena por la izquierda con los caracteres ASCII `b'0'` necesarios para conseguir una cadena de longitud *width*. El carácter prefijo de signo (`b'+'/b'-'`) se gestiona insertando el relleno *después* del carácter de signo en vez de antes. Para objetos *bytes*, se retorna la secuencia original si *width* es menor o igual que `len(s)`.

Por ejemplo:

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

Nota: La versión *bytearray* de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

4.8.4 Usando el formato tipo `printf` con bytes

Nota: Las operaciones de formato explicadas aquí tienen una serie de peculiaridades que conducen a ciertos errores comunes (Como fallar al representar tuplas y diccionarios correctamente). Si el valor a representar es una tupla o un diccionario, hay que envolverlos en una tupla.

Los objetos binarios (*bytes*/*bytearray*) tienen una operación básica: El operador `%` (módulo). Esta operación se conoce también como operador de *formato* o de *interpolación*. Dada la expresión `formato % valores` (Donde *formato* es un objeto binario), las especificaciones de conversión indicadas en la cadena con el símbolo `%` son reemplazadas por cero o más elementos de *valores*. El efecto es similar a usar la función del lenguaje C `sprintf()`.

Si *formato* tiene un único marcador, *valores* puede ser un objeto sencillo, no una tupla.⁵ En caso contrario, *valores* debe ser una tupla con exactamente el mismo número de elementos que marcadores usados en el objeto binario, o un único objeto de tipo mapa (Por ejemplo, un diccionario).

Un especificador de conversión consiste en dos o más caracteres y tiene los siguientes componentes, que deben aparecer en el siguiente orden:

1. El carácter `'%'`, que identifica el inicio del marcador.
2. Una clave de mapeo (opcional), consistente en una secuencia de caracteres entre paréntesis, como por ejemplo, `(somename)`.
3. Indicador de conversión (opcional), que afecta el resultado de ciertas conversiones de tipos.
4. Valor de ancho mínimo (opcional). Si se especifica un `'*'` (asterisco), el ancho real se lee del siguiente elemento de la tupla *valores*, y el objeto a convertir viene después del ancho mínimo, con un indicador de precisión opcional.

5. Precisión (Opcional), en la forma ' . ' (punto) seguido de la precisión. Si se especifica un ' * ' (Asterisco), el valor de precisión real se lee del siguiente elemento de la tupla *valores*, y el valor a convertir viene después de la precisión.
6. Modificador de longitud (Opcional).
7. Tipo de conversión.

Cuando el operador derecho es un diccionario (o cualquier otro objeto de tipo mapa), los marcadores en el objeto binario *deben* incluir un valor de clave entre paréntesis, inmediatamente después del carácter '%'. El valor de la clave se usa para seleccionar el valor a formatear desde el mapa. Por ejemplo:

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

En este caso, no se pueden usar el especificador * en la cadena de formato (Dado que requiere una lista secuencial de parámetros).

Los indicadores de conversión son:

Flag	Significado
' # '	El valor a convertir usara la «forma alternativa» (Que se definirá más adelante)
' 0 '	La conversión rellena con ceros por la izquierda para valores numéricos.
' - '	El valor convertido se ajusta a la izquierda (Sobreescribe la conversión ' 0 ' si se especifican los dos)
' '	(Un espacio) Se deba añadir un espacio en blanco antes de un número positivo (O una cadena vacía) si se usa una conversión con signo.
' + '	Un carácter signo (' + ' o ' - ') precede a la conversión (Sobreescribe el indicador de «espacio»)

Puede estar presente un modificador de longitud (h, l o L), pero se ignora y no es necesario para Python – por lo que, por ejemplo, la salida de %ld es idéntica a %d.

Los tipos de conversión son:

Con- ver- sión	Significado	No- tas
' d '	Entero decimal con signo.	
' i '	Entero decimal con signo.	
' o '	Valor octal con signo.	(1)
' u '	Obsoleto – es idéntico a ' d '.	(8)
' x '	Hexadecimal con signo (En minúsculas)	(2)
' X '	Hexadecimal con signo (En mayúsculas)	(2)
' e '	Formato en coma flotante exponencial (En minúsculas).	(3)
' E '	Formato en coma flotante exponencial (En mayúsculas).	(3)
' f '	Formato en coma flotante decimal.	(3)
' F '	Formato en coma flotante decimal.	(3)
' g '	Formato en coma flotante. Usa formato exponencial con minúsculas si el exponente es menor que -4 o no es menor que la precisión, en caso contrario usa el formato decimal.	(4)
' G '	Formato en coma flotante. Usa formato exponencial con mayúsculas si el exponente es menor que -4 o no es menor que la precisión, en caso contrario usa el formato decimal.	(4)
' c '	Byte único (Acepta números enteros o binarios de un único byte)	
' b '	Bytes (Cualquier objeto que siga el protocolo de objetos de tipo binario o implemente el método <code>__bytes__()</code>).	(5)
' s '	' s ' es un alias de ' b ' y solo debe ser usado para bases de código Python2/3.	(6)
' a '	Bytes (Convierte cualquier objeto Python usando <code>repr(obj).encode('ascii', 'backslashreplace')</code>).	(5)
' r '	' r ' es un alias de ' a ' y solo debe ser usado para bases de código Python2/3.	(7)
' % '	No se representa ningún argumento, obteniéndose en el resultado la cadena '% '.	

Notas:

- (1) La forma alternativa hace que se inserte antes del primer dígito un prefijo indicativo del formato octal ('0o')
- (2) La forma alternativa hace que se inserte antes del primer dígito uno de los dos prefijos indicativos del formato hexadecimal '0x' or '0X' (Que se use uno u otro depende de que indicador de formato se haya usado, 'x' or 'X').
- (3) La forma alternativa hace que se incluya siempre el símbolo del punto o coma decimal, incluso si no hubiera dígitos después.
La precisión determina el número de dígitos que vienen después del punto decimal, y por defecto es 6.
- (4) La forma alternativa hace que se incluya siempre el símbolo del punto o coma decimal, y los ceros a su derecha no se eliminan, como sería el caso en la forma normal.
La precisión determina el número de dígitos significativos que vienen antes y después del punto decimal, y por defecto es 6.
- (5) Si la precisión es N, la salida se trunca a N caracteres.
- (6) b'%s' está obsoleto, pero no se retirará durante la serie 3.x.
- (7) b'%r' está obsoleto, pero no se retirará durante la serie 3.x.
- (8) Véase [PEP 237](#).

Nota: La versión `bytearray` de este método *no* modifica los valores internamente (no opera *in place*): siempre produce un nuevo objeto, aun si no se hubiera realizado ningún cambio.

Ver también:

[PEP 461](#) - Añadir formateo usando % con bytes y bytearray

Nuevo en la versión 3.5.

4.8.5 Vistas de memoria

Los objetos de tipo `memoryview` permiten al código Python acceder a los datos internos de objetos que soporten el protocolo buffer sin necesidad de hacer copias.

class `memoryview` (*obj*)

Crea un `memoryview` que referencia *obj*. La variable *obj* debe soportar el protocolo buffer. Los tipos de datos básicos que soportan el protocolo buffer incluyen los `bytes` y `bytearray`.

La clase `memoryview` usa el concepto de *elemento*, que es la unidad de memoria atómica gestionada por el objeto original *obj*. Para muchos tipos de datos simples como `bytes` y `bytearray`, un elemento es un único byte, pero otros tipos, como la clase `array.array` pueden tener elementos más grandes.

El resultado de `len(view)` es igual a la longitud de `tolist`. Si `view.ndim = 0`, la longitud es 1. Si `view.ndim = 1`, la longitud es igual al número de elementos en la vista. Para dimensiones superiores, la longitud es igual a la de la representación como lista anidada de la vista. El atributo `itemsize` contiene el número de bytes que ocupa un único elemento.

Un objeto de tipo `memoryview` soporta operaciones de rebanado y acceso por índices a sus datos. Un rebanado unidimensional producirá una sub-*vista*:

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> bytes(v[1:4])
b'bce'
```

Si *format* es uno de los especificadores de formato nativos del módulo *struct*, el indexado con un número entero o una tupla de números enteros también es posible, y retorna un único *elemento* con el tipo adecuado. Objetos *memoryview* unidimensionales pueden ser indexados con un entero o con una tupla de enteros. Los *memoryview* con múltiples dimensiones pueden ser indexados con tuplas de exactamente *ndim* enteros, donde *ndim* es el número de dimensiones. Vistas *memoryviews* con cero dimensiones pueden ser indexados con una tupla vacía.

Aquí hay un ejemplo con un formato que no es un byte:

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

Si el objeto usado para crear la vista es modificable, la vista *memoryview* soporta asignación unidimensional mediante rebanadas. Sin embargo, no se permite el cambio de tamaño:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different structures
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

Los objetos *memoryviews* de una única dimensión que contienen tipos de datos *hashables* (De solo lectura) con formatos 'B', 'b' o 'c' son también *hashables*. El *hash* se define como `hash(m) == hash(m.tobytes())`:

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::2]) == hash(b'abcefg'[::2])
True
```

Distinto en la versión 3.3: Los objetos *memoryviews* de una única dimensión pueden ahora ser usados con operaciones de rebanado. Los objetos *memoryviews* de una única dimensión con formatos 'B', 'b' o 'c' son ahora *hashables*.

Distinto en la versión 3.4: los objetos *memoryview* son registrados automáticamente con la clase `collections.abc.Sequence`

Distinto en la versión 3.5: los objetos *memoryviews* se pueden ahora acceder usando como índices una tupla de números enteros.

La clase *memoryview* tiene varios métodos:

`__eq__` (*exporter*)

Un objeto *memoryview* y un exportador **PEP 3118** son iguales si sus formas son equivalentes y todos los valores correspondientes son iguales cuando los formatos respectivos de los operandos son interpretados usando la sintaxis de *struct*.

Para el subconjunto de formatos de *struct* soportados actualmente por *tolist()*, *v* y *w* son iguales si `v.tolist() == w.tolist()`:

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

Si cualquiera de las cadenas de formato no es soportada por el módulo *struct*, entonces la comparación de los objetos siempre los considerará diferentes (Incluso si las cadenas de formato y el contenido del *buffer* son idénticos):

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

Nótese que, al igual que con los números en coma flotante, *v is w* no implica que *v == w* para objetos del tipo *memoryview*.

Distinto en la versión 3.3: Versiones previas comparaban la memoria directamente, sin considerar ni el formato de los elementos ni la estructura lógica de *array*.

`tobytes` (*order=None*)

Retorna los datos en el *buffer* en forma de cadena de bytes. Equivale a llamar al constructor de la clase *bytes* en el objeto *memoryview*:

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

Para *arrays* no contiguos el resultado es igual a la representación en forma de lista aplanada, con todos los elementos convertidos a bytes. El método *tobytes()* soporta todos los formatos de texto, incluidos aquellos que no se encuentran en la sintaxis del módulo *struct*.

Nuevo en la versión 3.8: El valor de *order* puede ser {"C", "F", "A"}. Cuando *order* es "C" o "F", los datos en el *array* original se convierten al orden de C o Fortran. Para vistas contiguas, "A" retorna una copia exacta de la memoria física. En particular, el orden en memoria de Fortran se mantiene inalterado. Para vista no contiguas, los datos se convierten primero a C. Definir *order=None* es lo mismo que *order="C"*.

hex (*sep*, *bytes_per_sep*)

Retorna una cadena de caracteres que contiene dos dígitos hexadecimales por cada byte en el *buffer*:

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

Nuevo en la versión 3.5.

Distinto en la versión 3.8: De forma similar a *bytes.hex()*, *memoryview.hex()* soporta ahora los parámetros opcionales *sep* y *bytes_per_sep* para insertar separadores entre los bytes en la cadena hexadecimal de salida.

tolist ()

Retorna los datos en el *buffer* como una lista de elementos.

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

Distinto en la versión 3.3: El método *tolist()* soporta ahora todos los formatos nativos de un solo carácter definidos en el módulo *struct*, así como las representaciones de múltiples dimensiones.

toreadonly ()

Retorna una versión de solo lectura del objeto *memoryview*. El objeto original permanece inalterado:

```
>>> m = memoryview(bytearray(b'abc'))
>>> mm = m.toreadonly()
>>> mm.tolist()
[89, 98, 99]
>>> mm[0] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> m[0] = 43
>>> mm.tolist()
[43, 98, 99]
```

Nuevo en la versión 3.8.

release ()

Libera el buffer subyacente expuesto por el objeto *memoryview*. Muchos objetos realizan operaciones especiales cuando una vista los está conteniendo (Por ejemplo, un objeto *bytearray* temporalmente prohíbe el cambio de tamaño); la llamada a *release()* sirve para eliminar estas restricciones (Así como para tratar con los recursos pendientes) lo más pronto posible.

Después de que se ha llamado a este método, cualquier operación posterior sobre la vista producirá una excepción de tipo *ValueError* (Excepto por el propio método *release()*, que puede ser llamado las veces que se quiera):

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
```

(continué en la próxima página)

(proviene de la página anterior)

```
File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

El protocolo de gestión de contexto puede ser usado para obtener un efecto similar, usando la sentencia `with`:

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

Nuevo en la versión 3.2.

cast (*format* [, *shape*])

Transforma el formato o el tamaño de un objeto *memoryview*. El parámetro *shape* por defecto vale `[byte_length//new_itemsize]`, lo que significa que el resultado será unidimensional. El valor de retorno es un nuevo objeto de tipo *memoryview*, pero el buffer en sí no se copia. Las transformaciones pueden ser 1D -> C-contiguo y C-contiguo -> 1D.

El formato de destino está restringido a un único elemento de formato nativo en la sintaxis de *struct*. Uno de los formatos debe ser un formato de byte ('B', 'b' o 'c'). La longitud en bytes del resultado debe coincidir con la longitud original.

Transforma de 1D/long a bytes 1D/unsigned:

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

Transforma de 1D/unsigned a bytes 1D/char:

```
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value for format "B"
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

Transforma de 1D/bytes a 3D/ints a caracteres 1D/signed:

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

Transforma de *long* 1D/unsigned a *long* 2D/unsigned:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

Nuevo en la versión 3.3.

Distinto en la versión 3.5: El formato de origen ya no está restringido cuando se transforma a una vista de bytes.

Hay disponibles varios atributos de solo lectura:

obj

El objeto subyacente del *memoryview*:

```
>>> b = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

Nuevo en la versión 3.3.

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`. Este es el espacio, medido en bytes, que usará el *array* en una representación continua. No tiene que ser necesariamente igual a `len(m)`:

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
```

(continué en la próxima página)

(proviene de la página anterior)

```

>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12

```

Matrices de múltiples dimensiones:

```

>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96

```

Nuevo en la versión 3.3.

readonly

Un booleano que indica si la memoria es de solo lectura.

format

Una cadena de caracteres que contiene el formato (En el estilo del módulo *struct*) para cada elemento de la vista. Un objeto *memoryview* se puede crear a partir de un exportador con textos de formato arbitrarios, pero algunos métodos (Como, por ejemplo, *tolist()*) están restringidos a usar formatos de elementos nativos sencillos.

Distinto en la versión 3.3: el formato 'B' se gestiona ahora de acuerdo a la sintaxis descrita en el módulo *struct*. Esto significa que `memoryview(b'abc')[0] == b'abc'[0] == 97`.

itemsize

El tamaño en bytes de cada elemento del objeto *memoryview*:

```

>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True

```

ndim

Un número entero que indica cuantas dimensiones de una matriz multi-dimensional representa la memoria.

shape

Una tupla de números enteros, de longitud *ndim*, que indica la forma de la memoria en una matriz de *N* dimensiones.

Distinto en la versión 3.3: Una tupla vacía, en vez de *None*, cuando *ndim* = 0.

strides

Una tupla de números enteros, de longitud *ndim*, que indica el tamaño en bytes para acceder a cada

dimensión de la matriz.

Distinto en la versión 3.3: Una tupla vacía, en vez de `None`, cuando `ndom = 0`.

suboffsets

De uso interno para las matrices estilo *PIL*. El valor es solo informativo.

c_contiguous

Un booleano que indica si la memoria es *contiguous* al estilo *C*.

Nuevo en la versión 3.3.

f_contiguous

Un booleano que indica si la memoria es *contiguous* al estilo Fortran.

Nuevo en la versión 3.3.

contiguous

Un booleano que indica si la memoria es *contiguous*.

Nuevo en la versión 3.3.

4.9 Conjuntos — `set`, `frozenset`

Un objeto de tipo *conjunto* o *set* es una colección no ordenada de distintos objetos *hashable*. Los casos de uso habituales incluyen comprobar la pertenencia al conjunto de un elemento, eliminar duplicados de una secuencia y realizar operaciones matemáticas como la intersección, la unión, la diferencia o la diferencia simétrica (Para otros tipos de contenedores véanse las clases básicas *dict*, *list*, y *tuple*, así como el módulo *collections*).

Como otras colecciones, los conjuntos soportan `x in set`, `len(set)` y `for x in set`. Como es una colección sin orden, los conjuntos no registran ni la posición ni el orden de inserción de los elementos. Por lo mismo, los conjuntos no soportan indexado, ni operaciones de rebanadas, ni otras capacidades propias de las secuencias.

En la actualidad hay dos tipos básicos de conjuntos: *set* y *frozenset*. La clase *set* es mutable, es decir, el contenido del conjunto puede ser modificado con métodos como `add()` y `remove()`. Como es mutable, no tiene un valor de *hash* y no pueden ser usados como claves de diccionarios ni como elementos de otros conjuntos. La clase *frozenset* es inmutable y *hashable*, es decir, que sus contenidos no pueden ser modificados después de creados. Puede ser usado, por tanto, como claves de diccionario o como elemento de otro conjunto.

Se pueden crear conjuntos no vacíos (*sets*, no *frozensets*) escribiendo una lista de elementos separados por comas, entre llaves, por ejemplo `{'jack', 'sjoerd'}`, además de con el constructor de la clase *set*.

El constructor para ambas clases se usa de la misma forma:

```
class set ([iterable])
class frozenset ([iterable])
```

Retorna un nuevo *set* o *frozenset*, tomando los elementos a partir de *iterable*. Los elementos de un conjunto tienen que tener la propiedad de ser *hashable*. Para representar conjuntos anidados, o conjuntos de conjuntos, los conjuntos interiores tienen que ser instancias de *frozenset*. Si no se especifica el parámetro *iterable*, se retorna un conjunto vacío.

Sets can be created by several means:

- Use a comma-separated list of elements within braces: `{'jack', 'sjoerd'}`
- Use a set comprehension: `{c for c in 'abracadabra' if c not in 'abc'}`
- Use the type constructor: `set()`, `set('foobar')`, `set(['a', 'b', 'foo'])`

Las instancias de *set* y *frozenset* proporcionan las siguientes operaciones:

len(s)

Retorna el número de elementos en el conjunto *s* (Cardinalidad de *s*)

x in s

Comprueba que el elemento *x* está incluido en *s*.

x not in s

Comprueba que el elemento *x* no está incluido en *s*.

isdisjoint (other)

Retorna `True` si el conjunto no tienen ningún elemento en común con *other*. Dos conjuntos son disjuntos si y solo si su intersección es el conjunto vacío.

issubset (other)

set <= other

Comprueba si cada elemento del conjunto también se encuentra en *other*.

set < other

Comprueba si el conjunto es un subconjunto propio de *other*, es decir, `set <= other and set != other`.

issuperset (other)

set >= other

Comprueba que cada elemento de *other* está incluido en el conjunto.

set > other

Comprueba si el conjunto es un superconjunto propio de *other*, es decir, `set >= other and set != other`.

union (*others)

set | other | ...

Retorna un conjunto nuevo que contiene todos los elementos del conjunto y de *others*.

intersection (*others)

set & other & ...

Retorna un conjunto nuevo que contiene todos los elementos que están a la vez en conjunto y en *others*.

difference (*others)

set - other - ...

Retorna un conjunto nuevo que contiene todos los elementos del conjunto y que no están incluidos en *others*.

symmetric_difference (other)

set ^ other

Retorna un conjunto nuevo que contiene elementos que están incluidos en el conjunto o en *others*, pero no en los dos a la vez.

copy ()

Retorna una copia superficial del conjunto.

Hay que señalar que las versiones de las operaciones que son métodos (no los operadores) como `union()`, `intersection()`, `difference()`, `symmetric_difference()`, `issubset()`, y `issuperset()` aceptan cualquier iterable como parámetro. Por el contrario, los operadores requieren que los argumentos sean siempre conjuntos. Esto evita ciertas construcciones propensas a errores como `set('abc') & 'cbs'`, favoreciendo el uso formas más legibles como `set('abc').intersection('cbs')`.

Ambas clases `set` y `frozenset` soportan comparaciones entre sí. Dos conjuntos son iguales si y solo si cada elemento de cada conjunto está incluido en el otro (Cada uno de ellos es subconjunto del otro). Un conjunto es menor que otro si y solo si el primero es un subconjunto propio del segundo (Es un subconjunto, pero no son iguales). Un conjunto es mayor que otro si y solo si el primero es un superconjunto propio del segundo (Es un superconjunto, pero no son iguales).

Las instancias de `set` se comparan con las instancias de `frozenset` en base a sus elementos. Por ejemplo `set('abc') == frozenset('abc')` retorna `True` y lo mismo hace `set('abc') in set([frozenset('abc')])`.

Las comparaciones de subconjunto e igualdad no son tan generales que permitan una función de ordenación total. Por ejemplo, dos conjuntos cualesquiera que no estén vacíos y que sean disjuntos no son iguales y tampoco son subconjuntos uno del otro, así que todas estas operaciones retornan `False`: `a < b`, `a == b`, or `a > b`.

Como los conjuntos solo definen un orden parcial (Relaciones de conjuntos), la salida del método `list.sort()` no está definida para listas de conjuntos.

Los elementos de un conjunto, al igual que las claves de un diccionario, deben ser *hashable*.

Las operaciones binarias que mezclan instancias de *set* y *frozenset* retornan el tipo del primer operando. Por ejemplo `frozenset('ab') | set('bc')` retornará una instancia de *frozenset*.

La siguiente tabla muestra las operaciones disponibles para la clase *set* que no son aplicables a los conjuntos inmutables *frozenset*:

update (*others)

set |= other | ...

Actualiza el conjunto, añadiendo los elementos que se encuentren en *others*.

intersection_update (*others)

set &= other & ...

Actualiza el conjunto, manteniendo solo los elementos que se encuentren en si mismo y en *others*.

difference_update (*others)

set -= other | ...

Actualiza el conjunto, eliminado los elementos que se encuentren en *others*.

symmetric_difference_update (other)

set ^= other

Actualiza el conjunto, manteniendo solo los elementos que se encuentren en el conjunto o en *others*, pero no en los dos a la vez.

add (elem)

Añade al conjunto el elemento *elem*.

remove (elem)

Elimina del conjunto el elemento *elem*. Eleva la excepción *KeyError* si *elem* no estaba incluido en el conjunto.

discard (elem)

Elimina del conjunto el elemento *elem*, si estuviera incluido.

pop ()

Elimina y retorna un elemento cualquiera del conjunto. Eleva la excepción *KeyError* si el conjunto estaba vacío.

clear ()

Elimina todos los elementos del conjunto.

Hay que señalar que los métodos (no los operadores) `update()`, `intersection_update()`, `difference_update()`, y `symmetric_difference_update()` aceptan cualquier iterable como parámetro.

Nótese que el parámetro *elem* de los métodos `__contains__()`, `remove()` y `discard()` puede ser un conjunto. Para soportar la búsqueda por un *frozenset* equivalente se crea uno temporal a partir de *elem*.

4.10 Tipos Mapa — dict

Un objeto de tipo *mapping* relaciona valores (que deben ser *hashable*) con objetos de cualquier tipo. Los mapas son objetos mutables. En este momento solo hay un tipo estándar de mapa, los *diccionarios* (Para otros tipos contenedores, véanse las clases básicas *list*, *set*, y *tuple*, así como el módulo *collections*).

Las claves de un diccionario pueden ser *casi* de cualquier tipo. Los valores que no son *hashable*, como por ejemplo valores que contengan listas, diccionarios u otros tipo mutables (que son comparados por valor, no por referencia) no se pueden usar como claves. Los tipos numéricos, cuando se usan como claves siguen las reglas habituales de la comparación numérica: Si dos números se consideran iguales (Como 1 y 1.0), ambos valores pueden ser usados indistintamente para acceder al mismo valor (Pero hay que tener en cuenta que los ordenadores almacenan algunos números en coma flotante como aproximaciones, por lo que normalmente no es recomendable usarlos como claves).

Los diccionarios se pueden crear colocando una lista separada por comas de pares `key: value` entre llaves, por ejemplo: `{'jack': 4098, 'sjoerd': 4127}` o `{4098: 'jack', 4127: 'sjoerd'}`, o por el constructor `dict`.

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

Retorna un diccionario creado a partir de un parámetro opcional por posición, y por una serie de parámetros por nombre, también opcionales.

Dictionaries can be created by several means:

- Use a comma-separated list of `key: value` pairs within braces: `{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`
- Use a dict comprehension: `{x: x ** 2 for x in range(10)}`
- Use the type constructor: `dict()`, `dict([('foo', 100), ('bar', 200)])`, `dict(foo=100, bar=200)`

Si no se especifica el parámetro por posición, se crea un diccionario vacío. Si se pasa un parámetro por posición y es un objeto de tipo mapa, se crea el diccionario a partir de las parejas clave-valor definidos en el mapa. Si no fuera un mapa, se espera que el parámetro sea un objeto *iterable*. Cada elemento del iterable debe ser una dupla (Una tupla de dos elementos); el primer componente de la dupla se usará como clave y el segundo como valor a almacenar en el nuevo diccionario. Si una clave aparece más de una vez, el último valor será el que se almacene en el diccionario resultante.

Si se usan parámetros por nombre, los nombres de los parámetros y los valores asociados se añaden al diccionario creado a partir del parámetro por posición. Si un valor de clave ya estaba presente, el valor pasado con el parámetro por nombre reemplazara el valor del parámetro por posición.

A modo de ejemplo, los siguientes ejemplo retornan todos el mismo diccionario `{"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
```

Si queremos definir claves con parámetros por nombre, como en el primer ejemplo, entonces los valores de clave solo puede ser cadenas de texto conteniendo identificadores de Python válidos. En los otros casos, se puede usar cualquier valor como clave.

Estas son las operaciones soportados por los diccionarios (Y que, por tanto, deberían ser soportados por los tipos de mapa personalizados):

list(d)

Retorna una lista de todas las claves usadas en el diccionario *d*.

len(d)

Retorna el número de elementos almacenados en el diccionario *d*.

d[key]

Retorna el elemento dentro de *d* almacenado bajo la clave *key*. Eleva una excepción de tipo *KeyError* si la clave *key* no se encuentra en el diccionario *d*.

Si una subclase de un diccionario define el método `__missing__()` y *key* no está presente, la operación `d[key]` llama a este método pasando como parámetro el valor de *key*. La operación `d[key]` o bien retorna un valor o eleva la excepción que sea retornada por la llamada a `__missing__(key)`. Ninguna otra operación o método llama a `__missing__()`. Si el método `__missing__()` no está definido, se eleva *KeyError*. Si se define `__missing__()`, debe ser de forma obligatoria un método, no puede ser una variable de instancia:

```

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

```

El ejemplo anterior muestra parte de la implementación de la clase `collections.Counter`. Otro ejemplo de uso del método `__missing__` se puede encontrar en la clase `collections.defaultdict`.

d[key] = value

Asigna el valor *value* a `d[key]`.

del d[key]

Elimina `d[key]` de *d*. Eleva una excepción `KeyError` si *key* no está en el mapa.

key in d

Retorna `True` si *d* tiene una entrada en la clave *key*, `False` en caso contrario.

key not in d

Equivale a `not key in d`.

iter(d)

Retorna un *iterador* que recorre todas las claves de un diccionario. Es una forma abreviada de `iter(d, keys())`.

clear()

Elimina todos los contenidos del diccionario.

copy()

Retorna una copia superficial del diccionario.

classmethod fromkeys(iterable[, value])

Crea un nuevo diccionario con las claves obtenidos a partir del *iterable* y con valor *value*.

El método `fromkeys()` es un método de clase que retorna un diccionario nuevo. El valor de *value* por defecto es `None`. Todos los valores harán referencia a una única instancia, por lo que en general no tiene sentido que *value* sea un objeto mutable, como una lista vacía. Para poder obtener valores diferentes, se puede usar mejor un diccionario por comprensión.

get(key[, default])

Retorna el elemento dentro de *d* almacenado bajo la clave *key*, si *key* está en el diccionario; si no, retorna *default*. El valor de *default* por defecto es `None`, por lo que esta función nunca eleva la excepción `KeyError`.

items()

Retorna una nueva vista de los contenidos del diccionario (Pares (*key*, *value*)). Vea [documentación de los objetos vistas](#).

keys()

Retorna una nueva vista de las claves del diccionario. Véase la [documentación de las vistas](#).

pop(key[, default])

Si *key* está en el diccionario, lo elimina del diccionario y retorna su valor; si no está, retorna *default*. Si no se especifica valor para *default* y la clave no se encuentra en el diccionario, se eleva la excepción `KeyError`.

popitem()

Elimina y retorna una pareja (*key*, *value*) del diccionario. Las parejas se retornan en el orden LIFO (*last-in, first-out*: Último en entrar, primero en salir).

El método `popitem()` es útil para recorrer y a la vez vaciar un diccionario, un proceso usado a menudo en algoritmos de conjuntos. Si el diccionario está vacío, llamar a `popitem()` eleva la excepción `KeyError`.

Distinto en la versión 3.7: El orden *LIFO* ahora está garantizado. En versiones anteriores, el método `popitem()` retorna una pareja clave/valor arbitraria.

reversed(d)

Retorna un *iterador* que recorre las claves en orden inverso. Es una forma abreviada de `reversed(d.keys())`.

Nuevo en la versión 3.8.

setdefault(key[, default])

Si `key` está incluida en el diccionario, retorna el valor almacenado. Si no, inserta con la clave `key` el valor definido en `default` y retorna `default`. El valor por defecto de `default` es `None`.

update([other])

Actualiza el diccionario con las parejas clave/valor obtenidas de `other`, escribiendo encima de las claves existentes. retorna `None`.

El método `update()` acepta tanto un diccionario como un iterable que Retorna parejas de claves, valor (ya sea como tuplas o como otros iterables de longitud 2). Si se especifican parámetros por nombre, el diccionario se actualiza con esas combinaciones de clave y valor: `d.update(red=1, blue=2)`.

values()

Retorna una nueva vista de los valores del diccionario. Véase la [documentación sobre objetos vistas](#).

Una comparación de igualdad entre una vista `dict.values()` y otra siempre retornará `False`. Esto también pasa cuando se compara `dict.values()` consigo mismo:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

Los diccionarios se consideran iguales si y solo si tienen el mismo conjunto de parejas (`key`, `value`) (Independiente de su orden). Los intentos de comparar usando los operadores “<”, “<=”, “>=”, “>” elevan una excepción de tipo `TypeError`.

Los diccionarios mantiene de forma interna el orden de inserción. Actualizar una entrada no modifica ese orden. Las claves que vuelven a ser insertadas después de haber sido borradas se añaden al final.:

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

Distinto en la versión 3.7: Se garantiza que el orden del diccionario es el de inserción. Este comportamiento era un detalle de implementación en CPython desde la versión 3.6.

Tanto los diccionarios como las vistas basadas en diccionarios son reversibles:

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> list(reversed(d))
['four', 'three', 'two', 'one']
>>> list(reversed(d.values()))
[4, 3, 2, 1]
>>> list(reversed(d.items()))
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

Distinto en la versión 3.8: Los diccionarios son ahora reversibles.

Ver también:

Se puede usar un objeto de tipo `types.MappingProxyType` para crear una vista de solo lectura de un objeto `dict`.

4.10.1 Objetos tipos vista de diccionario

Los objetos retornados por los métodos `dict.keys()`, `dict.values()` y `dict.items()` son objetos tipo vista o *view*. Estos objetos proporcionan una vista dinámica del contenido del diccionario, lo que significa que si el diccionario cambia, las vistas reflejan estos cambios.

Las vistas de un diccionario pueden ser iteradas para retornar sus datos respectivos, y soportan operaciones de comprobación de pertenencia:

`len(dictview)`

Retorna el número de entradas en un diccionario.

`iter(dictview)`

Retorna un *iterador* sobre las claves, valores o elementos (representados en forma de tuplas `(key, value)`) de un diccionario.

Las claves y los valores se iteran en orden de inserción. Esto permite la creación de parejas `(value, key)` usando la función `zip()`: `pairs = zip(d.values(), d.keys())`. Otra forma de crear la misma lista es `pairs = [(v, k) for (k, v) in d.items()]`.

Iterar sobre un diccionario a la vez que se borran o añaden entradas puede elevar una excepción de tipo `RuntimeError`, o puede provocar que no se iteren sobre todas las entradas.

Distinto en la versión 3.7: Se garantiza que el orden de los diccionarios es el de inserción.

`x in dictview`

Retorna `True` si `x` está incluido en las claves, los valores o los elementos del diccionario. En el último caso, `x` debe ser una tupla `(key, value)`.

`reversed(dictview)`

Retorna un iterador inverso sobre las claves y valores del diccionario. El orden de la vista sera el inverso del orden de inserción.

Distinto en la versión 3.8: Las vistas de un diccionario no son reversibles.

Las vistas de claves son similares a conjuntos, dado que todas las claves deben ser únicas y *hashables*. Si todos los valores son también *hashables*, de forma que las parejas `(key, value)` son también únicas y *hashables*, entonces la vista *items* es también similar a un conjunto (La vista *values* no son consideradas similar a un conjunto porque las valores almacenados en el diccionario no tiene que ser únicos). Las vistas similares a conjuntos pueden usar todas las operaciones definidas en la clase abstracta `collections.abc.Set`, como por ejemplo `==`, `<`, or `^`.

Un ejemplo de uso de una vista de diccionario:

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

>>> # iteration
>>> n = 0
```

(continué en la próxima página)

(proviene de la página anterior)

```

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

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

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

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}

```

4.11 Tipos Gestores de Contexto

La expresión `with` de Python soporta el concepto de un contexto en tiempo de ejecución definido mediante un gestor de contexto. Para implementar esto, se permite al usuario crear clases para definir estos contextos definiendo dos métodos, uno a ejecutar ante de entrar del bloque de código y otro a ejecutar justo después de salir del mismo:

`contextmanager.__enter__()`

Entra en el contexto en tiempo de ejecución, y retorna o bien este objeto u otro relacionado con el contexto. El valor retornado por este método se vincula al identificador que viene tras la palabra clave `as` usada en la sentencia `with` que define el contexto.

Un ejemplo de gestor de contexto que se retorna a si mismo es un objeto de tipo *file object*. Los objetos de tipo `File` se retornan a si mismo en la llamada a `__enter__()`, lo que permite que `open()` sea usado como gestores de contexto en una sentencia `with`.

Un ejemplo de gestor de contexto que retorna otro objeto relacionado en el que define la función `decimal.localcontext()`. Este gestor define el contexto de uso en operaciones decimales a partir de una copia del contexto original, y retorna esa copia. De esta manera se puede cambiar el contexto decimal dentro del cuerpo del `with` sin afectar al código fuera del mismo.

`contextmanager.__exit__(exc_type, exc_val, exc_tb)`

Sale del contexto y retorna un indicador booleano que indica si se debe ignorar cualquier posible excepción que hubiera ocurrido dentro del mismo. Si se produce una excepción mientras se ejecutan las sentencias definidas en el cuerpo de la sentencia `with`, los parámetros de esta función contienen el tipo y valor de la excepción, así como la información relativa a la traza de ejecución. Si no se produce ninguna excepción, los tres parámetros valen `None`.

Si este método retorna un valor `True`, la sentencia `with` ignora la excepción y el flujo del programa continúa con la primera sentencia inmediatamente después del cuerpo. En caso contrario la excepción producida continúa propagándose después de que este método termine de ejecutarse. Cualquier excepción que pudieran producirse dentro de este método reemplaza a la excepción que se hubiera producido en el cuerpo del `with`.

La excepción pasada nunca debe volver a elevarse explícitamente; en vez de eso, el método debería retornar un valor falso para indicar que el método ha terminado de ejecutarse sin problemas y que no se desea suprimir la excepción. Esto permite a los gestores de contexto detectar fácilmente si el método `__exit__()` ha podido terminar o no.

Python define varios gestores de contexto para facilitar la sincronía entre hilos, asegurarse del cierre de ficheros y otros objetos similares y para modificar de forma simple el contexto para las expresiones aritméticas con decimales. Los tipos específicos no se tratan especialmente fuera de su implementación del protocolo de administración de contexto. Ver el módulo `contextlib` para algunos ejemplos.

Los *generator* y el decoradores definidos en la clase `contextlib.contextmanager` permiten implementar de forma sencilla estos protocolos. Si una función generadora se decora con la clase `contextlib.contextmanager`, retornará un gestor de contexto que incluye los necesarios métodos `__enter__()` y `__exit__()`, en vez del *iterador* que se produciría si no se decora la función.

Nótese que no hay una ranura específica para ninguno de estos métodos en la estructura usada para los objetos Python en el nivel de la API C. Objetos que quieran definir estos métodos deben implementarlos como métodos normales de Python. Comparado con la complejidad de definir un contexto en tiempo de ejecución, lo complejidad de una búsqueda simple en un diccionario es mínima.

4.12 Otros tipos predefinidos

El intérprete soporta otros tipos de objetos variados. La mayoría de ellos solo implementan una o dos operaciones.

4.12.1 Módulos

La única operación especial que implementan los módulos es el acceso como atributos: `m.name`, donde *m* es un módulo y *name* accede a un nombre definido en la tabla de símbolos del módulo *m*. También se puede asignar valores a los atributos de un módulo (Nótese que la sentencia `import` no es, estrictamente hablando, una operación del objeto de tipo módulo. La sentencia `import foo` no requiere la existencia de un módulo llamado *foo*, sino una *definición* (externa) de un módulo *foo* en alguna parte).

Un atributo especial de cada módulo es `__dict__`. Es un diccionario que contiene la tabla de símbolos del módulo. Cambiar el diccionario cambiará por tanto el contenido de la tabla de símbolos, pero no es posible realizar una asignación directa al atributo `__dict__` (Se puede realizar una asignación como `m.__dict__['a'] = 1`, que define el valor de `m.a` como 1, pero no se puede hacer `m.__dict__ = {}`). No se recomienda manipular los contenidos del atributo `__dict__` directamente.

Los módulos incluidos en el intérprete se escriben así: `<module 'sys' (built-in)>`. Si se cargan desde un archivo, se escriben como `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`.

4.12.2 Clases e instancias de clase

Véase `objects` y `class` para más información.

4.12.3 Funciones

Los objetos de tipo función se crean mediante definiciones de función. La única operación posible con un objeto de tipo función es llamarla: `func(argument-list)`.

Hay dos tipos de funciones: Las funciones básicas o predefinidas y las funciones definidas por el usuario. Las dos soportan la misma operación (ser llamadas), pero la implementación es diferente, de ahí que se consideren de distintos tipo.

Véase `function` para más información.

4.12.4 Métodos

Los métodos son funciones que se llaman usando la notación de atributos. Hay de dos tipos: métodos básicos o predefinidos (Como el método `append()` en las listas) y métodos de instancia de clase. Los métodos básicos o predefinidos se describen junto con los tipos que los soportan.

Si se accede a un método (Una función definida dentro de un espacio de nombres de una clase) a través de una instancia, se obtiene un objeto especial, un *método ligado* (También llamado *método de instancia*). Cuando se llama, se añade automáticamente el parámetro `self` a la lista de parámetros. Los métodos ligados tienen dos atributos especiales de solo lectura: `m.__self__` es el objeto sobre el que está operando el método, y `m.__func__` es la función que implementa el método.

Al igual que los objetos de tipo función, los métodos ligados o de instancia soportan asignación de atributos arbitrarios. Sin embargo, como los atributos de los métodos se almacenan en la función subyacente (`meth.__func__`), definir cualquier atributo en métodos ligados está desaconsejado. Intentar asignar un atributo a un método produce que se eleve una excepción de tipo `AttributeError`. Para poder definir un atributo a un método, este debe ser definido explícitamente en la función subyacente:

```
>>> 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: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

Véase `types` para más información.

4.12.5 Objetos código

Los objetos de tipo código son usados por la implementación del lenguaje para representar código ejecutable «pseudo-compilado», como por ejemplo el cuerpo de una función. A diferencia de los objetos de tipo función, no contienen una referencia a un entorno global de ejecución. Los objetos de tipo código se pueden obtener usando la función básica `compile()` o se pueden extraer a partir de objetos de tipo función a través de su atributo `__code__`. Para más detalle véase el módulo `code`.

Accessing `__code__` raises an *auditing event* object. `__getattr__` with arguments `obj` and `"__code__"`.

Un objeto de tipo código puede ser evaluado o ejecutando pasándolo como parámetros a las funciones básicas `exec()` o `eval()` (Que también aceptan código Python en forma de cadena de texto).

Véase `types` para más información.

4.12.6 Objetos Tipo

Los objetos de tipo Tipo (*Type*) representan a los distintos tipos de datos. El tipo de un objeto particular puede ser consultado usando la función básica `type()`. Los objetos Tipo no tienen ninguna operación especial. El módulo `types` define nombres para todos los tipos básicos definidos en la biblioteca estándar.

Los tipos se escriben de la siguiente forma: `<class 'int'>`.

4.12.7 El objeto nulo (*Null*)

Todas las funciones que no definen de forma explícita un valor de retorno retornan este objeto. Los objetos nulos no soportan ninguna operación especial. Solo existe un único objeto nulo, llamado `None` (Un nombre predefinido o básico). La expresión `type(None)()` produce el mismo objeto `None`, esto se conoce como *Singleton*.

Se escribe `None`.

4.12.8 El objeto puntos suspensivos (*Ellipsis*)

Este objeto es usado a menudo en operaciones de rebanadas (Véase *slicings*). No soporta ninguna operación especial. Solo existe un único objeto de puntos suspensivos, llamado `Ellipsis` (Un nombre predefinido o básico). La expresión `type(Ellipsis)()` produce el mismo objeto `Ellipsis`, esto se conoce como *Singleton*.

Se puede escribir como `Ellipsis` o `...`.

4.12.9 El objeto *NotImplemented*

Este objeto se retorna en todas las operaciones binarias y comparaciones cuando se intenta operar con tipos que no están soportados. Véase *comparisons* para más información. Solo existe un objeto de tipo `NotImplemented`. La expresión `type(NotImplemented)()` produce el mismo objeto, esto se conoce como *Singleton*.

Se escribe `NotImplemented`.

4.12.10 Valores booleanos

Los valores booleanos o lógicos son los dos objetos constantes `False` y `True`. Se usan para representar valores de verdad (Aunque otros valores pueden ser considerados también como verdaderos o falsos). En contextos numéricos (Por ejemplo, cuando se usan como argumentos de una operación aritmética) se comportan como los números enteros 0 y 1 respectivamente. Se puede usar la función incorporada `bool()` para convertir valores de cualquiera tipo a Booleanos, si dicho valor puede ser interpretado como valores verdaderos/falsos (Véase la sección *Evaluar como valor verdadero/falso* anterior).

Se escriben `False` y `True` respectivamente.

4.12.11 Objetos internos

Véase la sección *types* para saber más de estos objetos. Se describen los objetos marco de pila, los objetos de traza de ejecución (*traceback*) y los objetos de tipo rebanada (*slice*).

4.13 Atributos especiales

La implementación añade unos cuantos atributos de solo lectura a varios tipos de objetos, cuando resulta relevante. Algunos de estos atributos son reportados por la función incorporada `dir()`.

`object.__dict__`

Un diccionario u otro tipo de mapa usado para almacenar los atributos de un objeto (Si son modificables).

`instance.__class__`

La clase a la que pertenece una instancia.

`class.__bases__`

La tupla de clases base de las que deriva una clase.

`definition.__name__`

El nombre de la clase, función, método, descriptor o instancia generadora.

definition. `__qualname__`

El nombre calificado (*qualified name*) de la clase, función, método, descriptor o instancia generadora.

Nuevo en la versión 3.3.

class. `__mro__`

Este atributo es una tupla de las clases que serán consideradas cuando se busque en las clases base para resolver un método.

class. `mro()`

Este método puede ser reescrito por una *metaclass* para personalizar el orden de resolución de métodos para sus instancias. Es llamado en la creación de la clase, y el resultado se almacena en el atributo `__mro__`.

class. `__subclasses__()`

Cada clase mantiene una lista de referencias débiles a sus subclase inmediatamente anteriores. Este método retorna una lista de todas las referencias que todavía estén vivas. Por ejemplo:

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

4.14 Integer string conversion length limitation

CPython has a global limit for converting between `int` and `str` to mitigate denial of service attacks. This limit *only* applies to decimal or other non-power-of-two number bases. Hexadecimal, octal, and binary conversions are unlimited. The limit can be configured.

The `int` type in CPython is an arbitrary length number stored in binary form (commonly known as a «bignum»). There exists no algorithm that can convert a string to a binary integer or a binary integer to a string in linear time, *unless* the base is a power of 2. Even the best known algorithms for base 10 have sub-quadratic complexity. Converting a large value such as `int('1' * 500_000)` can take over a second on a fast CPU.

Limiting conversion size offers a practical way to avoid [CVE-2020-10735](#).

The limit is applied to the number of digit characters in the input or output string when a non-linear conversion algorithm would be involved. Underscores and the sign are not counted towards the limit.

When an operation would exceed the limit, a `ValueError` is raised:

```
>>> import sys
>>> sys.set_int_max_str_digits(4300) # Illustrative, this is the default.
>>> _ = int('2' * 5432)
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300) for integer string conversion: value has 5432_
↳digits; use sys.set_int_max_str_digits() to increase the limit.
>>> i = int('2' * 4300)
>>> len(str(i))
4300
>>> i_squared = i*i
>>> len(str(i_squared))
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300) for integer string conversion: value has 8599_
↳digits; use sys.set_int_max_str_digits() to increase the limit.
>>> len(hex(i_squared))
7144
>>> assert int(hex(i_squared), base=16) == i*i # Hexadecimal is unlimited.
```

The default limit is 4300 digits as provided in `sys.int_info.default_max_str_digits`. The lowest limit that can be configured is 640 digits as provided in `sys.int_info.str_digits_check_threshold`.

Verification:

```
>>> import sys
>>> assert sys.int_info.default_max_str_digits == 4300, sys.int_info
>>> assert sys.int_info.str_digits_check_threshold == 640, sys.int_info
>>> msg = int('578966293710682886880994035146873798396722250538762761564'
...          '9252925514383915483333812743580549779436104706260696366600'
...          '571186405732').to_bytes(53, 'big')
...
...
```

Nuevo en la versión 3.8.14.

4.14.1 Affected APIs

The limitation only applies to potentially slow conversions between *int* and *str* or *bytes*:

- `int(string)` with default base 10.
- `int(string, base)` for all bases that are not a power of 2.
- `str(integer)`.
- `repr(integer)`.
- any other string conversion to base 10, for example `f"{integer}", "{}".format(integer)`, or `b"%d" % integer`.

The limitations do not apply to functions with a linear algorithm:

- `int(string, base)` with base 2, 4, 8, 16, or 32.
- `int.from_bytes()` and `int.to_bytes()`.
- `hex()`, `oct()`, `bin()`.
- *Especificación de formato Mini-Lenguaje* for hex, octal, and binary numbers.
- *str* to *float*.
- *str* to *decimal.Decimal*.

4.14.2 Configuring the limit

Before Python starts up you can use an environment variable or an interpreter command line flag to configure the limit:

- `PYTHONINTMAXSTRDIGITS`, e.g. `PYTHONINTMAXSTRDIGITS=640 python3` to set the limit to 640 or `PYTHONINTMAXSTRDIGITS=0 python3` to disable the limitation.
- `-X int_max_str_digits`, e.g. `python3 -X int_max_str_digits=640`
- `sys.flags.int_max_str_digits` contains the value of `PYTHONINTMAXSTRDIGITS` or `-X int_max_str_digits`. If both the env var and the `-X` option are set, the `-X` option takes precedence. A value of `-1` indicates that both were unset, thus a value of `sys.int_info.default_max_str_digits` was used during initialization.

From code, you can inspect the current limit and set a new one using these *sys* APIs:

- `sys.get_int_max_str_digits()` and `sys.set_int_max_str_digits()` are a getter and setter for the interpreter-wide limit. Subinterpreters have their own limit.

Information about the default and minimum can be found in `sys.int_info`:

- `sys.int_info.default_max_str_digits` is the compiled-in default limit.
- `sys.int_info.str_digits_check_threshold` is the lowest accepted value for the limit (other than 0 which disables it).

Nuevo en la versión 3.8.14.

Prudencia: Setting a low limit *can* lead to problems. While rare, code exists that contains integer constants in decimal in their source that exceed the minimum threshold. A consequence of setting the limit is that Python source code containing decimal integer literals longer than the limit will encounter an error during parsing, usually at startup time or import time or even at installation time - anytime an up to date `.pyc` does not already exist for the code. A workaround for source that contains such large constants is to convert them to `0x` hexadecimal form as it has no limit.

Test your application thoroughly if you use a low limit. Ensure your tests run with the limit set early via the environment or flag so that it applies during startup and even during any installation step that may invoke Python to precompile `.py` sources to `.pyc` files.

4.14.3 Recommended configuration

The default `sys.int_info.default_max_str_digits` is expected to be reasonable for most applications. If your application requires a different limit, set it from your main entry point using Python version agnostic code as these APIs were added in security patch releases in versions before 3.11.

Example:

```
>>> import sys
>>> if hasattr(sys, "set_int_max_str_digits"):
...     upper_bound = 68000
...     lower_bound = 4004
...     current_limit = sys.get_int_max_str_digits()
...     if current_limit == 0 or current_limit > upper_bound:
...         sys.set_int_max_str_digits(upper_bound)
...     elif current_limit < lower_bound:
...         sys.set_int_max_str_digits(lower_bound)
```

If you need to disable it entirely, set it to 0.

Notas al pie

Excepciones incorporadas

En Python, todas las excepciones deben ser instancias de una clase que se derive de `BaseException`. En una instrucción `try` con una cláusula `except` que menciona una clase determinada, esa cláusula también controla las clases de excepción derivadas de esa clase (excepto las clases de excepción de las que se deriva *it*). Dos clases de excepción que no están relacionadas mediante subclases nunca son equivalentes, incluso si tienen el mismo nombre.

Las excepciones predefinidas enumeradas a continuación pueden ser generadas por el intérprete o funciones predefinidas. Excepto donde se mencione lo contrario, tienen un *associated value* que indica la causa detallada del error. Esto podría una cadena de caracteres o una tupla elementos con grandes elementos de información (por ejemplo, un código de error y una cadena que explica el código). El valor asociado generalmente se pasa como argumentos al constructor de la clase de excepción.

El código de usuario puede generar excepciones predefinidas. Esto puede ser usado para probar un gestor de excepciones o para notificar una condición de error «igual» a la situación en la cual el intérprete lance la misma excepción; pero tenga en cuenta que no hay nada que impida que el código del usuario produzca un error inapropiado.

Las clases de excepción predefinidas pueden ser usadas como subclases para definir otras nuevas; se recomienda a los programadores que deriven nuevas excepciones a partir de la clase `Exception` o de una de sus subclases, y no de `BaseException`. Hay disponible más información sobre la definición de excepciones en el Tutorial de Python en `tut-userexceptions`.

Al lanzar (o relanzar) una excepción en una cláusula `except` o `finally` cláusula `__context__` se establece automáticamente en la última excepción detectada; si no se controla la nueva excepción, el seguimiento que finalmente se muestra como resultado incluirá las excepciones de origen y la excepción final.

When raising a new exception (rather than using a bare `raise` to re-raise the exception currently being handled), the implicit exception context can be supplemented with an explicit cause by using `from` with `raise`:

```
raise new_exc from original_exc
```

The expression following `from` must be an exception or `None`. It will be set as `__cause__` on the raised exception. Setting `__cause__` also implicitly sets the `__suppress_context__` attribute to `True`, so that using `raise new_exc from None` effectively replaces the old exception with the new one for display purposes (e.g. converting `KeyError` to `AttributeError`), while leaving the old exception available in `__context__` for introspection when debugging.

La visualización por defecto de la traza de error muestra estas excepciones encadenadas además de la traza de la propia excepción. Siempre se muestra una excepción encadenada explícitamente en `__cause__` cuando está presente. Una excepción implícitamente encadenada en `__context__` solo se muestra si `__cause__` es `None` y `__suppress_context__` es falso.

En cualquier caso, la excepción en sí siempre se muestra después de cualquier excepción encadenada para que la línea final del seguimiento siempre muestre la última excepción que se generó.

5.1 Clases base

Las siguientes excepciones se utilizan principalmente como clases base para otras excepciones.

exception `BaseException`

La clase base para todas las excepciones predefinidas. No está pensada para ser heredada directamente por las clases definidas por el usuario (para eso, use *Exception*). Si se llama a `str()` en una instancia de esta clase, se retorna la representación de los argumentos de la instancia o la cadena vacía cuando no había argumentos.

args

La tupla de argumentos dada al constructor de excepción. Algunas excepciones predefinidas (como *OSError*) esperan un cierto número de argumentos y asignan un significado especial a los elementos de esta tupla, mientras que otras normalmente se llaman solo con una sola cadena que da un mensaje de error.

with_traceback (*tb*)

Este método establece *tb* como el nuevo `traceback` para la excepción y retorna el objeto de excepción. Normalmente se utiliza en código de control de excepciones como este:

```
try:
    ...
except SomeException:
    tb = sys.exc_info()[2]
    raise OtherException(...).with_traceback(tb)
```

exception `Exception`

Todas las excepciones predefinidas *non-system-exiting*, que no salen del sistema se derivan de esta clase. Todas las excepciones definidas por el usuario también deben derivarse de esta clase.

exception `ArithmeticError`

La clase base para las excepciones predefinidas que se generan para varios errores aritméticos: *OverflowError*, *ZeroDivisionError*, *FloatingPointError*.

exception `BufferError`

Se genera cuando `buffer` no se puede realizar una operación relacionada.

exception `LookupError`

La clase base para las excepciones que se generan cuando una clave o índice utilizado en un mapa o secuencia que no es válido: *IndexError*, *KeyError*. Esto se puede generar directamente por `codecs.lookup()`.

5.2 Excepciones específicas

Las siguientes excepciones son las excepciones que normalmente se generan.

exception `AssertionError`

Se genera cuando se produce un error en una instrucción `assert`.

exception `AttributeError`

Se genera cuando se produce un error en una referencia de atributo (ver *attribute-references*) o la asignación falla. (Cuando un objeto no admite referencias de atributos o asignaciones de atributos en absoluto, se genera *TypeError*.)

exception `EOFError`

Se genera cuando la función `input()` alcanza una condición de fin de archivo (EOF) sin leer ningún dato. (Note que el `io.IOBase.read()` y `io.IOBase.readline()` retornan una cadena vacía cuando llegan a EOF.)

exception FloatingPointError

No se usa actualmente.

exception GeneratorExit

Se genera cuando un *generator* o *coroutine* está cerrado; ver `generator.close()` y `coroutine.close()`. Hereda directamente de *BaseException* en lugar de *Exception* ya que técnicamente no es un error.

exception ImportError

Se genera cuando la instrucción `import` tiene problemas al intentar cargar un módulo. También se produce cuando la *from list* en `from ... import` tiene un nombre que no se puede encontrar.

Los atributos `name` y `path` solo se pueden establecer utilizando argumentos de palabra clave en el constructor. Cuando se establece, representan el nombre del módulo que se intentó importar y la ruta de acceso a cualquier archivo que desencadenó la excepción, respectivamente.

Distinto en la versión 3.3: Se han añadido los atributos `name` y `path`.

exception ModuleNotFoundError

Una subclase de *ImportError* que se genera mediante `import` cuando no se pudo encontrar un módulo. También se genera cuando `None` se encuentra en `sys.modules`.

Nuevo en la versión 3.6.

exception IndexError

Se genera cuando un subíndice de secuencia está fuera del rango. (Los índices de la rebanada son truncados silenciosamente para caer en el intervalo permitido; si un índice no es un entero, se genera *TypeError*.)

exception KeyError

Se genera cuando no se encuentra una clave de asignación (diccionario) en el conjunto de claves existentes (mapa).

exception KeyboardInterrupt

Se genera cuando el usuario pulsa la tecla de interrupción (normalmente `Control-C` o `Delete`). Durante la ejecución, se realiza una comprobación de interrupciones con regularidad. La excepción hereda de *BaseException* para no ser detectada de forma accidental por *Exception* y, por lo tanto, prevenir que el intérprete salga.

exception MemoryError

Se genera cuando una operación se queda sin memoria pero la situación aún puede ser recuperada (eliminando algunos objetos). El valor asociado es una cadena que indica que tipo de operación (interna) se quedó sin memoria. Tenga en cuenta que debido a la arquitectura de administración de memoria (función `malloc()` de C), el intérprete no siempre puede recuperarse completamente de esta situación; sin embargo, plantea una excepción para que se pueda imprimir un seguimiento de la pila, en caso de que un programa fuera de servicio fuera lo causa.

exception NameError

Se genera cuando no se encuentra un nombre local o global. Esto se aplica solo a nombres no calificados. El valor asociado es un mensaje de error que incluye el nombre que no se pudo encontrar.

exception NotImplementedError

Esta excepción se deriva de *RuntimeError*. En las clases base definidas por el usuario, los métodos abstractos deberían generar esta excepción cuando requieren clases derivadas para anular el método, o mientras se desarrolla la clase para indicar que la implementación real aún necesita ser agregada.

Nota: No debe usarse para indicar que un operador o método no debe ser admitido en absoluto – en ese caso, deje el operador / método sin definir o, si es una subclase, se establece en *None*.

Nota: *NotImplementedError* y *NotImplemented* no son intercambiables, a pesar de que tienen nombres y propósitos similares. Consulte *NotImplemented* para obtener detalles sobre cuándo usarlo.

exception OSError ([arg])

exception OSError (*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]]])

Esta excepción se produce cuando una función del sistema retorna un error relacionado con el sistema, que incluye fallas de E/S como `file not found` o `disk full` (no para tipos de argumentos ilegales u otros errores incidentales).

La segunda forma del constructor establece los atributos correspondientes, que se describen a continuación. Los atributos predeterminados son `None` si no se especifican. Para compatibilidad con versiones anteriores, si se pasan tres argumentos, el atributo `args` contiene solo una tupla de 2 de los dos primeros argumentos del constructor.

El constructor a menudo retorna una subclase de `OSError`, como se describe en *OS exceptions* a continuación. La subclase particular depende del valor final `errno`. Este comportamiento solo ocurre cuando se construye `OSError` directamente o mediante un alias, y no se hereda al derivar.

errno

Un código de error numérico de la variable C, `errno`.

winerror

En Windows, esto le proporciona el código de error nativo de Windows. El atributo `errno` es entonces una traducción aproximada, en términos POSIX, de ese código de error nativo.

En Windows, si el argumento del constructor `winerror` es un entero, el atributo `errno` se determina a partir del código de error de Windows y el argumento `errno` se ignora. En otras plataformas, el argumento `winerror` se ignora y el atributo `winerror` no existe.

strerror

El mensaje de error correspondiente, tal como lo proporciona el sistema operativo. Está formateado por las funciones de C, `perror()` en POSIX, y `FormatMessage()` en Windows.

filename

filename2

Para las excepciones que involucran una ruta del sistema de archivos (como `open()` o `os.unlink()`), `filename` es el nombre del archivo pasado a la función. Para las funciones que involucran dos rutas del sistema de archivos (como `os.rename()`), `filename2` corresponde al segundo nombre de archivo pasado a la función.

Distinto en la versión 3.3: `EnvironmentError`, `IOError`, `WindowsError`, `socket.error`, `select.error` y `mmap.error` se han fusionado en `OSError`, y el constructor puede retornar una subclase.

Distinto en la versión 3.4: El atributo `filename` es ahora el nombre del archivo original pasado a la función, en lugar del nombre codificado o decodificado de la codificación del sistema de archivos. Además, se agregó el argumento constructor `filename2` y el atributo.

exception OverflowError

Se genera cuando el resultado de una operación aritmética es demasiado grande para ser representado. Esto no puede ocurrir para los enteros (para lo cual es mejor elevar `MemoryError` que darse por vencido). Sin embargo, por razones históricas, `OverflowError` a veces se genera para enteros que están fuera del rango requerido. Debido a la falta de estandarización del manejo de excepciones de coma flotante en C, la mayoría de las operaciones de coma flotante no se verifican.

exception RecursionError

Esta excepción se deriva de `RuntimeError`. Se eleva cuando el intérprete detecta que se excede la profundidad máxima de recursión (ver `sys.getrecursionlimit()`).

Nuevo en la versión 3.5: Anteriormente, se planteó un simple `RuntimeError`.

exception ReferenceError

Esta excepción se produce cuando un proxy de referencia débil, creado por la función `weakref.proxy()`, se utiliza para acceder a un atributo del referente después de que se ha recolectado basura. Para obtener más información sobre referencias débiles, consulte el módulo `weakref`.

exception RuntimeError

Se genera cuando se detecta un error que no corresponde a ninguna de las otras categorías. El valor asociado es una cadena que indica exactamente qué salió mal.

exception StopIteration

Generado por la función incorporada `next()` y un *iterator*'s `__next__()` para indicar que no hay más elementos producidos por el iterador.

El objeto de excepción tiene un solo atributo `value`, que se proporciona como argumento al construir la excepción, y por defecto es `None`.

Cuando se retorna una función *generator* o *coroutine*, se genera una nueva instancia `StopIteration`, y el valor retornado por la función se utiliza como parámetro `value` para constructor de la excepción.

Si un generador lanza directa o indirectamente `StopIteration`, se convierte en `RuntimeError` (conservando `StopIteration` como la causa de la nueva excepción).

Distinto en la versión 3.3: Se agregó el atributo `*value*` y la capacidad de las funciones del generador de usarlo para retornar un valor.

Distinto en la versión 3.5: Introdujo la transformación `RuntimeError` a través de `from __future__ import generator_stop`, ver [PEP 479](#).

Distinto en la versión 3.7: Habilitar [PEP 479](#) para todo el código por defecto: a `StopIteration` generado en un generador se transforma en `RuntimeError`.

exception StopAsyncIteration

Se debe generar mediante `__anext__()` de un objeto *asynchronous iterator* para detener la iteración.

Nuevo en la versión 3.5.

exception SyntaxError

Se genera cuando el analizador (parser) encuentra un error de sintaxis. Esto puede ocurrir en una declaración `import`, en una llamada a las funciones predefinidas `exec()` o `eval()`, o al leer el script inicial o la entrada estándar (también de forma interactiva).

The `str()` of the exception instance returns only the error message.

filename

The name of the file the syntax error occurred in.

lineno

Which line number in the file the error occurred in. This is 1-indexed: the first line in the file has a `lineno` of 1.

offset

The column in the line where the error occurred. This is 1-indexed: the first character in the line has an `offset` of 1.

text

The source code text involved in the error.

exception IndentationError

Clase base para errores de sintaxis relacionados con sangría incorrecta. Esta es una subclase de `SyntaxError`.

exception TabError

Se genera cuando la sangría contiene un uso inconsistente de pestañas y espacios. Esta es una subclase de `IndentationError`.

exception SystemError

Se genera cuando el intérprete encuentra un error interno, pero la situación no parece tan grave como para abandonar toda esperanza. El valor asociado es una cadena que indica qué salió mal (a bajo nivel).

Debe informar esto al autor o responsable de su intérprete de Python. Asegúrese de informar la versión del intérprete de Python (`sys.version`; también se imprime al comienzo de una sesión interactiva de Python), el mensaje de error exacto (el valor asociado de la excepción) y, si es posible, la fuente del programa que activó el error.

exception SystemExit

Esta excepción es provocada por la función `sys.exit()`. Hereda de `BaseException` en lugar de

Exception para que no sea gestionada accidentalmente por el código que captura *Exception*. Esto permite que la excepción se propague correctamente y que el intérprete salga. Cuando no se maneja, el intérprete de Python sale; no se imprime el seguimiento de pila. El constructor acepta el mismo argumento opcional pasado a `sys.exit()`. Si el valor es un entero, especifica el estado de salida del sistema (se pasa a la función `exit()` de C); si es `None`, el estado de salida es cero; Si tiene otro tipo (como una cadena), se imprime el valor del objeto y el estado de salida es uno.

Una llamada `sys.exit()` se traduce en una excepción para que los gestores de limpieza (*finally* cláusulas de *try*) puedan ejecutarse, y para que un depurador pueda ejecutar un *script* sin correr el riesgo de perder el control. La función `os._exit()` se puede usar si es absolutamente necesario salir (por ejemplo, en el proceso secundario después de una llamada a `os.fork()`).

code

El estado de salida o mensaje de error que se pasa al constructor. (El valor predeterminado es `None`.)

exception TypeError

Se genera cuando una operación o función se aplica a un objeto de tipo inapropiado. El valor asociado es una cadena que proporciona detalles sobre la falta de coincidencia de tipos.

El código de usuario puede generar esta excepción para indicar que un intento de operación en un objeto no es compatible y no debe serlo. Si un objeto está destinado a soportar una operación dada pero aún no ha proporcionado una implementación, *NotImplementedError* es la excepción adecuada para generar.

Pasar argumentos del tipo incorrecto (por ejemplo, pasar a *list* cuando se espera un *int*) debería dar como resultado *TypeError*, pero pasar argumentos con el valor incorrecto (por ejemplo, un número fuera límites esperados) debería dar como resultado *ValueError*.

exception UnboundLocalError

Se genera cuando se hace referencia a una variable local en una función o método, pero no se ha vinculado ningún valor a esa variable. Esta es una subclase de *NameError*.

exception UnicodeError

Se genera cuando se produce un error de codificación o decodificación relacionado con Unicode. Es una subclase de *ValueError*.

UnicodeError tiene atributos que describen el error de codificación o decodificación. Por ejemplo, `err.object[err.start:err.end]` proporciona la entrada inválida particular en la que falló el códec.

encoding

El nombre de la codificación que provocó el error.

reason

Una cadena que describe el error de códec específico.

object

El objeto que el códec intentaba codificar o decodificar.

start

El primer índice de datos no válidos en *object*.

end

El índice después de los últimos datos no válidos en *object*.

exception UnicodeEncodeError

Se genera cuando se produce un error relacionado con Unicode durante la codificación. Es una subclase de *UnicodeError*.

exception UnicodeDecodeError

Se genera cuando se produce un error relacionado con Unicode durante la decodificación. Es una subclase de *UnicodeError*.

exception UnicodeTranslateError

Se genera cuando se produce un error relacionado con Unicode durante la codificación. Es una subclase de *UnicodeError*.

exception ValueError

Se genera cuando una operación o función recibe un argumento que tiene el tipo correcto pero un valor inapropiado, y la situación no se describe con una excepción más precisa como *IndexError*.

exception ZeroDivisionError

Se genera cuando el segundo argumento de una operación de división o módulo es cero. El valor asociado es una cadena que indica el tipo de operandos y la operación.

Las siguientes excepciones se mantienen por compatibilidad con versiones anteriores; a partir de Python 3.3, son alias de *OSError*.

exception EnvironmentError**exception IOError****exception WindowsError**

Solo disponible en Windows.

5.2.1 Excepciones del sistema operativo

Las siguientes excepciones son subclases de *OSError*, se generan según el código de error del sistema.

exception BlockingIOError

Se genera cuando una operación se bloquearía en un objeto (por ejemplo, un *socket*) configurado para una operación sin bloqueo. Corresponde a `errno` EAGAIN, EALREADY, EWOULDBLOCK y EINPROGRESS.

Además de los de *OSError*, *BlockingIOError* puede tener un atributo más:

characters_written

Un entero que contiene el número de caracteres escritos en la secuencia antes de que se bloquee. Este atributo está disponible cuando se utilizan las clases de E/S almacenadas en el módulo *io*.

exception ChildProcessError

Se genera cuando falla una operación en un proceso secundario. Corresponde `errno` ECHILD.

exception ConnectionError

Una clase base para problemas relacionados con la conexión.

Las subclases son *BrokenPipeError*, *ConnectionAbortedError*, *ConnectionRefusedError* y *ConnectionResetError*.

exception BrokenPipeError

Una subclase de *ConnectionError*, que se genera cuando se intenta escribir en una tubería mientras el otro extremo se ha cerrado, o cuando se intenta escribir en un *socket* que se ha cerrado por escritura. Corresponde a `errno` EPIPE y ESHUTDOWN.

exception ConnectionAbortedError

Una subclase de *ConnectionError*, que se genera cuando el par interrumpe un intento de conexión. Corresponde a `errno` ECONNABORTED.

exception ConnectionRefusedError

Una subclase de *ConnectionError*, que se genera cuando el par interrumpe un intento de conexión. Corresponde a `errno` ECONNABORTED.

exception ConnectionResetError

Una subclase de *ConnectionError*, que se genera cuando el par restablece una conexión. Corresponde a `errno` ECONNRESET.

exception FileExistsError

Se genera al intentar crear un archivo o directorio que ya existe. Corresponde a `errno` EEXIST.

exception FileNotFoundError

Se genera cuando se solicita un archivo o directorio pero no existe. Corresponde a `errno` ENOENT.

exception InterruptedError

Se genera cuando una llamada entrante interrumpe una llamada del sistema. Corresponde a `errno` EINTR.

Distinto en la versión 3.5: Python ahora vuelve a intentar las llamadas del sistema cuando una señal interrumpe un `syscall`, excepto si el gestor señala generar una excepción (ver [PEP 475](#) para la justificación), en lugar de lanzar `InterruptedError`.

exception IsADirectoryError

Se genera cuando se solicita una operación de archivo (como `os.remove()`) en un directorio. Corresponde a: `errno.EISDIR`.

exception NotADirectoryError

Se genera cuando se solicita una operación de directorio (como `os.listdir()`) en algo que no es un directorio. Corresponde a `errno.ENOTDIR`.

exception PermissionError

Se genera cuando se intenta ejecutar una operación sin los derechos de acceso adecuados, por ejemplo, permisos del sistema de archivos. Corresponde a `errno.EACCES` y `EPERM`.

exception ProcessLookupError

Generado cuando un proceso dado no existe. Corresponde a `errno.ESRCH`.

exception TimeoutError

Se genera cuando se agota el tiempo de espera de una función del sistema a nivel del sistema. Corresponde a `errno.ETIMEDOUT`.

Nuevo en la versión 3.3: Por lo anterior se agregaron las subclases `OSError`.

Ver también:

[PEP 3151](#) - Reelaborando el sistema operativo y la jerarquía de excepciones E/S

5.3 Advertencias

Las siguientes excepciones se utilizan como categorías de advertencia; consulte la documentación de [Categorías de advertencia](#) para obtener más detalles.

exception Warning

Clase base para categorías de advertencia.

exception UserWarning

Clase base para advertencias generadas por código de usuario.

exception DeprecationWarning

Clase base para advertencias sobre características obsoletas cuando esas advertencias están destinadas a otros desarrolladores de Python.

exception PendingDeprecationWarning

Clase base para advertencias sobre características que están obsoletas y que se espera que sean obsoletas en el futuro, pero que no lo son en este momento.

Esta clase rara vez se usa para emitir una advertencia sobre una posible desaprobación próxima es inusual, y `DeprecationWarning` se prefiere para las desaprobaciones ya activas.

exception SyntaxWarning

Clase base para advertencias sobre sintaxis dudosa.

exception RuntimeWarning

Clase base para advertencias sobre comportamiento dudoso en tiempo de ejecución.

exception FutureWarning

Clase base para advertencias sobre características en desuso cuando esas advertencias están destinadas a usuarios finales de aplicaciones escritas en Python.

exception ImportWarning

Clase base para advertencias sobre posibles errores en la importación de módulos.

exception UnicodeWarning

Clase base para advertencias relacionadas con Unicode.

exception BytesWarning

Clase base para advertencias relacionadas con *bytes* y *bytearray*.

exception ResourceWarning

Clase base para advertencias relacionadas con el uso de recursos. Ignorado por los filtros de advertencia pre-determinados.

Nuevo en la versión 3.2.

5.4 Jerarquía de excepción

La jerarquía de clases para las excepciones incorporadas es:

```

BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        |   +-- FloatingPointError
        |   +-- OverflowError
        |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
        |   +-- ModuleNotFoundError
    +-- LookupError
        |   +-- IndexError
        |   +-- KeyError
    +-- MemoryError
    +-- NameError
        |   +-- UnboundLocalError
    +-- OSError
        |   +-- BlockingIOError
        |   +-- ChildProcessError
        |   +-- ConnectionError
        |       |   +-- BrokenPipeError
        |       |   +-- ConnectionAbortedError
        |       |   +-- ConnectionRefusedError
        |       |   +-- ConnectionResetError
        |   +-- FileExistsError
        |   +-- FileNotFoundError
        |   +-- InterruptedError
        |   +-- IsADirectoryError
        |   +-- NotADirectoryError
        |   +-- PermissionError
        |   +-- ProcessLookupError
        |   +-- TimeoutError
    +-- ReferenceError
    +-- RuntimeError
        |   +-- NotImplementedError
        |   +-- RecursionError
    +-- SyntaxError
        |   +-- IndentationError

```

(continué en la próxima página)

(proviene de la página anterior)

```
|         +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|     +-- UnicodeError
|         +-- UnicodeDecodeError
|         +-- UnicodeEncodeError
|         +-- UnicodeTranslateError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning
```

Servicios de procesamiento de texto

Los módulos descritos en este capítulo proporcionan una amplia gama de operaciones de manipulación de cadenas de texto y otros servicios de procesamiento de texto.

El módulo *codecs* descrito en *Servicios de datos binarios* también es muy relevante para el procesamiento de texto. Además, consulta la documentación para el tipo *string* de Python en *Cadenas de caracteres — str*.

6.1 string — Operaciones comunes de cadena de caracteres

Source code: [Lib/string.py](#)

Ver también:

Cadenas de caracteres — str

Métodos de las cadenas de caracteres

6.1.1 Constantes de cadenas

Las constantes definidas en este módulo son:

`string.ascii_letters`

La concatenación de las constantes abajo descritas *ascii_lowercase* y *ascii_uppercase*. Este valor es independiente de la configuración regional.

`string.ascii_lowercase`

Las letras minúsculas 'abcdefghijklmnopqrstuvwxyz'. Este valor es independiente de la configuración regional y no cambiará.

`string.ascii_uppercase`

Las letras mayúsculas 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. Este valor es independiente de la configuración regional y no cambiará.

`string.digits`

La cadena '0123456789'.

`string.hexdigits`

La cadena '0123456789abcdefABCDEF'.

string.octdigits

La cadena de caracteres '01234567'.

string.punctuation

Cadena de caracteres ASCII que se consideran caracteres de puntuación en la configuración regional C: `!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~.`

string.printable

Cadena de caracteres ASCII que se consideran imprimibles. Esta es una combinación de *digits*, *ascii_letters*, *punctuation*, y *whitespace*.

string.whitespace

Una cadena cuyos caracteres ASCII se consideran todos espacios en blanco. Esto incluye los caracteres espacio, tabulador, salto de línea, retorno, salto de página y tabulador vertical.

6.1.2 Formato de cadena de caracteres personalizado

La clase cadena es una clase incorporada (*built-in*) que proporciona la capacidad de realizar sustituciones complejas de variables y formateo de valor a través del método *format()* descrito en **PEP 3101**. La clase *Formatter* del módulo *string* permite crear y personalizar sus propios comportamientos de formateo de cadena utilizando la misma implementación que el método integrado *format()*.

class string.Formatter

La clase *Formatter* tiene los siguientes métodos públicos:

format (*format_string*, /, **args*, ***kwargs*)

Método principal de la API. Recibe una cadena de formato y argumentos posicionales y de palabra clave arbitrarios. Es sólo un envoltorio que llama a *vformat()*.

Distinto en la versión 3.7: Un argumento de cadena de formato ahora es *solo posicional*.

vformat (*format_string*, *args*, *kwargs*)

Esta función realiza es la que realmente hace el trabajo de formateo. Se expone como una función independiente para los casos en los que desea pasar un diccionario predefinido de argumentos, en lugar de desempaquetar y volver a empaquetar el diccionario como argumentos individuales mediante la sintaxis **args* y ***kwargs*. *vformat()* hace el trabajo de dividir la cadena de formato en datos de caracteres y campos de reemplazo. Llama a los diversos métodos descritos a continuación.

Además de eso, la clase *Formatter* define varios métodos que se espera sean reemplazados por las subclases:

parse (*format_string*)

Itera sobre *format_string* y retorna un iterable de tuplas (*literal_text*, *field_name*, *format_spec*, *conversion*). Es usado por *vformat()* para dividir la cadena de caracteres en texto literal o en campos de reemplazo.

Los valores en la tupla representan conceptualmente un intervalo de texto literal seguido por un único campo de reemplazo. Si no hay ningún texto literal (lo cual puede darse si dos campos de reemplazo ocurren consecutivamente), *literal_text* será una cadena de caracteres de longitud cero. Si no hay ningún campo de reemplazo, los valores de *field_name*, *format_spec* y *conversion* serán *None*.

get_field (*field_name*, *args*, *kwargs*)

Dado un *field_name* retornado por *parse()* (véase arriba), el mismo es convertido a un objeto a formatear. retorna una tupla (*obj*, *used_key*). La versión por defecto toma cadenas de caracteres acorde a lo definido en **PEP 3101**, tales como «0[name]» o «label.title». *args* y *kwargs* se pasan al método *vformat()*. El valor retornado *used_key* tiene el mismo significado que el parámetro *key* para *get_value()*.

get_value (*key*, *args*, *kwargs*)

Recuperar un valor de campo determinado. El argumento *key* será un entero o una cadena de caracteres. Si es un entero, representa el índice del argumento posicional en *args*; si es una cadena, representa un argumento definido en *kwargs*.

El parámetro *args* se establece como lista de argumentos posicionales en *vformat()*, y el parámetro *kwargs* se establece como diccionario de argumentos de palabra clave.

Para nombres de campo compuesto, estas funciones son únicamente llamadas para el primer componente del campo. Los componentes que le siguen son tratados a través de operaciones normales de atributo e indexación.

Por ejemplo, la expresión de campo “0.name” haría que `get_value()` se llame con un argumento *key* igual a 0. El atributo “name” se buscará después del retorno de `get_value()` llamando a la función incorporada `getattr()`.

Si el índice o la palabra clave hace referencia a un elemento que no existe, se debe generar un `IndexError` o un `KeyError`.

check_unused_args (*used_args*, *args*, *kwargs*)

Implementa el chequeo de argumentos no utilizados si así se desea. Los argumentos de esta función son el conjunto de todas las claves de argumento a las que se hizo referencia en la cadena de formato (enteros para argumentos posicionales y cadenas de caracteres para argumentos con nombre) y una referencia a los *args* y *kwargs* que se pasaron a `vformat`. El conjunto de *args* no utilizados se puede calcular a partir de estos parámetros. Se asume que `check_unused_args()` genera una excepción si se produce un error en el chequeo.

format_field (*value*, *format_spec*)

`format_field()` simplemente llama a la función incorporada `format()`. El método se proporciona para que las subclases puedan sobrescribirlo.

convert_field (*value*, *conversion*)

Convierte el valor (retornado por `get_field()`) dado un tipo de conversión (tal como la tupla retornada por el método `parse()`). La versión por defecto entiende los tipos de conversión “s” (str), “r” (repr) y “a” (ascii).

6.1.3 Sintaxis de formateo de cadena

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). The syntax is related to that of formatted string literals, but it is less sophisticated and, in particular, does not support arbitrary expressions.

Las cadenas de caracteres de formato contienen «campos de reemplazo» rodeados de llaves {}. Todo lo que no está contenido entre llaves se considera texto literal, que se copia sin cambios en la salida. Si se necesita incluir un carácter de llave en el texto literal, se puede escapar duplicando: {{ and }}.

La gramática para un campo de reemplazo es la siguiente:

```
replacement_field ::= "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name         ::= arg_name ("." attribute_name | "[" element_index "]") *
arg_name           ::= [identifier | digit+]
attribute_name     ::= identifier
element_index      ::= digit+ | index_string
index_string       ::= <any source character except "]"> +
conversion         ::= "r" | "s" | "a"
format_spec        ::= <described in the next section>
```

En términos menos formales, el campo de reemplazo puede comenzar con un *field_name* (nombre de campo) que especifica el objeto cuyo valor se va a formatear e insertar en la salida en lugar del campo de reemplazo. El nombre de campo (*field_name*) va seguido opcionalmente de un campo *conversion* (conversión), que va precedido de un signo de exclamación '!', y un *format_spec*, que va precedido de dos puntos ':'. Estos especifican un formato no predeterminado para el valor de reemplazo.

Véase también la sección *Especificación de formato Mini-Lenguaje*.

El *field_name* (nombre de campo) comienza con un *arg_name* que es un número o una palabra clave. Si es un número, hace referencia a un argumento posicional y, si es una palabra clave, hace referencia a un argumento de palabra clave. Si los *arg_names* numéricos en una cadena de caracteres de formato son una secuencia como 0, 1, 2, ..., todos pueden ser omitidos (no sólo algunos) y los números 0, 1, 2, ... se insertarán automáticamente en ese orden.

Dado que *arg_name* no está delimitado por comillas, no es posible especificar claves de diccionario arbitrarias (por ejemplo, las cadenas '10' or ':-]') dentro de una cadena de caracteres de formato. El *arg_name* puede ir seguido de cualquier número de expresiones de índice o atributo. Una expresión con forma '.name' selecciona el atributo con nombre mediante *getattr()*, mientras que una expresión con forma '[index]' realiza una búsqueda de índice mediante *__getitem__()*.

Distinto en la versión 3.1: Los especificadores de argumentos posicionales pueden ser omitidos en *str.format()*, así '{0} {1}'.format(a, b) es equivalente a '{0} {1}'.format(a, b).

Distinto en la versión 3.4: Para la clase *Formatter*, los especificadores de argumento posicional pueden ser omitidos.

Algunos ejemplos simples de cadena de formato:

```
"First, thou shalt count to {0}" # References first positional argument
"Bring me a {}"                 # Implicitly references the first positional
↪argument
"From {} to {}".format(1, 2)    # 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'.
```

El campo *conversion* causa una coerción de tipo antes del formateo. Normalmente, el formateo es hecho el método *__format__()* del valor mismo. Sin embargo, en algunos es deseable forzar el tipo a ser formateado como una cadena de caracteres, sobrescribiendo su propia definición de formateo. Cuando se convierte el valor a una cadena de caracteres antes de llamar al método *__format__()*, la lógica normal de formateo es evitada.

Tres banderas de conversión son admitidas actualmente: '!s', que llama a *str()* con el valor; '!r', que llama a *repr()*; y '!a' que llama a *ascii()*.

Algunos ejemplos:

```
"Harold's a clever {0!s}"        # Calls str() on the argument first
"Bring out the holy {name!r}"    # Calls repr() on the argument first
"More {!a}"                     # Calls ascii() on the argument first
```

El campo *format_spec* contiene la especificación de cómo presentar el valor, incluyendo detalles como ancho de campo, alineación, relleno, precisión decimal, etc. Cada tipo de valor puede definir su propio «mini lenguaje de formateo» o interpretación de *format_spec*.

La mayoría de los tipos integrados admiten un formateo común de mini-idioma descrito en la siguiente sección.

Un campo *format_spec* también puede incluir campos de reemplazo anidados dentro de él. Estos campos de reemplazo anidados pueden contener un nombre de campo, una bandera de conversión y una especificación de formato, pero no se permite anidamiento más profundo. Los campos de reemplazo dentro de *format_spec* se sustituyen antes de que la cadena *format_spec* se interprete. Esto permite especificar dinámicamente el formato de un valor.

Para más ejemplos, véase la sección *Ejemplos de formateo*.

Especificación de formato Mini-Lenguaje

Las «especificaciones de formato» son usadas dentro de campos de reemplazo contenidos en una cadena de formateo para definir como se presentan los valores individuales (véase *Sintaxis de formateo de cadena* y f-strings). Los mismos pueden también ser pasados directamente a la función incorporada *format()*. Cada tipo formateable puede definir cómo interpretar la especificación de formato.

La mayoría de los tipos integrados implementan las siguientes opciones para especificaciones de formato, aunque algunas de las opciones de formateo sólo son posibles con los tipos numéricos.

Una convención general es que una especificación de formato vacía produce el mismo resultado que llamar a la función *str()* con el valor. Una especificación no vacía típicamente modifica el resultado.

La forma general de un *especificador estándar de formato* es:

```

format_spec      ::=  [[fill]align][sign][#][0][width][grouping_option][.precision][type]
fill             ::=  <any character>
align            ::=  "<" | ">" | "=" | "^"
sign             ::=  "+" | "-" | " "
width            ::=  digit+
grouping_option  ::=  "_" | ",",
precision        ::=  digit+
type             ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" |

```

Si se especifica un valor *align* válido, puede ir precedido por un carácter *fill*, que puede ser cualquier carácter y cuyo valor predeterminado es un espacio si se omite. No es posible utilizar una llave literal (`>>` or `<<`) como el carácter *fill* en un formato literal de cadena o cuando se utiliza el método `str.format()`. Sin embargo, es posible insertar una llave con un campo de reemplazo anidado. Esta limitación no afecta a la función `format()`.

El significado de las distintas opciones de alineación es el siguiente:

Op- ción	Significado
'<'	Fuerza el campo a ser alineado a la izquierda dentro del espacio disponible (éste es el comportamiento por defecto para la mayoría de los objetos).
'>'	Fuerza el campo a ser alineado a la derecha dentro del espacio disponible (éste es el comportamiento por defecto para números).
'='	Fuerza el relleno a ser colocarlo después del signo (si existe) pero antes de los dígitos. Esto se utiliza para imprimir campos con el formato <code>"+000000120"</code> . Esta opción de alineación solo es válida para tipos numéricos. Se convierte en el valor predeterminado cuando <code>"0"</code> precede inmediatamente al ancho del campo.
'^'	Fuerza el centrado del campo dentro del espacio disponible.

Notar que, a menos que se defina un ancho de campo mínimo, el ancho del campo siempre tendrá el mismo tamaño que los datos para rellenarlo, de modo que la opción de alineación no tiene ningún significado en este caso.

La opción *sign* (signo) sólo es válida para los tipos numéricos y puede ser una de las siguientes:

Op- ción	Significado
'+'	indica que el signo debe ser usado tanto para los números positivos como negativos.
'-'	indica que el signo debe ser usado sólo para números negativos (éste es el comportamiento por defecto).
espa- cio	indica que el espacio inicial debe ser usado para números positivos y el signo menos para números negativos.

The `'#'` option causes the «alternate form» to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float and complex types. For integers, when binary, octal, or hexadecimal output is used, this option adds the prefix respective `'0b'`, `'0o'`, or `'0x'` to the output value. For float and complex the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for `'g'` and `'G'` conversions, trailing zeros are not removed from the result.

La opción `','` señala el uso de una coma como separador de miles. En cambio, para un separador consciente de localización (*local aware*), usar el tipo de presentación de enteros `'n'`.

Distinto en la versión 3.1: Se agregó la opción `','` (véase también [PEP 378](#)).

La opción `'_'` señala el uso del guión bajo como separador de miles para tipos de presentación de punto flotante y para tipos de presentación de enteros `'d'`. Para los tipos de presentación de enteros `'b'`, `'o'`, `'x'` y `'X'`, el guión bajo se insertará cada 4 dígitos. Para otros tipos de presentación, especificar esta opción es un error.

Distinto en la versión 3.6: Se agregó la opción `'_'` (véase también [PEP 515](#)).

width es un entero decimal que define el ancho total de campo mínimo, incluyendo prefijos, separadores y otros caracteres de formato. Si no se especifica, el ancho de campo será determinado por el contenido.

Cuando no se proporciona ninguna alineación explícita, si el campo *width* es precedido por un carácter cero ('0'), se habilita el relleno cero con reconocimiento de signos para los tipos numéricos. Esto equivale a un carácter *fill* de '0' con un tipo de *alignment* de '='.

El argumento *precision* (precisión) es un número decimal que indica cuántos dígitos se deben mostrar después del punto decimal para un valor de punto flotante formateado con 'f' y 'F', o bien antes y después del punto decimal para un valor de punto flotante formateado con 'g' or 'G'. Para los tipos no numéricos, el campo indica el tamaño máximo del campo, es decir, cuántos caracteres se utilizarán del contenido del campo. El argumento *precision* no es admitido para los valores enteros.

Finalmente, *type* (el tipo) determina como presentar los datos.

Los tipos de presentación cadena disponibles son:

Tipo	Significado
's'	Formato de cadena de caracteres. Éste es el tipo por defecto y puede ser omitido.
None	Lo mismo que 's'.

Los tipos disponibles para la presentación de enteros son:

Ti-po	Significado
'b'	Formato binario. retorna el número en base 2.
'c'	Carácter. Convierte el entero en el carácter unicode correspondiente antes de imprimirlo.
'd'	Decimal entero. retorna el número en base 10.
'o'	Formato octal. retorna el número en base 8.
'x'	Formato hexadecimal. retorna el número en base 16, utilizando letras minúsculas para los dígitos superiores a 9.
'X'	Formato hexadecimal. retorna el número en base 16, utilizando letras mayúsculas para los dígitos superiores a 9.
'n'	Número. Es lo mismo que 'd', excepto que usa la configuración regional actual para insertar el número apropiado de caracteres separadores.
No-ne	Lo mismo que 'd'.

Además de los tipos de presentación arriba expuestos, los enteros se pueden formatear con los tipos de presentación de punto flotante enumerados a continuación (excepto 'n' y 'None'). Al hacerlo, *float()* se utiliza para convertir el entero en un número de punto flotante antes de ser formateado.

The available presentation types for *float* and *Decimal* values are:

Ti-po	Significado
'e'	Scientific notation. For a given precision <i>p</i> , formats the number in scientific notation with the letter “e” separating the coefficient from the exponent. The coefficient has one digit before and <i>p</i> digits after the decimal point, for a total of <i>p</i> + 1 significant digits. With no precision given, uses a precision of 6 digits after the decimal point for <i>float</i> , and shows all coefficient digits for <i>Decimal</i> . If no digits follow the decimal point, the decimal point is also removed unless the # option is used.
'E'	Scientific notation. Same as 'e' except it uses an upper case “E” as the separator character.
'f'	Fixed-point notation. For a given precision <i>p</i> , formats the number as a decimal number with exactly <i>p</i> digits following the decimal point. With no precision given, uses a precision of 6 digits after the decimal point for <i>float</i> , and uses a precision large enough to show all coefficient digits for <i>Decimal</i> . If no digits follow the decimal point, the decimal point is also removed unless the # option is used.
'F'	Notación de punto fijo. Igual que 'f', pero convierte (nulos) nan a NAN e inf a INF.
'g'	General format. For a given precision <i>p</i> >= 1, this rounds the number to <i>p</i> significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. A precision of 0 is treated as equivalent to a precision of 1. Las reglas precisas son las siguientes: supongamos que el resultado formateado con el tipo de presentación 'e' y la precisión <i>p</i> -1 tiene exponente <i>exp</i> . Entonces, si <i>m</i> <= <i>exp</i> < <i>p</i> , donde <i>m</i> es -4 para <i>floats</i> y -6 para <i>Decimals</i> , el número se formatea con el tipo de presentación 'f' y la precisión <i>p</i> -1- <i>exp</i> . De lo contrario, el número se formatea con el tipo de presentación 'e' y precisión <i>p</i> -1. En ambos casos, los ceros finales insignificantes se eliminan del significado, y el punto decimal también se elimina si no hay dígitos restantes que lo sigan, a menos que se utilice la opción '#'. With no precision given, uses a precision of 6 significant digits for <i>float</i> . For <i>Decimal</i> , the coefficient of the result is formed from the coefficient digits of the value; scientific notation is used for values smaller than 1e-6 in absolute value and values where the place value of the least significant digit is larger than 1, and fixed-point notation is used otherwise. Infinito positivo y negativo, cero positivo y negativo, y nulos (<i>nans</i>) son respectivamente formateados como <i>inf</i> , <i>-inf</i> , 0, <i>-0</i> y <i>nan</i> , independientemente de la precisión.
'G'	Formato general. Igual que 'g' excepto que cambia a 'E' si el número se vuelve muy grande. Las representaciones de infinito y NaN también se convierten a mayúsculas.
'n'	Número. Es lo mismo que 'g', excepto que usa la configuración local para insertar los caracteres separadores de número apropiados.
'%'	Porcentaje. Multiplica el número por 100 y lo muestra en formato fijo ('f') seguido del signo porcentaje.
No-ne	For <i>float</i> this is the same as 'g', except that when fixed-point notation is used to format the result, it always includes at least one digit past the decimal point. The precision used is as large as needed to represent the given value faithfully. For <i>Decimal</i> , this is the same as either 'g' or 'G' depending on the value of <i>context.capitals</i> for the current decimal context. The overall effect is to match the output of <i>str()</i> as altered by the other format modifiers.

Ejemplos de formateo

Esta sección contiene ejemplos de la sintaxis `str.format()` y comparaciones con el antiguo método de formateo usando %.

En la mayoría de los casos, la sintaxis es similar al antiguo formato %, con la adición de {} y con : utilizado en lugar de %. Por ejemplo, '%03.2f' puede ser traducido como '{:03.2f}'.

La nueva sintaxis de formato también soporta opciones diferentes y nuevas que se muestran en los ejemplos siguientes.

Accediendo argumentos por posición:


```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{} {}, {}'.format('a', 'b', 'c') # 3.1+ 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'
```

Accediendo argumentos por nombre:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-
↳115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Accediendo los atributos de los argumentos:

```
>>> 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:
...     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)'
```

Accediendo ítems de los argumentos:

```
>>> coord = (3, 5)
>>> 'X: {0[0]}; Y: {0[1]}'.format(coord)
'X: 3; Y: 5'
```

Reemplazar %s y %r:

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

Alinear el texto y especificar el ancho:

```
>>> '{:<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*****'
```

Reemplazar %+f, %-f, y % f y especificar el signo:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14) # show it always
'+3.140000; -3.140000'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> '{: 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'
```

Reemplazando %x y %o y convirtiendo el valor a diferentes 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:
>>> "int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(42)
'int: 42; hex: 0x2a; oct: 0o52; bin: 0b101010'
```

Usando la coma como separador de los miles:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

Expresar un porcentaje:

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

Uso del formateo específico de tipo:

```
>>> 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'
```

Anidando argumentos y ejemplos más complejos:

```
>>> 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), end=' ')
...     print()
...
5      5      5      101
6      6      6      110
7      7      7      111
8      8      10     1000
9      9      11     1001
```

(continúe en la próxima página)

(proviene de la página anterior)

10	A	12	1010
11	B	13	1011

6.1.4 Cadenas de plantillas

Las cadenas de caracteres de plantilla proporcionan sustituciones de cadenas más sencillas como se describe en [PEP 292](#). Un caso de uso principal para las cadenas de plantilla es la internacionalización (`i18n`) ya que en ese contexto, la sintaxis y la funcionalidad más sencillas hacen la traducción más fácil que otras instalaciones de formato de cadena integradas en Python. Como ejemplo de una biblioteca creada sobre cadenas de caracteres de plantilla para `i18n`, véase el paquete `flufl.i18n`.

Las cadenas de caracteres de plantilla admiten sustituciones basadas en `$` de acuerdo a las siguientes reglas:

- `$$` es un escape. Es reemplazado con un único `$`.
- `$identifier` (identificador) nombra un comodín de sustitución que coincide con una clave de asignación de `"identifier"` (identificador). De forma predeterminada, `"identifier"` está restringido a cualquier cadena alfanumérica ASCII (insensible a mayúsculas/minúsculas e incluyendo los guiones bajos) que comience con un guión bajo o una letra ASCII. El primer carácter no identificador después del carácter `$` termina esta especificación de comodín.
- `${*identifier*}` (identificador) es equivalente a `$identifier`. Es requerido cuando caracteres identificadores válidos siguen al comodín pero no son parte de él, por ejemplo `"${noun}ification"`.

Cualquier otra aparición de `$` en la cadena de caracteres resultará en una excepción `ValueError`.

El módulo `string` provee una clase `Template` que implementa esas reglas. Los métodos de `Template` son:

class `string.Template` (*template*)

El constructor sólo lleva un argumento, la cadena plantilla.

substitute (*mapping*=`{}`, */*, ****kws**)

Realiza la sustitución de plantilla y retorna una nueva cadena de caracteres. *mapping* (mapeo) es un objeto tipo diccionario con claves (*keys*) que coinciden con los *placeholders* (comodines) de la plantilla. Como alternativa, es posible pasar argumentos de palabra clave cuyas palabras clave son los *placeholders* (comodines). Cuando *mapping* y *kws* son dados y hay elementos duplicados, los *placeholders* (comodines) de *kws* tienen prioridad.

safe_substitute (*mapping*=`{}`, */*, ****kws**)

Igual que `substitute()`, excepto que si faltan comodines de *mapping* y *kws*, en lugar de generar una excepción `KeyError`, el comodín original aparecerá en la cadena de caracteres resultante intacta. Además, a diferencia de `substitute()`, cualquier otra aparición de `$` simplemente retornará `$` en lugar de generar `ValueError`.

Mientras que otras excepciones aún pueden ocurrir, este método es llamado «seguro» (*safe*) porque siempre intenta retornar una cadena de caracteres que pueda ser usada en lugar de levantar una excepción. Desde otro punto de vista, el método `safe_substitute()` es en realidad para nada seguro, dado que ignorará plantillas defectuosas con delimitadores colgados, llaves sin cerrar, o comodines que no son identificadores válidos en Python.

Las instancias de `Template` también proporcionan un atributo de datos públicos:

template

Éste es el objeto que se le pasa como argumento *template* al constructor. En general, no debería ser modificado, pero el acceso de sólo lectura (*read-only*) no es impuesto.

Aquí un ejemplo de cómo usar una plantilla (*Template*):

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
```

(continué en la próxima página)

(proviene de la página anterior)

```

>>> 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'

```

Uso avanzado: es posible derivar subclases de *Template* para personalizar la sintaxis de *placeholder*, caracteres de delimitación, o bien la expresión regular entera usada para procesar cadenas de plantillas. Para ello, es posible sobrescribir los siguientes atributos de clase:

- *delimiter*: es la cadena de caracteres literal que describe al delimitador que introduce un comodín. El valor predeterminado es `$`. Notar que esto *no debe* ser una expresión regular, ya que la implementación llamará a `re.escape()` en esta cadena de caracteres según sea necesario. Tener en cuenta además, que se no puede cambiar el delimitador después haber creado la clase (es decir, se debe establecer un delimitador diferente en el espacio de nombres de la subclase).
- *idpattern* – Esta es la expresión regular que describe el patrón para comodines fuera de las llaves. El valor predeterminado es la expresión regular `(?a: [_a-z] [_a-z0-9] *)`. Si se proporciona este argumento y *braceidpattern* es `None` este patrón también se aplicará a los comodines entre llaves.

Nota: Dado que el valor predeterminado de *flags* es `re.IGNORECASE`, el patrón `[a-z]` puede coincidir con algunos caracteres que no son ASCII. Por ello se utiliza aquí la bandera local `a`.

Distinto en la versión 3.7: *braceidpattern* puede ser usado para definir patrones separados, usados dentro y fuera de los corchetes.

- *braceidpattern* – Es como *idpattern* pero describe el patrón para comodines entre llaves. El valor por defecto es `None`, lo que significa volver a *idpattern* (es decir, el mismo patrón se utiliza tanto dentro como fuera de las llaves). Si se proporciona, esto le permite definir diferentes patrones para comodines dentro y fuera de las llaves.

Nuevo en la versión 3.7.

- *flags* – Banderas de expresión regular que se aplicarán al compilar la expresión regular utilizada para reconocer sustituciones. El valor por defecto es `re.IGNORECASE`. Téngase en cuenta que `re.VERBOSE` siempre se agregará a las banderas, por lo que *idpattern* (s) personalizado(s) debe(n) seguir las convenciones de expresiones regulares detalladas.

Nuevo en la versión 3.2.

Como alternativa, se puede proporcionar el patrón de expresión regular completo, reemplazando así el atributo de clase *pattern*. Si eso ocurre, el valor debe ser un objeto de expresión regular con cuatro grupos de captura con nombre. Los grupos de captura corresponden a las reglas indicadas anteriormente, junto con la regla de marcador de posición no válida:

- *escaped* – Este grupo coincide con la secuencia de escape en el patrón predeterminado, por ejemplo, `$$`.
- *named* – Este grupo coincide con el nombre comodín fuera de las llaves. No debe incluir el delimitador del grupo de captura.
- *braced* – Este grupo coincide con el nombre del comodín adjunto; no debe incluir ni el delimitador ni las llaves en el grupo de captura.
- *invalid* – Este grupo se empareja con cualquier otro patrón de delimitación (usualmente un único carácter) y debe ser lo último en aparecer en la expresión regular.

6.1.5 Funciones de ayuda

`string.capitalize(s, sep=None)`

Separa el argumento en dos palabras utilizando el método `str.split()`, convierte a mayúsculas vía `str.capitalize()` y une las palabras en mayúscula con el método `str.join()`. Si el segundo argumento opcional `sep` está ausente o es `None`, los espacios en blanco se reemplazarán con un único espacio y los espacios en blanco iniciales y finales se eliminarán; caso contrario, `sep` se usa para separar y unir las palabras.

6.2 re — Operaciones con expresiones regulares

Código fuente: [Lib/re.py](#)

Este módulo proporciona operaciones de coincidencia de expresiones regulares similares a las encontradas en Perl.

Tanto los patrones como las cadenas de texto a buscar pueden ser cadenas de Unicode (`str`) así como cadenas de 8 bits (`bytes`). Sin embargo, las cadenas Unicode y las cadenas de 8 bits no se pueden mezclar: es decir, no se puede hacer coincidir una cadena Unicode con un patrón de bytes o viceversa; del mismo modo, al pedir una sustitución, la cadena de sustitución debe ser del mismo tipo que el patrón y la cadena de búsqueda.

Las expresiones regulares usan el carácter de barra inversa (`'\'`) para indicar formas especiales o para permitir el uso de caracteres especiales sin invocar su significado especial. Esto choca con el uso de Python de este carácter para el mismo propósito con los literales de cadena; por ejemplo, para hacer coincidir una barra inversa literal, se podría escribir `'\\'` como patrón, porque la expresión regular debe ser `\\`, y cada barra inversa debe ser expresada como `\\` dentro de un literal de cadena regular de Python. También, notar que cualquier secuencia de escape inválida mientras se use la barra inversa de Python en los literales de cadena ahora genera un `DeprecationWarning` y en el futuro esto se convertirá en un `SyntaxError`. Este comportamiento ocurrirá incluso si es una secuencia de escape válida para una expresión regular.

La solución es usar la notación de cadena *raw* de Python para los patrones de expresiones regulares; las barras inversas no se manejan de ninguna manera especial en un literal de cadena prefijado con `'r'`. Así que `r"\n"` es una cadena de dos caracteres que contiene `'\'` y `'n'`, mientras que `"\n"` es una cadena de un carácter que contiene una nueva línea. Normalmente los patrones se expresan en código Python usando esta notación de cadena *raw*.

Es importante señalar que la mayoría de las operaciones de expresiones regulares están disponibles como funciones y métodos a nivel de módulo en *expresiones regulares compiladas* (expresiones regulares compiladas). Las funciones son atajos que no requieren de compilar un objeto regex primero, aunque pasan por alto algunos parámetros de ajuste.

Ver también:

El módulo de terceros `regex`, cuenta con una API compatible con el módulo de la biblioteca estándar `re`, el cual ofrece una funcionalidad adicional y un soporte Unicode más completo.

6.2.1 Sintaxis de expresiones regulares

Una expresión regular (o RE, por sus siglas en inglés) especifica un conjunto de cadenas que coinciden con ella; las funciones de este módulo permiten comprobar si una determinada cadena coincide con una expresión regular dada (o si una expresión regular dada coincide con una determinada cadena, que se reduce a lo mismo).

Las expresiones regulares pueden ser concatenadas para formar nuevas expresiones regulares; si *A* y *B* son ambas expresiones regulares, entonces *AB* es también una expresión regular. En general, si una cadena *p* coincide con *A* y otra cadena *q* coincide con *B*, la cadena *porque* coincidirá con *AB*. Esto se mantiene a menos que *A* o *B* contengan operaciones de baja precedencia; condiciones límite entre *A* y *B*; o tengan referencias de grupo numeradas. Así, las expresiones complejas pueden construirse fácilmente a partir de expresiones primitivas más simples como las que se describen aquí. Para detalles de la teoría e implementación de las expresiones regulares, consulte el libro de Friedl [Frie09], o casi cualquier libro de texto sobre la construcción de compiladores.

A continuación se explica brevemente el formato de las expresiones regulares. Para más información y una presentación más amena, consultar la *regex-howto*.

Las expresiones regulares pueden contener tanto caracteres especiales como ordinarios. La mayoría de los caracteres ordinarios, como 'A', 'a', o '0' son las expresiones regulares más sencillas; simplemente se ajustan a sí mismas. Se pueden concatenar caracteres ordinarios, así que `last` coincide con la cadena `'last'`. (En el resto de esta sección, se escribirán los RE en este estilo especial, normalmente sin comillas, y las cadenas que deban coincidir 'entre comillas simples'.)

Algunos caracteres, como '|' o '(', son especiales. Los caracteres especiales representan clases de caracteres ordinarios, o afectan a la forma en que se interpretan las expresiones regulares que los rodean.

Los delimitadores de repetición (*, +, ?, {m, n}, etc.) no pueden ser anidados directamente. Esto evita la ambigüedad con el sufijo modificador no *greedy* (codiciosos) ?, y con otros modificadores en otras implementaciones. Para aplicar una segunda repetición a una repetición interna, se pueden usar paréntesis. Por ejemplo, la expresión `(?:a{6})*` coincide con cualquier múltiplo de seis caracteres 'a'.

Los caracteres especiales son:

- . (Punto.) En el modo predeterminado, esto coincide con cualquier carácter excepto con una nueva línea. Si se ha especificado el indicador `DOTALL`, esto coincide con cualquier carácter que incluya una nueva línea.
- ^ (Circunflejo.) Coincide con el comienzo de la cadena, y en modo `MULTILINE` también coincide inmediatamente después de cada nueva línea.
- \$ Coincide con el final de la cadena o justo antes de la nueva línea al final de la cadena, y en modo `MULTILINE` también coincide antes de una nueva línea. `foo` coincide con “foo” y “foobar”, mientras que la expresión regular `foo$` sólo coincide con “foo”. Más interesante aún, al buscar `foo.$` en `'foo1\nfoo2\n'` coincide con “foo2” normalmente, pero solo “foo1” en `MULTILINE`; si busca un solo \$ en `'foo\n'` encontrará dos coincidencias (vacías): una justo antes de una nueva línea, y otra al final de la cadena.
- * Hace que el RE resultante coincida con 0 o más repeticiones del RE precedente, tantas repeticiones como sean posibles. `ab*` coincidirá con “a”, “ab” o “a” seguido de cualquier número de “b”.
- + Hace que la RE resultante coincida con 1 o más repeticiones de la RE precedente. `ab+` coincidirá con “a” seguido de cualquier número distinto de cero de “b”; no coincidirá solo con “a”.
- ? Hace que la RE resultante coincida con 0 o 1 repeticiones de la RE precedente. `ab?` coincidirá con “a” o “ab”.
- *?, +?, ?? Los delimitadores «*», «+» y «?» son todos *greedy* (codiciosos); coinciden con la mayor cantidad de texto posible. A veces este comportamiento no es deseado; si el RE `<.*>` se utiliza para coincidir con `<a>b<c>`, coincidirá con toda la cadena, y no sólo con `<a>`. Añadiendo ? después del delimitador hace que se realice la coincidencia de manera *non-greedy* o *minimal*; coincidirá la *mínima* cantidad de caracteres como sea posible. Usando el RE `<.*?>` sólo coincidirá con `<a>`.
- {m} Especifica que exactamente *m* copias de la RE anterior deben coincidir; menos coincidencias hacen que la RE entera no coincida. Por ejemplo, `a{6}` coincidirá exactamente con seis caracteres 'a', pero no con cinco.
- {m, n} Hace que el RE resultante coincida de *m* a *n* repeticiones del RE precedente, tratando de coincidir con el mayor número de repeticiones posible. Por ejemplo, `a{3, 5}` coincidirá de 3 a 5 caracteres 'a'. Omitiendo *m* se especifica un límite inferior de cero, y omitiendo *n* se especifica un límite superior infinito. Por ejemplo, `a{4, }b` coincidirá con `'aaaab'` o mil caracteres 'a' seguidos de una 'b', pero no `'aaab'`. La coma no puede ser omitida o el modificador se confundiría con la forma descrita anteriormente.
- {m, n}? Hace que el RE resultante coincida de *m* a *n* repeticiones del RE precedente, tratando de coincidir con el *mínimo* de repeticiones posible. Esta es la versión *non-greedy* (no codiciosa) del delimitador anterior. Por ejemplo, en la cadena de 6 caracteres `'aaaaaa'`, `a{3, 5}` coincidirá con 5 caracteres 'a', mientras que `a{3, 5}?` solo coincidirá con 3 caracteres.
- \ O bien se escapan a los caracteres especiales (lo que le permite hacer coincidir caracteres como '*', '?', y así sucesivamente), o se señala una secuencia especial; las secuencias especiales se explican más adelante.

Si no se utiliza una cadena *raw* para expresar el patrón, recuerde que Python también utiliza la barra inversa como secuencia de escape en los literales de la cadena; si el analizador sintáctico de Python no reconoce la secuencia de escape, la barra inversa y el carácter subsiguiente se incluyen en la cadena resultante. Sin embargo, si Python quisiera reconocer la secuencia resultante, la barra inversa debería repetirse dos veces. Esto es complicado y difícil de entender, por lo que se recomienda encarecidamente utilizar cadenas *raw* para todas las expresiones salvo las más simples.

[] Se utiliza para indicar un conjunto de caracteres. En un conjunto:

- Los caracteres pueden ser listados individualmente, ej. `[amk]` coincidirá con `'a'`, `'m'`, o `'k'`.
- Los rangos de caracteres se pueden indicar mediante dos caracteres y separándolos con un `-`. Por ejemplo, `[a-z]` coincidirá con cualquier letra ASCII en minúscula, `[0-5]` `[0-9]` coincidirá con todos los números de dos dígitos desde el 00 hasta el 59, y `[0-9A-Fa-f]` coincidirá con cualquier dígito hexadecimal. Si se escapa `-` (por ejemplo, `[a\ -z]`) o si se coloca como el primer o el último carácter (por ejemplo, `[-a]` o `[a-]`), coincidirá con un literal `'-'`.
- Los caracteres especiales pierden su significado especial dentro de los sets. Por ejemplo, `[(+*)]` coincidirá con cualquiera de los caracteres literales `'('`, `'+'`, `'*'`, o `')'`.
- Las clases de caracteres como `\w` o `\S` (definidas más adelante) también se aceptan dentro de un conjunto, aunque los caracteres que coinciden dependen de si el modo `ASCII` o `LOCALE` está activo.
- Los caracteres que no están dentro de un rango pueden ser coincidentes con *complementing* el conjunto. Si el primer carácter del conjunto es `^`, todos los caracteres que *no* están en el conjunto coincidirán. Por ejemplo, `[^5]` coincidirá con cualquier carácter excepto con `'5'`, y `[^^]` coincidirá con cualquier carácter excepto con `^`. `^` no tiene un significado especial si no es el primer carácter del conjunto.
- Para coincidir con un `']'` literal dentro de un set, se debe preceder con una barra inversa, o colocarlo al principio del set. Por ejemplo, tanto `[()] [\] [\{\}]` como `[] () [\{\}]` coincidirá con los paréntesis, corchetes y llaves.
- El soporte de conjuntos anidados y operaciones de conjuntos como en [Unicode Technical Standard #18](#) podría ser añadido en el futuro. Esto cambiaría la sintaxis, así que por el momento se planteará un *FutureWarning* en casos ambiguos para facilitar este cambio. Ello incluye conjuntos que empiecen con un literal `'['` o que contengan secuencias de caracteres literales `'-'`, `'&&'`, `'~~'` y `'||'`. Para evitar una advertencia, utilizar el código de escape con una barra inversa.

Distinto en la versión 3.7: *FutureWarning* se genera si un conjunto de caracteres contiene construcciones que cambiarán semánticamente en el futuro.

| `A|B`, donde `A` y `B` pueden ser RE arbitrarias, crea una expresión regular que coincidirá con `A` or `B`. Un número arbitrario de RE puede ser separado por `'|'` de esta manera. Esto puede también ser usado dentro de grupos (ver más adelante). Cuando la cadena de destino es procesada, los RE separados por `'|'` son probados de izquierda a derecha. Cuando un patrón coincide completamente, esa rama es aceptada. Esto significa que una vez que `A` coincida, `B` no se comprobará más, incluso si se produce una coincidencia general más larga. En otras palabras, el operador de `'|'` nunca es codicioso. Para emparejar un literal `'|'`, se usa `\|`, o se envuelve dentro de una clase de caracteres, como en `[|]`.

(...) Coincide con cualquier expresión regular que esté dentro de los paréntesis, e indica el comienzo y el final de un grupo; el contenido de un grupo puede ser recuperado después de que se haya realizado una coincidencia, y puede coincidir más adelante en la cadena con la secuencia especial `\number`, que se describe más adelante. Para hacer coincidir los literales `'('` o `')'`, se usa `\(` o `\)`, o se envuelve dentro de una clase de caracteres: `[()]`, `[()]`.

(?...) Esta es una notación de extensión (un `'?'` después de un `'('` no tiene ningún otro significado). El primer carácter después de `'?'` determina el significado y la sintaxis de la construcción. Las extensiones normalmente no crean un nuevo grupo; `(?P<name>...)` es la única excepción a esta regla. A continuación se muestran las extensiones actualmente soportadas.

(**?aiLmsux**) (Una o más letras del conjunto `'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'`.) El grupo coincide con la cadena vacía; las letras ponen los indicadores correspondientes: `re.A` (coincidencia sólo en ASCII), `re.I` (ignorar mayúsculas o minúsculas), `re.L` (dependiente de la configuración regional), `re.M` (multilínea), `re.S` (el punto coincide con todo), `re.U` (coincidencia con Unicode), y `re.X` (modo *verbose*), para toda la expresión regular. (Los indicadores se describen en [Contenidos del módulo](#).) Esto es útil si se desea incluir los indicadores como parte de la expresión regular, en lugar de pasar un argumento *flag* (indicador) a la función `re.compile()`. Los indicadores deben ser usados primero en la cadena de expresión.

(?:...) Una versión no capturable de los paréntesis regulares. Hace coincidir cualquier expresión regular que esté dentro de los paréntesis, pero la subcadena coincidente con el grupo *no puede* ser recuperada después de realizar una coincidencia o referenciada más adelante en el patrón.

(?aiLmsux-imsx:...) (Cero o más letras del conjunto 'a', 'i', 'L', 'm', 's', 'u', 'x', opcionalmente seguido de '-' seguido de una o más letras de 'i', 'm', 's', 'x'.) Las letras ponen o quitan los indicadores correspondientes: *re.A* (coincidencia sólo en ASCII), *re.I* (ignorar mayúsculas o minúsculas), *re.L* (dependiente de la configuración regional), *re.M* (multilínea), *re.S* (el punto coincide con todo), *re.U* (coincidencia con Unicode), y *re.X* (modo *verbose*) para la parte de la expresión. (Los indicadores se describen en *Contenidos del módulo*.)

Las letras 'a', 'L' y 'u' se excluyen mutuamente cuando se usan como indicadores en línea, así que no pueden combinarse o ser seguidos por '-'. En cambio, cuando uno de ellos aparece en un grupo dentro de la línea, anula el modo de coincidencia en el grupo que lo rodea. En los patrones Unicode, (?a:...) cambia al modo de concordancia sólo en ASCII, y (?u:...) cambia al modo de concordancia Unicode (por defecto). En el patrón de bytes (?L:...) se cambia a una correspondencia en función de la configuración regional, y (?a:...) se cambia a una correspondencia sólo en ASCII (predeterminado). Esta anulación sólo tiene efecto para el grupo de línea restringida, y el modo de coincidencia original se restaura fuera del grupo.

Nuevo en la versión 3.6.

Distinto en la versión 3.7: Las letras 'a', 'L' y 'u' también pueden ser usadas en un grupo.

(?P<name>...) Similar a los paréntesis regulares, pero la subcadena coincidente con el grupo es accesible a través del nombre simbólico del grupo, *name*. Los nombres de grupo deben ser identificadores válidos de Python, y cada nombre de grupo debe ser definido sólo una vez dentro de una expresión regular. Un grupo simbólico es también un grupo numerado, del mismo modo que si el grupo no tuviera nombre.

Los grupos con nombre pueden ser referenciados en tres contextos. Si el patrón es (?P<quote>["']) .*? (?P=quote) (es decir, hacer coincidir una cadena citada con comillas simples o dobles):

Contexto de la referencia al grupo <i>quote</i> (cita)	Formas de hacer referencia
en el mismo patrón en sí mismo	<ul style="list-style-type: none"> (?P=quote) (como se muestra) \1
cuando se procesa el objeto de la coincidencia <i>m</i>	<ul style="list-style-type: none"> m.group('quote') m.end('quote') (etc.)
en una cadena pasada al argumento <i>repl</i> de <i>re.sub()</i>	<ul style="list-style-type: none"> \g<quote> \g<1> \1

(?P=name) Una referencia inversa a un grupo nombrado; coincide con cualquier texto correspondido por el grupo anterior llamado *name*.

(?#...) Un comentario; el contenido de los paréntesis es simplemente ignorado.

(?=...) Coincide si ... coincide con el siguiente patrón, pero no procesa nada de la cadena. Esto se llama una *lookahead assertion* (aserción de búsqueda anticipada). Por ejemplo, Isaac (?=Asimov) coincidirá con 'Isaac ' sólo si va seguido de 'Asimov'.

(?!...) Coincide si ... no coincide con el siguiente. Esta es una *negative lookahead assertion* (aserción negativa de búsqueda anticipada). Por ejemplo, Isaac (?!Asimov) coincidirá con 'Isaac ' sólo si *no* es seguido por 'Asimov'.

(?<=...) Coincide si la posición actual en la cadena es precedida por una coincidencia para ... que termina en la posición actual. Esto se llama una *positive lookbehind assertion* (aserciones positivas de búsqueda tardía). (?<=abc)def encontrará una coincidencia en 'abcdef', ya que la búsqueda tardía hará una copia de seguridad de 3 caracteres y comprobará si el patrón contenido coincide. El patrón contenido sólo debe coincidir con cadenas de alguna longitud fija, lo que significa que abc o a|b están permitidas, pero a* y a{3,4} no lo están. Hay que tener en cuenta que los patrones que empiezan con aserciones positivas de búsqueda tardía no coincidirán con el principio de la cadena que se está buscando; lo más probable es que se quiera usar la función *search()* en lugar de la función *match()*:

```
>>> import re
>>> m = re.search('(?!<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

Este ejemplo busca una palabra seguida de un guión:

```
>>> m = re.search(r'(?<=)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

Distinto en la versión 3.5: Se añadió soporte a las referencias de grupo de longitud fija.

(?!...) Coincide si la posición actual en la cadena no está precedida por una coincidencia de «...». Esto se llama una *negative lookbehind assertion* (Aserciones negativas de búsqueda tardía). Similar a las aserciones positivas de búsqueda tardía, el patrón contenido sólo debe coincidir con cadenas de alguna longitud fija. Los patrones que empiezan con aserciones negativas pueden coincidir al principio de la cadena que se busca.

(?(id/name)yes-pattern|no-pattern) Tratará de coincidir con el `yes-pattern` (con patrón) si el grupo con un *id* o *nombre* existe, y con el `no-pattern` (sin patrón) si no existe. El `no-pattern` es opcional y puede ser omitido. Por ejemplo, `(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|)$` es un patrón de coincidencia de correo electrónico deficiente, ya que coincidirá con `'<user@host.com>'` así como con `'user@host.com'`, pero no con `'<user@host.com'` ni con `'user@host.com>'`.

Las secuencias especiales consisten en `'\'` y un carácter de la lista que aparece más adelante. Si el carácter ordinario no es un dígito ASCII o una letra ASCII, entonces el RE resultante coincidirá con el segundo carácter. Por ejemplo, `\$` coincide con el carácter `'$'`.

\number Coincide con el contenido del grupo del mismo número. Los grupos se numeran empezando por el 1. Por ejemplo, `(.+)\1` coincide con `'el el'` o `'55 55'`, pero no con `'el el'` (notar el espacio después del grupo). Esta secuencia especial sólo puede ser usada para hacer coincidir uno de los primeros 99 grupos. Si el primer dígito del *número* es 0, o el *número* tiene 3 dígitos octales, no se interpretará como una coincidencia de grupo, sino como el carácter con valor octal *número*. Dentro de los `'[` y `']'` de una clase de caracteres, todos los escapes numéricos son tratados como caracteres.

\A Coincide sólo al principio de la cadena.

\b Coincide con la cadena vacía, pero sólo al principio o al final de una palabra. Una palabra se define como una secuencia de caracteres de palabras. Notar que formalmente, `\b` se define como el límite entre un carácter `\w` y un carácter `\W` (o viceversa), o entre `\w` y el principio/fin de la cadena. Esto significa que `r'\bfoo\b'` coincide con `'foo'`, `'foo.'`, `'(foo)'`, `'bar foo baz'` pero no `'foobar'` o `'foo3'`.

Por defecto, los alfanuméricos Unicode son los que se usan en los patrones Unicode, pero esto se puede cambiar usando el indicador *ASCII*. Los límites de las palabras están determinados por la configuración regional actual si se usa el indicador *LOCALE*. Dentro de un rango de caracteres, `\b` representa el carácter de retroceso (*backspace*), para compatibilidad con los literales de las cadenas de Python.

\B Coincide con la cadena vacía, pero sólo cuando *no* está al principio o al final de una palabra. Esto significa que `r'py\B'` coincide con `'python'`, `'py3'`, `'py2'`, pero no con `'py'`, `'py.'` o `'py!'`. `\B` es justo lo opuesto a `\b`, por lo que los caracteres de las palabras en los patrones de Unicode son alfanuméricos o el subrayado, aunque esto puede ser cambiado usando el indicador *ASCII*. Los límites de las palabras están determinados por la configuración regional actual si se usa el indicador *LOCALE*.

\d

Para los patrones de Unicode (str): Coincide con cualquier dígito decimal de Unicode (es decir, cualquier carácter de la categoría de caracteres de Unicode [Nd]). Esto incluye a `[0-9]`, y también muchos otros caracteres de dígitos. Si se usa el indicador *ASCII*, sólo coincide con `[0-9]`.

Para patrones de 8 bits (bytes): Coincide con cualquier dígito decimal; esto equivale a `[0-9]`.

\D Coincide con cualquier carácter que no sea un dígito decimal. Esto es lo opuesto a `\d`. Si se usa el indicador *ASCII* esto se convierte en el equivalente a `[^0-9]`.

\s

Para los patrones de Unicode (str): Coincide con los caracteres de los espacios en blanco de Unicode (que incluye [`\t\n\r\f\v`], y también muchos otros caracteres, por ejemplo los espacios duros exigidos por las reglas tipográficas en muchos idiomas). Si se usa el indicador `ASCII`, sólo [`\t\n\r\f\v`] coincide.

Para patrones de 8 bits (bytes): Coincide con los caracteres considerados como espacios en blanco en el conjunto de caracteres ASCII, lo que equivale a [`\t\n\r\f\v`].

\S Coincide con cualquier carácter que no sea un carácter de espacio en blanco. Esto es lo opuesto a `\s`. Si se usa el indicador `ASCII` se convierte en el equivalente a [`tmrfv`].

\w

Para los patrones de Unicode (str): Coincide con los caracteres de palabras de Unicode; esto incluye la mayoría de los caracteres que pueden formar parte de una palabra en cualquier idioma, así como los números y el guión bajo. Si se usa el indicador `ASCII`, sólo coincide con [`a-zA-Z0-9_`].

Para patrones de 8 bits (bytes): Coincide con los caracteres considerados alfanuméricos en el conjunto de caracteres ASCII; esto equivale a [`a-zA-Z0-9_`]. Si se usa el indicador `LOCALE`, coincide con los caracteres considerados alfanuméricos en la configuración regional actual y el guión bajo.

\W Coincide con cualquier carácter que no sea un carácter de una palabra. Esto es lo opuesto a `\w`. Si se usa el indicador `ASCII` esto se convierte en el equivalente a [`^a-zA-Z0-9_`]. Si se usa el indicador `LOCALE`, coincide con los caracteres que no son ni alfanuméricos en la configuración regional actual ni con el guión bajo.

\Z Coincide sólo el final de la cadena.

La mayoría de los escapes estándar soportados por los literales de la cadena de Python también son aceptados por el analizador de expresiones regulares:

<code>\a</code>	<code>\b</code>	<code>\f</code>	<code>\n</code>
<code>\N</code>	<code>\r</code>	<code>\t</code>	<code>\u</code>
<code>\U</code>	<code>\v</code>	<code>\x</code>	<code>\\</code>

(Notar que `\b` se usa para representar los límites de las palabras, y significa «retroceso» (*backspace*) sólo dentro de las clases de caracteres.)

Las secuencias de escape `'\u'`, `'\U'` y `'\N'` sólo se reconocen en los patrones Unicode. En los patrones de bytes son errores. Los escapes desconocidos de las letras ASCII se reservan para su uso posterior y se consideran errores.

Los escapes octales se incluyen en una forma limitada. Si el primer dígito es un 0, o si hay tres dígitos octales, se considera un escape octal. De lo contrario, es una referencia de grupo. En cuanto a los literales de cadena, los escapes octales siempre tienen como máximo tres dígitos de longitud.

Distinto en la versión 3.3: Se han añadido las secuencias de escape `'\u'` y `'\U'`.

Distinto en la versión 3.6: Los escapes desconocidos que consisten en `'\'` y una letra ASCII ahora son errores.

Distinto en la versión 3.8: Se añadió la secuencia de escape `'\N{name}'`. Como en los literales de cadena, se expande al carácter Unicode nombrado (por ej. `'\N{EM DASH}'`).

6.2.2 Contenidos del módulo

El módulo define varias funciones, constantes y una excepción. Algunas de las funciones son versiones simplificadas de los métodos completos de las expresiones regulares compiladas. La mayoría de las aplicaciones no triviales utilizan siempre la forma compilada.

Distinto en la versión 3.6: Ahora las constantes de indicadores son instancias de `RegexFlag`, que es una subclase de `enum.IntFlag`.

`re.compile(pattern, flags=0)`

Compila un patrón de expresión regular en un *objeto de expresión regular*, que puede ser usado para las coincidencias usando `match()`, `search()` y otros métodos, descritos más adelante.

El comportamiento de la expresión puede modificarse especificando un valor de *indicadores*. Los valores pueden ser cualquiera de las siguientes variables, combinadas usando el operador OR (el operador `|`).

La secuencia

```
prog = re.compile(pattern)
result = prog.match(string)
```

es equivalente a

```
result = re.match(pattern, string)
```

pero usando `re.compile()` y guardando el objeto resultante de la expresión regular para su reutilización es más eficiente cuando la expresión será usada varias veces en un solo programa.

Nota: Las versiones compiladas de los patrones más recientes pasaron a `re.compile()` y las funciones de coincidencia a nivel de módulo están en caché, así que los programas que usan sólo unas pocas expresiones regulares a la vez no tienen que preocuparse de compilar expresiones regulares.

re.A

re.ASCII

Hace que `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` y `\S` realicen una coincidencia ASCII en lugar de una concordancia Unicode. Esto sólo tiene sentido para los patrones de Unicode, y se ignora para los patrones de bytes. Corresponde al indicador en línea `(?a)`.

Notar que para la compatibilidad con versiones anteriores, el indicador `re.U` todavía existe (así como su sinónimo `re.UNICODE` y su contraparte incrustada `(?u)`), pero estos son redundantes en Python 3 ya que las coincidencias son Unicode por defecto para las cadenas (y no se permite la coincidencia Unicode para los bytes).

re.DEBUG

Muestra información de depuración (*debug*) sobre la expresión compilada. No hay un indicador en línea que corresponda.

re.I

re.IGNORECASE

Realiza una coincidencia insensible a las mayúsculas y minúsculas; expresiones como `[A-Z]` también coincidirán con las minúsculas. La coincidencia completa de Unicode (como Û coincidencia ü) también funciona a menos que el indicador `re.ASCII` se utilice para desactivar las coincidencias que no sean ASCII. La configuración regional vigente no cambia el efecto de este indicador a menos que también se use el indicador `re.LOCALE`. Corresponde al indicador en línea `(?i)`.

Note that when the Unicode patterns `[a-z]` or `[A-Z]` are used in combination with the `IGNORECASE` flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters: “İ” (U+0130, Latin capital letter I with dot above), “ı” (U+0131, Latin small letter dotless i), “ſ” (U+017F, Latin small letter long s) and “K” (U+212A, Kelvin sign). If the `ASCII` flag is used, only letters “a” to “z” and “A” to “Z” are matched.

re.L

re.LOCALE

Hace que las coincidencias `\w`, `\W`, `\b`, `\B` y las coincidencias insensibles a mayúsculas y minúsculas dependan de la configuración regional vigente. Este indicador sólo puede ser usado con patrones de bytes. Se desaconseja su uso ya que el mecanismo de configuración regional no es fiable, sólo maneja una «cultura» a la vez, y sólo funciona con localizaciones de 8 bits. La coincidencia Unicode ya está activada por defecto en Python 3 para los patrones Unicode (str), y es capaz de manejar diferentes localizaciones/idiomas. Corresponde al indicador en línea `(?L)`.

Distinto en la versión 3.6: `re.LOCALE` sólo se puede usar con patrones de bytes y no es compatible con `re.ASCII`.

Distinto en la versión 3.7: Los objetos expresión regular compilados con el indicador `re.LOCALE` ya no dependen del lugar en el momento de la compilación. Sólo la configuración regional durante la coincidencia afecta al resultado obtenido.

re.M

re.MULTILINE

Cuando se especifica, el patrón de caracteres `'^'` coincide al principio de la cadena y al principio de cada línea (inmediatamente después de cada nueva línea); y el patrón de caracteres `'$'` coincide al final de la cadena y al final de cada línea (inmediatamente antes de cada nueva línea). Por defecto, `'^'` coincide sólo al principio de la cadena, y `'$'` sólo al final de la cadena e inmediatamente antes de la nueva línea (si la hay) al final de la cadena. Corresponde al indicador en línea `(?m)`.

re.S**re.DOTALL**

Hace que el carácter especial `'.'` coincida con cualquier carácter, incluyendo una nueva línea. Sin este indicador, `'.'` coincidirá con cualquier cosa, *excepto* con una nueva línea. Corresponde al indicador en línea `(?s)`.

re.X**re.VERBOSE**

Este indicador permite escribir expresiones regulares que se ven mejor y son más legibles al facilitar la separación visual de las secciones lógicas del patrón y añadir comentarios. Los espacios en blanco dentro del patrón se ignoran, excepto cuando están en una clase de caracteres, o cuando están precedidos por una barra inversa no escapada, o dentro de fichas como `*?`, `(?: o (?P<...>`. Cuando una línea contiene un `#` que no está en una clase de caracteres y no está precedida por una barra inversa no escapada, se ignoran todos los caracteres desde el más a la izquierda (como `#`) hasta el final de la línea.

Esto significa que los dos siguientes objetos expresión regular que coinciden con un número decimal son funcionalmente iguales:

```
a = re.compile(r"""\d + # the integral part
                \.    # the decimal point
                \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

Corresponde al indicador en línea `(?x)`.

re.search (*pattern*, *string*, *flags=0*)

Examina a través de la *string* («cadena») buscando el primer lugar donde el *pattern* («patrón») de la expresión regular produce una coincidencia, y retorna un *objeto match* correspondiente. Retorna `None` si ninguna posición en la cadena coincide con el patrón; notar que esto es diferente a encontrar una coincidencia de longitud cero en algún punto de la cadena.

re.match (*pattern*, *string*, *flags=0*)

Si cero o más caracteres al principio de la *string* («cadena») coinciden con el *pattern* («patrón») de la expresión regular, retorna un *objeto match* correspondiente. Retorna `None` si la cadena no coincide con el patrón; notar que esto es diferente de una coincidencia de longitud cero.

Notar que incluso en el modo *MULTILINE*, *re.match()* sólo coincidirá al principio de la cadena y no al principio de cada línea.

Si se quiere localizar una coincidencia en cualquier lugar de la *string* («cadena»), se utiliza *search()* en su lugar (ver también *search()* vs. *match()*).

re.fullmatch (*pattern*, *string*, *flags=0*)

Si toda la *string* («cadena») coincide con el *pattern* («patrón») de la expresión regular, retorna un correspondiente *objeto match*. Retorna `None` si la cadena no coincide con el patrón; notar que esto es diferente de una coincidencia de longitud cero.

Nuevo en la versión 3.4.

re.split (*pattern*, *string*, *maxsplit=0*, *flags=0*)

Divide la *string* («cadena») por el número de ocurrencias del *pattern* («patrón»). Si se utilizan paréntesis de captura en *pattern*, entonces el texto de todos los grupos en el patrón también se retornan como parte de la lista resultante. Si *maxsplit* (máxima divisibilidad) es distinta de cero, como mucho se producen *maxsplit* divisiones, y el resto de la cadena se retorna como elemento final de la lista.

```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', '', ' ', 'words', '', ' ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

Si hay grupos de captura en el separador y coincide al principio de la cadena, el resultado comenzará con una cadena vacía. Lo mismo ocurre con el final de la cadena:

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', '', ' ', 'words', '...', '']
```

De esa manera, los componentes de los separadores se encuentran siempre en los mismos índices relativos dentro de la lista de resultados.

Las coincidencias vacías para el patrón dividen la cadena sólo cuando no están adyacentes a una coincidencia vacía anterior.

```
>>> re.split(r'\b', 'Words, words, words.')
['', 'Words', '', ' ', 'words', '', ' ', 'words', '.', '']
>>> re.split(r'\W*', '...words...')
['', '', 'w', 'o', 'r', 'd', 's', '', '']
>>> re.split(r'(\W*)', '...words...')
['', '...', '', '', 'w', '', 'o', '', 'r', '', 'd', '', 's', '...', '', '', '']
```

Distinto en la versión 3.1: Se añadió el argumento de los indicadores opcionales.

Distinto en la versión 3.7: Se añadió el soporte de la división en un patrón que podría coincidir con una cadena vacía.

re.findall (*pattern*, *string*, *flags=0*)

Retorna todas las coincidencias no superpuestas del *pattern* («patrón») en la *string* («cadena»), como una lista de cadenas. La cadena es examinada de izquierda a derecha, y las coincidencias son retornadas en el orden en que fueron encontradas. Si uno o más grupos están presentes en el patrón, retorna una lista de grupos; esta será una lista de tuplas si el patrón tiene más de un grupo. Las coincidencias vacías se incluyen en el resultado.

Distinto en la versión 3.7: Las coincidencias no vacías ahora pueden empezar justo después de una coincidencia vacía anterior.

re.finditer (*pattern*, *string*, *flags=0*)

Retorna un *iterator* que produce *objetos de coincidencia* sobre todas las coincidencias no superpuestas para *pattern* («patrón») de RE en la *string* («cadena»). La *string* es examinada de izquierda a derecha, y las coincidencias son retornadas en el orden en que se encuentran. Las coincidencias vacías se incluyen en el resultado.

Distinto en la versión 3.7: Las coincidencias no vacías ahora pueden empezar justo después de una coincidencia vacía anterior.

re.sub (*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Retorna la cadena obtenida reemplazando las ocurrencias no superpuestas del *pattern* («patrón») en la *string* («cadena») por el reemplazo de *repl*. Si el patrón no se encuentra, se retorna *string* sin cambios. *repl* puede ser una cadena o una función; si es una cadena, cualquier barra inversa escapada en ella es procesada. Es decir, `\n` se convierte en un carácter de una sola línea nueva, `\r` se convierte en un retorno de carro, y así sucesivamente. Los escapes desconocidos de las letras ASCII se reservan para un uso futuro y se tratan como errores. Otros escapes desconocidos como `&` no se utilizan. Las referencias inversas, como `\6`, se reemplazan por la subcadena que corresponde al grupo 6 del patrón. Por ejemplo:

```
>>> 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{'
```

Si *repl* es una función, se llama para cada ocurrencia no superpuesta de *pattern*. La función toma un solo argumento *objeto match*, y retorna la cadena de sustitución. Por ejemplo:

```
>>> 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'
```

El patrón puede ser una cadena o un *objeto patrón*.

El argumento opcional *count* («recuento») es el número máximo de ocurrencias de patrones a ser reemplazados; *count* debe ser un número entero no negativo. Si se omite o es cero, todas las ocurrencias serán reemplazadas. Las coincidencias vacías del patrón se reemplazan sólo cuando no están adyacentes a una coincidencia vacía anterior, así que `sub('x*', '-', 'abxd')` retorna `'-a-b--d-'`.

En los argumentos *repl* de tipo cadena, además de los escapes de caracteres y las referencias inversas descritas anteriormente, `\g<name>` usará la subcadena coincidente con el grupo llamado *name*, como se define en la sintaxis `(?P<name>...)`. `\g<number>` utiliza el número de grupo correspondiente; `\g<2>` es por lo tanto equivalente a `\2`, pero no es ambiguo en un reemplazo como sucede con `\g<2>0`. `\20` se interpretaría como una referencia al grupo 20, no como una referencia al grupo 2 seguido del carácter literal `'0'`. La referencia inversa `\g<0>` sustituye en toda la subcadena coincidente con la RE.

Distinto en la versión 3.1: Se añadió el argumento de los indicadores opcionales.

Distinto en la versión 3.5: Los grupos no coincidentes son reemplazados por una cadena vacía.

Distinto en la versión 3.6: Los escapes desconocidos en el *pattern* que consisten en `'\'` y una letra ASCII ahora son errores.

Distinto en la versión 3.7: Los escapes desconocidos en *repl* que consisten en `'\'` y una letra ASCII ahora son errores.

Distinto en la versión 3.7: Las coincidencias vacías para el patrón se reemplazan cuando están adyacentes a una coincidencia anterior no vacía.

re.subn (*pattern*, *repl*, *string*, *count=0*, *flags=0*)
Realiza la misma operación que `sub()`, pero retorna una tupla (*new_string*, *number_of_subs_made*).

Distinto en la versión 3.1: Se añadió el argumento de los indicadores opcionales.

Distinto en la versión 3.5: Los grupos no coincidentes son reemplazados por una cadena vacía.

re.escape (*pattern*)
Caracteres de escape especiales en *pattern* («patrón»). Esto es útil si quieres hacer coincidir una cadena literal arbitraria que puede tener metacaracteres de expresión regular en ella. Por ejemplo:

```
>>> print(re.escape('http://www.python.org'))
http://www\.python\.org

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789!\#$%&'*\+|-\.^_`|~:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print(''.join(map(re.escape, sorted(operators, reverse=True))))
/|\-|\+|\*|\*|\*
```

Esta función no debe usarse para la cadena de reemplazo en `sub()` y `subn()`, sólo deben escaparse las barras inversas. Por ejemplo:

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

Distinto en la versión 3.3: El carácter de '_' ya no se escapa.

Distinto en la versión 3.7: Sólo se escapan los caracteres que pueden tener un significado especial en una expresión regular. Como resultado, '!', '"', '%', "'", ',', '/', ':', ';', '<', '=', '>', '@' y "`" ya no se escapan.

re.purge()

Despeja la caché de expresión regular.

exception re.error (*msg, pattern=None, pos=None*)

Excepción señalada cuando una cadena enviada a una de las funciones descritas aquí no es una expresión regular válida (por ejemplo, podría contener paréntesis no coincidentes) o cuando se produce algún otro error durante la compilación o la coincidencia. Nunca es un error si una cadena no contiene ninguna coincidencia para un patrón. La instancia de error tiene los siguientes atributos adicionales:

msg

El mensaje de error sin formato.

pattern

El patrón de expresión regular.

pos

El índice en *pattern* («patrón») donde la compilación falló (puede ser *None*).

lineno

La línea correspondiente a *pos* (puede ser *None*).

colno

La columna correspondiente a *pos* (puede ser *None*).

Distinto en la versión 3.5: Se añadieron atributos adicionales.

6.2.3 Objetos expresión regular

Los objetos expresión regular compilados soportan los siguientes métodos y atributos:

Pattern.search (*string*, [*pos*, *endpos*])

Escanea a través de la *string* («cadena») buscando la primera ubicación donde esta expresión regular produce una coincidencia, y retorna un *objeto match* correspondiente. Retorna *None* si ninguna posición en la cadena coincide con el patrón; notar que esto es diferente a encontrar una coincidencia de longitud cero en algún punto de la cadena.

El segundo parámetro opcional *pos* proporciona un índice en la cadena donde la búsqueda debe comenzar; por defecto es 0. Esto no es completamente equivalente a dividir la cadena; el patrón de carácter '^' coincide en el inicio real de la cadena y en las posiciones justo después de una nueva línea, pero no necesariamente en el índice donde la búsqueda va a comenzar.

El parámetro opcional *endpos* limita hasta dónde se buscará la cadena; será como si la cadena fuera de *endpos* caracteres de largo, por lo que sólo se buscará una coincidencia entre los caracteres de *pos* a *endpos* - 1. Si *endpos* es menor que *pos*, no se encontrará ninguna coincidencia; de lo contrario, si *rx* es un objeto de expresión regular compilado, *rx.search(string, 0, 50)* es equivalente a *rx.search(string[:50], 0)*.

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")           # Match at index 0
<re.Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)        # No match; search doesn't include the "d"
```

`Pattern.match(string[, pos[, endpos]])`

Si cero o más caracteres en el *beginning* («comienzo») de la *string* («cadena») coinciden con esta expresión regular, retorna un *objeto match* correspondiente. Retorna `None` si la cadena no coincide con el patrón; notar que esto es diferente de una coincidencia de longitud cero.

Los parámetros opcionales *pos* y *endpos* tienen el mismo significado que para el método `search()`.

```
>>> 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".
<re.Match object; span=(1, 2), match='o'>
```

Si se quiere encontrar una coincidencia en cualquier lugar de *string*, utilizar `search()` en su lugar (ver también `search()` vs. `match()`).

`Pattern.fullmatch(string[, pos[, endpos]])`

Si toda la *string* («cadena») coincide con esta expresión regular, retorna un *objeto match* correspondiente. Retorna `None` si la cadena no coincide con el patrón; notar que esto es diferente de una coincidencia de longitud cero.

Los parámetros opcionales *pos* y *endpos* tienen el mismo significado que para el método `search()`.

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")        # No match as "o" is not at the start of "dog"
→ ".
>>> pattern.fullmatch("ogre")       # No match as not the full string matches.
>>> pattern.fullmatch("doggie", 1, 3) # Matches within given limits.
<re.Match object; span=(1, 3), match='og'>
```

Nuevo en la versión 3.4.

`Pattern.split(string, maxsplit=0)`

Idéntico a la función `split()`, usando el patrón compilado.

`Pattern.findall(string[, pos[, endpos]])`

Similar a la función `findall()`, usando el patrón compilado, pero también acepta parámetros opcionales *pos* y *endpos* que limitan la región de búsqueda como para `search()`.

`Pattern.finditer(string[, pos[, endpos]])`

Similar a la función `finditer()`, usando el patrón compilado, pero también acepta parámetros opcionales *pos* y *endpos* que limitan la región de búsqueda como para `search()`.

`Pattern.sub(repl, string, count=0)`

Idéntico a la función `sub()`, usando el patrón compilado.

`Pattern.subn(repl, string, count=0)`

Idéntico a la función `subn()`, usando el patrón compilado.

`Pattern.flags`

Los indicadores regex de coincidencia. Esta es una combinación de los indicadores dados a `compile()`, cualquier indicador (`?. . .`) en línea en el patrón, y los indicadores implícitos como `UNICODE` si el patrón es una cadena de Unicode.

`Pattern.groups`

El número de grupos de captura en el patrón.

`Pattern.groupindex`

Un diccionario que mapea cualquier nombre de grupo simbólico definido por `(?P<id>)` para agrupar números. El diccionario está vacío si no se utilizaron grupos simbólicos en el patrón.

`Pattern.pattern`

La cadena de patrones a partir de la cual el objeto de patrón fue compilado.

Distinto en la versión 3.7: Se añadió el soporte de `copy.copy()` y `copy.deepcopy()`. Los objetos expresión regular compilados se consideran atómicos.

6.2.4 Objetos de coincidencia

Los objetos de coincidencia siempre tienen un valor booleano de `True` («Verdadero»). Ya que `match()` y `search()` retornan `None` cuando no hay coincidencia. Se puede probar si hubo una coincidencia con una simple declaración `if`:

```
match = re.search(pattern, string)
if match:
    process(match)
```

Los objetos de coincidencia admiten los siguientes métodos y atributos:

`Match.expand(template)`

Retorna la cadena obtenida al hacer la sustitución de la barra inversa en la cadena de la plantilla *template*, como se hace con el método `sub()`. Escapes como `\n` son convertidos a los caracteres apropiados, y las referencias inversas numéricas (`\1`, `\2`) y las referencias inversas con nombre (`\g<1>`, `\g<name>`) son reemplazadas por el contenido del grupo correspondiente.

Distinto en la versión 3.5: Los grupos no coincidentes son reemplazados por una cadena vacía.

`Match.group([group1, ...])`

Retorna uno o más subgrupos de la coincidencia. Si hay un solo argumento, el resultado es una sola cadena; si hay múltiples argumentos, el resultado es una tupla con un elemento por argumento. Sin argumentos, *group1* tiene un valor por defecto de cero (se retorna la coincidencia completa). Si un argumento *groupN* es cero, el valor de retorno correspondiente es toda la cadena coincidente; si está en el rango inclusivo `[1..99]`, es la cadena coincidente con el grupo correspondiente entre paréntesis. Si un número de grupo es negativo o mayor que el número de grupos definidos en el patrón, se produce una excepción `IndexError`. Si un grupo está contenido en una parte del patrón que no coincidió, el resultado correspondiente es `None`. Si un grupo está contenido en una parte del patrón que coincidió varias veces, se retorna la última coincidencia.

```
>>> 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')
```

Si la expresión regular usa la sintaxis `(?P<name>...)`, los argumentos *groupN* también pueden ser cadenas que identifican a los grupos por su nombre de grupo. Si un argumento de cadena no se usa como nombre de grupo en el patrón, se produce una excepción `IndexError`.

Un ejemplo moderadamente complicado:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Los grupos nombrados también pueden ser referidos por su índice:

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

Si un grupo coincide varias veces, sólo se puede acceder a la última coincidencia:


```
>>> m = re.match(r"(.)+", "a1b2c3") # Matches 3 times.
>>> m.group(1)                       # Returns only the last match.
'c3'
```

`Match.__getitem__(g)`

Esto es idéntico a `m.group(g)`. Esto permite un acceso más fácil a un grupo individual de una coincidencia:

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]          # The entire match
'Isaac Newton'
>>> m[1]          # The first parenthesized subgroup.
'Isaac'
>>> m[2]          # The second parenthesized subgroup.
'Newton'
```

Nuevo en la versión 3.6.

`Match.groups (default=None)`

Retorna una tupla que contenga todos los subgrupos de la coincidencia, desde 1 hasta tantos grupos como haya en el patrón. El argumento *default* («por defecto») se utiliza para los grupos que no participaron en la coincidencia; por defecto es `None`.

Por ejemplo:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

Si hacemos que el decimal y todo lo que sigue sea opcional, no todos los grupos podrían participar en la coincidencia. Estos grupos serán por defecto `None` a menos que se utilice el argumento *default*:

```
>>> 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')
```

`Match.groupdict (default=None)`

Retorna un diccionario que contiene todos los subgrupos *nombrados* de la coincidencia, tecleado por el nombre del subgrupo. El argumento *por defecto* se usa para los grupos que no participaron en la coincidencia; por defecto es `None`. Por ejemplo:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

`Match.start ([group])`

`Match.end ([group])`

Retorna los índices del comienzo y el final de la subcadena coincidiendo con el *group*; el *group* por defecto es cero (es decir, toda la subcadena coincidente). Retorna `-1` si *grupo* existe pero no ha contribuido a la coincidencia. Para un objeto coincidente *m*, y un grupo *g* que sí contribuyó a la coincidencia, la subcadena coincidente con el grupo *g* (equivalente a `m.group(g)`) es

```
m.string[m.start(g):m.end(g)]
```

Notar que `m.start(group)` será igual a `m.end(group)` si *group* coincidió con una cadena nula. Por ejemplo, después de `m = re.search('b(c?)', 'cba')`, `m.start(0)` es 1, `m.end(0)` es 2, `m.start(1)` y `m.end(1)` son ambos 2, y `m.start(2)` produce una excepción *IndexError*.

Un ejemplo que eliminará *remove_this* («quita esto») de las direcciones de correo electrónico:

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

Match.span([group])

Para una coincidencia *m*, retorna la tupla-2 (*m.inicio(grupo)*, *m.fin(grupo)*). Notar que si *group* no contribuyó a la coincidencia, esto es *(-1, -1)*. *group* por se convierte a cero para toda la coincidencia.

Match.pos

El valor de *pos* que fue pasado al método *search()* o *match()* de un *objeto regex*. Este es el índice de la cadena en la que el motor RE comenzó a buscar una coincidencia.

Match.endpos

El valor de *endpos* que se pasó al método *search()* o *match()* de un *objeto regex*. Este es el índice de la cadena más allá de la cual el motor RE no irá.

Match.lastindex

El índice entero del último grupo de captura coincidente, o "None" si no hay ningún grupo coincidente. Por ejemplo, las expresiones *(a)b*, *((a)(b))* y *((ab))* tendrán *lastindex == 1* si se aplican a la cadena 'ab', mientras que la expresión *(a)(b)* tendrá *lastindex == 2*, si se aplica a la misma cadena.

Match.lastgroup

El nombre del último grupo capturador coincidente, o "None" si el grupo no tenía nombre, o si no había ningún grupo coincidente.

Match.re

El *objeto de expresión regular* cuyo método *match()* o *search()* produce esta instancia de coincidencia.

Match.string

La cadena pasada a *match()* o *search()*.

Distinto en la versión 3.7: Se añadió el soporte de *copy.copy()* y *copy.deepcopy()*. Los objetos de coincidencia se consideran atómicos.

6.2.5 Ejemplos de expresiones regulares

Buscando un par

En este ejemplo, se utilizará la siguiente función de ayuda para mostrar los objetos de coincidencia con un poco más de elegancia:

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

Supongamos que se está escribiendo un programa de póquer en el que la mano de un jugador se representa como una cadena de 5 caracteres en la que cada carácter representa una carta, «a» para el as, «k» para el rey, «q» para la reina, «j» para la jota, «t» para el 10, y del «2» al «9» representando la carta con ese valor.

Para ver si una cadena dada es una mano válida, se podría hacer lo siguiente:

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q")) # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e")) # Invalid.
"<Match: 'akt5e', groups=()>"
>>> displaymatch(valid.match("akt")) # Invalid.
"<Match: 'akt', groups=()>"
>>> displaymatch(valid.match("727ak")) # Valid.
"<Match: '727ak', groups=()>"
```

Esa última mano, "727ak", contenía un par, o dos de las mismas cartas de valor. Para igualar esto con una expresión regular, se podrían usar referencias inversas como tales:

```
>>> 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',)>"
```

Para averiguar en qué carta consiste el par, se podría utilizar el método `group()` del objeto de coincidencia de la siguiente manera:

```
>>> pair = re.compile(r".*(.)*\1")
>>> 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'
```

Simular scanf()

Python no tiene actualmente un equivalente a `scanf()`. Las expresiones regulares son generalmente más poderosas, aunque también más verbosas, que las cadenas de formato `scanf()`. La tabla siguiente ofrece algunos mapeos más o menos equivalentes entre tokens de formato `scanf()` y expresiones regulares.

Token <code>scanf()</code>	Expresión regular
<code>%c</code>	<code>.</code>
<code>%5c</code>	<code>.{5}</code>
<code>%d</code>	<code>[-+]? \d+</code>
<code>%e, %E, %f, %g</code>	<code>[-+]? (\d+ (\.\d*)? \.\d+) ([eE] [-+]? \d+)?</code>
<code>%i</code>	<code>[-+]? (0[xX] [\dA-Fa-f]+ 0[0-7]* \d+)</code>
<code>%o</code>	<code>[-+]? [0-7]+</code>
<code>%s</code>	<code>\S+</code>
<code>%u</code>	<code>\d+</code>
<code>%x, %X</code>	<code>[-+]? (0[xX])? [\dA-Fa-f]+</code>

Para extraer el nombre de archivo y los números de una cadena como

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

se usaría un formato `scanf()` como

```
%s - %d errors, %d warnings
```

La expresión regular equivalente sería

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python ofrece dos operaciones primitivas diferentes basadas en expresiones regulares: `re.match()` comprueba si hay una coincidencia sólo al principio de la cadena, mientras que `re.search()` comprueba si hay una coincidencia en cualquier parte de la cadena (esto es lo que hace Perl por defecto).

Por ejemplo:

```
>>> re.match("c", "abcdef")      # No match
>>> re.search("c", "abcdef")     # Match
<re.Match object; span=(2, 3), match='c'>
```

Las expresiones regulares que comienzan con '^' pueden ser usadas con `search()` para restringir la coincidencia al principio de la cadena:

```
>>> re.match("c", "abcdef")      # No match
>>> re.search("^c", "abcdef")    # No match
>>> re.search("^a", "abcdef")    # Match
<re.Match object; span=(0, 1), match='a'>
```

Notar, sin embargo, que en el modo `MULTILINE` `match()` sólo coincide al principio de la cadena, mientras que usando `search()` con una expresión regular que comienza con '^' coincidirá al principio de cada línea.

```
>>> re.match('X', 'A\nB\nX', re.MULTILINE) # No match
>>> re.search('^X', 'A\nB\nX', re.MULTILINE) # Match
<re.Match object; span=(4, 5), match='X'>
```

Haciendo una guía telefónica

`split()` divide una cadena en una lista delimitada por el patrón recibido. El método es muy útil para convertir datos textuales en estructuras de datos que pueden ser fácilmente leídas y modificadas por Python, como se demuestra en el siguiente ejemplo en el que se crea una guía telefónica.

Primero, aquí está la información. Normalmente puede venir de un archivo, aquí se usa la sintaxis de cadena de triple comilla

```
>>> 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"""
```

Las entradas (*entries*) están separadas por una o más líneas nuevas. Ahora se convierte la cadena en una lista en la que cada línea no vacía tiene su propia entrada:

```
>>> 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']
```

Finalmente, se divide cada entrada en una lista con nombre, apellido, número de teléfono y dirección. Se utiliza el parámetro `maxsplit` (división máxima) de `split()` porque la dirección tiene espacios dentro del patrón de división:

```
>>> [re.split("?: ", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
```

(continué en la próxima página)

(proviene de la página anterior)

```
[ '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']]
```

El patrón `:?` coincide con los dos puntos después del apellido, de manera que no aparezca en la lista de resultados. Con `maxsplit` de 4, se podría separar el número de casa del nombre de la calle:

```
>>> [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']]
```

Mungear texto

`sub()` reemplaza cada ocurrencia de un patrón con una cadena o el resultado de una función. Este ejemplo demuestra el uso de `sub()` con una función para «mungear» (*munge*) el texto, o aleatorizar el orden de todos los caracteres en cada palabra de una frase excepto el primer y último carácter:

```
>>> 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)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reopr t yuor asnebces potlmpy.'
```

Encontrar todos los adverbios

`findall()` coincide con *todas* las ocurrencias de un patrón, no sólo con la primera, como lo hace `search()`. Por ejemplo, si un escritor quisiera encontrar todos los adverbios en algún texto, podría usar `findall()` de la siguiente manera:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

Encontrar todos los adverbios y sus posiciones

Si uno quiere más información sobre todas las coincidencias de un patrón en lugar del texto coincidente, `finditer()` es útil ya que proporciona *objetos de coincidencia* en lugar de cadenas. Continuando con el ejemplo anterior, si un escritor quisiera encontrar todos los adverbios y *sus posiciones* en algún texto, usaría `finditer()` de la siguiente manera:

```
>>> 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
```

Notación de cadena *raw*

La notación de cadena *raw* (`r "text"`) permite escribir expresiones regulares razonables. Sin ella, para «escapar» cada barra inversa (`'\''`) en una expresión regular tendría que ser precedida por otra. Por ejemplo, las dos siguientes líneas de código son funcionalmente idénticas:

```
>>> re.match(r"\W(.)\1\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
```

Cuando uno quiere igualar una barra inversa literal, debe escaparse en la expresión regular. Con la notación de cadena *raw*, esto significa `r"\"`. Sin la notación de cadena, uno debe usar `"\\\"`, haciendo que las siguientes líneas de código sean funcionalmente idénticas:

```
>>> re.match(r"\"", r"\"")
<re.Match object; span=(0, 1), match='\"'>
>>> re.match("\\\"", r"\\")
<re.Match object; span=(0, 1), match='\"'>
```

Escribir un Tokenizador

Un **tokenizador** o **analizador léxico** analiza una cadena para categorizar grupos de caracteres. Este es un primer paso útil para escribir un compilador o intérprete.

Las categorías de texto se especifican con expresiones regulares. La técnica consiste en combinarlas en una única expresión regular maestra y en hacer un bucle sobre las sucesivas coincidencias:

```
from typing import NamedTuple
import re

class Token(NamedTuple):
    type: str
    value: str
    line: int
    column: int

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'), # Integer or decimal number
        ('ASSIGN',   r':='),          # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),   # Identifiers
        ('OP',       r'[+-*/*]'),     # Arithmetic operators
        ('NEWLINE',  r'\n'),          # Line endings
        ('SKIP',     r'[ \t]+'),      # Skip over spaces and tabs
        ('MISMATCH', r'.'),           # Any other character
    ]
    tok_regex = '|'.join('%s' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        kind = value
```

(continué en la próxima página)

(proviene de la página anterior)

```

    elif kind == 'NEWLINE':
        line_start = mo.end()
        line_num += 1
        continue
    elif kind == 'SKIP':
        continue
    elif kind == 'MISMATCH':
        raise RuntimeError(f'{value!r} unexpected on line {line_num}')
    yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
'''

for token in tokenize(statements):
    print(token)

```

El tokenizador produce el siguiente resultado:

```

Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)

```

6.3 difflib — Funciones auxiliares para calcular deltas

Código fuente: [Lib/difflib.py](#)

This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce information about file differences in various formats, including HTML and context and unified diffs. For comparing directories and files, see also, the [filecmp](#) module.

class `difflib.SequenceMatcher`

Esta es una clase flexible para comparar pares de secuencias de cualquier tipo, siempre que los elementos de la secuencia sean *hashable*. El algoritmo básico es anterior, y un poco mas agradable, que el publicado a fines de los 80” por Ratcliff y Obershelp, bajo el nombre hiperbólico de «*gestalt pattern matching*». La idea es encontrar la subsecuencia coincidente contigua mas larga que no contenga elementos «no deseados»; estos elementos «no deseados» son aquellos que no son de interés por algún motivo, como ser líneas en blanco o espacios. (El tratamiento de elementos no deseados es una extensión al algoritmo de Ratcliff y Obershelp). La

misma idea se aplica recursivamente a las partes de la secuencia a la derecha e izquierda de cada subsecuencia correspondiente. Esto no proporciona secuencias de edición mínimas, pero tiende a producir coincidencias que «parecen correctas» a las personas.

Complejidad temporal: En el peor de los casos el algoritmo Ratcliff-Obershelp básico es de complejidad cúbica y de complejidad temporal cuadrática en el caso esperado. *SequenceMatcher* es de complejidad temporal cuadrática en el peor de los casos y el comportamiento del caso esperado depende de manera complicada de cuántos elementos tienen en común las secuencias; en el mejor de los casos la complejidad temporal es lineal.

Heurística automática de elementos no deseados: *SequenceMatcher* implementa un método heurístico que identifica automáticamente a ciertos elementos como no deseados. El método heurístico consiste en contar cuantas veces aparece cada elemento en la secuencia. Si las apariciones del duplicado de un elemento (después del primero) contabilizan mas del 1% de la secuencia, y a su vez la secuencia contiene mas de 200 elementos, este es identificado como «popular» y es considerado no deseado. Este método heurístico puede desactivarse estableciendo el argumento `autojunk` como `False` al crear la clase *SequenceMatcher*.

Nuevo en la versión 3.2: El parámetro *autojunk*.

class `difflib.Differ`

Esta clase se utiliza para comparar secuencias de líneas de texto y producir diferencias o deltas en una forma legible por humanos. *Differ* usa *SequenceMatcher* tanto para comparar secuencias de líneas, como para comparar secuencias de caracteres entre líneas similares.

Cada línea de un delta de *Differ* comienza con un código de dos letras:

Código	Significado
'- '	línea única para la secuencia 1
'+'	línea única para la secuencia 2
' ' '	línea común a ambas secuencias
'? '	línea ausente en todas las secuencias de entrada

Las líneas que empiezan con “?” intentan guiar al ojo hacia las diferencias intralíneas, y no estuvieron presentes en ninguna de las secuencias de entrada. Estas líneas pueden ser confusas si la secuencia contiene caracteres de tabulación.

class `difflib.HtmlDiff`

Esta clase puede ser usada para crear una tabla HTML (o un archivo HTML completo que contenga la tabla) mostrando comparaciones lado a lado y línea por línea del texto, con cambios interlineales e intralíneales. La tabla se puede generar en modo de diferencia completa o contextual.

El constructor de esta clase es:

__init__ (*tabsize=8, wrapcolumn=None, linejunk=None, charjunk=IS_CHARACTER_JUNK*)

Inicializa una instancia de *HtmlDiff*.

tabsize es un argumento por palabra clave opcional para especificar el espaciado de tabulación. Su valor predeterminado es 8.

wrapcolumn es un argumento por palabra clave opcional para especificar el número de columnas donde las líneas serán divididas y ajustadas al ancho de columna, su valor por defecto es `None`, donde las líneas no son ajustadas.

linejunk y *charjunk* son argumentos por palabra clave opcionales que serán pasados a *ndiff()* (que es utilizado por *HtmlDiff* para generar las diferencias lado a lado en HTML). Refiérase a la documentación de *ndiff()* para conocer los detalles y valores por defecto de sus argumentos.

Los siguientes métodos son públicos:

make_file (*fromlines, tolines, fromdesc="", todesc="", context=False, numlines=5, *, charset='utf-8'*)

Compara *fromlines* y *toline*s (listas de cadenas de texto) y retorna una cadena de caracteres que representa un archivo HTML completo que contiene una tabla mostrando diferencias línea por línea del texto con cambios interlineales e intralíneales resaltados.

fromdesc y *todesc* son argumentos por palabra clave opcionales para especificar los encabezados de las columnas desde *fromdesc* hasta *todesc* en el archivo (ambas cadenas están vacías por defecto).

context y *numlines* son parámetros opcionales. Establezca *context* como `True` para mostrar diferencias contextuales, de lo contrario su valor por defecto es `False` que muestra los archivos completos. El valor por defecto de *numlines* es 5. Cuando *context* es `True`, *numlines* controla el número de líneas de contexto que rodean las diferencias resaltadas. Cuando *context* es `False`, *numlines* controla el número de líneas que se muestran antes de una diferencia resaltada cuando se usan los hipervínculos «next» (un valor de cero produce que los hipervínculos «next» ubiquen el siguiente resaltado en la parte superior del navegador, sin ningún contexto principal).

Nota: *fromdesc* y *todesc* se interpretan como HTML no escapado y se deben escapar correctamente si los datos son recibidos de fuentes no confiables.

Distinto en la versión 3.5: Se agregó el argumento sólo de palabra clave *charset*. La codificación de caracteres por defecto para documentos HTML se cambió de `'ISO-8859-1'` a `'utf-8'`.

make_table (*fromlines*, *tolines*, *fromdesc*=", *todesc*", *context*=False, *numlines*=5)

Compara *fromlines* y *tolines* (listas de cadenas de texto) y retorna una cadena de caracteres que representa una tabla HTML mostrando comparaciones lado a lado y línea por línea del texto con cambios interlineales e intralineales.

Los argumentos para este método son los mismos que los del método *make_file* ().

`Tools/scripts/diff.py` es una herramienta de línea de comandos para esta clase y contiene un buen ejemplo sobre su uso.

`difflib.context_diff(a, b, fromfile=", tofile=", fromfiledate=", tofiledate=", n=3, lineterm='\n')`

Compara *a* y *b* (listas de cadenas de texto); retorna un delta (un *generator* que genera las líneas delta) en formato de diferencias de contexto.

El formato de diferencias de contexto es una forma compacta de mostrar solamente las líneas que fueron modificadas y algunas líneas adicionales de contexto. Los cambios son mostrados utilizando el estilo antes/después. El número de líneas de contexto es determinado por *n*, cuyo valor por defecto es 3.

Por defecto, las líneas de control (aquellas que comienzan con `*** o ---`) son creadas con una línea nueva. Esto es de ayuda para que las entradas creadas por `io.IOBase.readlines()` generen diferencias que puedan ser utilizadas con `io.IOBase.writelines()` ya que ambas, la entrada y la salida, tienen líneas nuevas al final.

Para entradas que no tienen nuevas líneas finales, establezca el argumento *lineterm* como `" "` de forma que la salida sea uniforme y libre de nuevas líneas.

El formato de diferencias de contexto normalmente tiene un encabezado para nombres de archivos y tiempos de modificaciones. Alguno o todos estos debe ser especificado utilizando las cadenas de texto para *fromfile*, *tofile*, *fromfiledate* y *tofiledate*. Los tiempos de modificación son normalmente expresados en formato ISO 8601. Si no es especificado las cadenas por defecto son espacios en blanco.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(context_diff(s1, s2, fromfile='before.py', tofile=
↳ 'after.py'))
*** before.py
--- after.py
*****
*** 1,4 ****
! bacon
! eggs
! ham
! guido
--- 1,4 ----
! python
```

(continué en la próxima página)

(proviene de la página anterior)

```
! eggy
! hamster
  guido
```

Vea *Una interfaz de línea de comandos para difflib* para un ejemplo mas detallado.

`difflib.get_close_matches(word, possibilities, n=3, cutoff=0.6)`

Retorna una lista de las mejores coincidencias «lo suficientemente buenas». *word* es una secuencia para la cual coincidencias cercanas son deseadas (usualmente una cadena de texto), y *possibilities* es una lista de secuencias contra la cual se compara *word* (comunmente una lista de cadenas de caracteres).

Argumento opcional *n* (por defecto 3) es el máximo número de coincidencias cercanas a retornar; *n* debe ser mayor que 0.

Argumento opcional *cutoff* (por defecto 0.6) es un flotante en el rango [0, 1]. Las posibilidades que no alcanzan un puntaje al menos similar al de *word* son ignoradas.

Las mejores (no mas de *n*) coincidencias entre las posibilidades son retornadas en una lista, ordenadas por similitud de puntaje, las mas similares primero.

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('pineapple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

`difflib.ndiff(a, b, linejunk=None, charjunk=IS_CHARACTER_JUNK)`

Compara *a* y *b* (listas de cadenas de texto); retorna un delta del estilo *Differ* (un *generator* que genera los deltas de las líneas).

Parámetros de palabra clave opcional *linejunk* y *charjunk* son funciones de filtrado (o `None`):

linejunk: Una función que acepta una sola cadena de caracteres como argumento, y retorna verdadero si la cadena de texto es un elemento no deseado, o falso si no lo es. Su valor por defecto es `None`. Hay también una función a nivel del módulo `IS_LINE_JUNK()`, que filtra líneas sin caracteres visibles, excepto como mucho un carácter de libra ('#') – de cualquier forma la clase subyacente *SequenceMatcher* realiza un análisis dinámico sobre cuáles líneas son tan frecuentes como para constituir ruido, y esto usualmente funciona mejor que utilizando esta función.

charjunk: Una función que acepta un carácter (una cadena de caracteres de longitud 1) como argumento, y retorna *True* si el carácter es un elemento no deseado, o *False* si no lo es. El valor por defecto es una función a nivel del módulo `IS_CHARACTER_JUNK()`, que filtra caracteres de espacios en blanco (un espacio en blanco o tabulación; es una mala idea incluir saltos de líneas en esto!)

`Tools/scripts/ndiff.py` es una interfaz de línea de comandos para esta función.

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...             'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

`difflib.restore(sequence, which)`

Retorna uno de las dos secuencias que generaron un delta.

Dada una *sequence* producida por `Differ.compare()` o `ndiff()`, extrae las líneas originadas por el archivo 1 o 2 (parámetro *which*), quitando los prefijos de la línea.

Ejemplo:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...             'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print(''.join(restore(diff, 1)), end="")
one
two
three
>>> print(''.join(restore(diff, 2)), end="")
ore
tree
emu
```

`difflib.unified_diff(a, b, fromfile="", tofile="", fromfiledate="", tofiledate="", n=3, lineterm='\n')`

Compara *a* y *b* (listas de cadenas de caracteres); retorna un delta (un *generator* que genera los delta de líneas) en formato de diferencias unificado.

Las diferencias unificadas son una forma compacta de mostrar sólo las líneas que presentan cambios y algunas líneas de contexto adicionales. Los cambios son mostrados en una sola línea (en lugar de bloques separados antes y después). El número de líneas de contexto se establece mediante *n*, cuyo valor por defecto es tres.

Por defecto, las líneas de control de diferencias (aquellas con ---, +++, o @@) son creadas con un salto de línea nuevo. Esto es de ayuda para que las entradas creadas por `io.IOBase.readlines()` generen diferencias que puedan ser utilizadas con `io.IOBase.writelines()` ya que ambas, la entrada y la salida, tienen líneas nuevas al final.

Para entradas que no tienen nuevas líneas finales, establezca el argumento *lineterm* como "" de forma que la salida sea uniforme y libre de nuevas líneas.

El formato de diferencias de contexto normalmente tiene un encabezado para nombres de archivos y tiempos de modificaciones. Alguno o todos estos debe ser especificado utilizando las cadenas de texto para *fromfile*, *tofile*, *fromfiledate* y *tofiledate*. Los tiempos de modificación son normalmente expresados en formato ISO 8601. Si no es especificado las cadenas por defecto son espacios en blanco.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(unified_diff(s1, s2, fromfile='before.py', tofile=
→ 'after.py'))
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
 guido
```

Vea *Una interfaz de línea de comandos para difflib* para un ejemplo mas detallado.

`difflib.diff_bytes(dfunc, a, b, fromfile=b", tofile=b", fromfiledate=b", tofiledate=b", n=3, lineterm=b'\n')`

Compara *a* y *b* (listas de objetos de bytes) usando *dfunc*; produce una secuencia de líneas delta (también bytes) en el formato retornado por *dfunc*. *dfunc* debe ser invocable, comúnmente cualquiera de `unified_diff()` o `context_diff()`.

Permite comparar datos con codificación desconocida o inconsistente. Todas las entradas, excepto *n*, deben ser objetos de bytes, no cadenas de texto. Funciona convirtiendo sin pérdidas todas las entradas (excepto *n*) a cadenas de texto, e invoca `dfunc(a, b, fromfile, tofile, fromfiledate, tofiledate, n, lineterm)`. La salida de *dfunc* es entonces convertida nuevamente a bytes, de forma que las líneas delta que son recibidas tienen la misma codificación desconocida/inconsistente que *a* y *b*.

Nuevo en la versión 3.5.

`difflib.IS_LINE_JUNK(line)`

Retorna `True` para líneas que deben ser ignoradas. La línea *line* es ignorada si *line* es un espacio vacío o contiene un solo '#', en cualquier otro caso no es ignorado. Es usado como valor por defecto para el parámetro *linejunk* por `ndiff()` en versiones anteriores.

`difflib.IS_CHARACTER_JUNK(ch)`

Retorna `True` para los caracteres que deben ser ignorados. El carácter *ch* es ignorado si *ch* es un espacio en blanco o tabulación, en cualquier otro caso no es ignorado. Es utilizado como valor por defecto para el parámetro *charjunk* en `ndiff()`.

Ver también:

Pattern Matching: The Gestalt Approach Discusión de un algoritmo similar por John W. Ratcliff y D. E. Metzner. Esto fue publicado en *Dr. Dobbs' Journal* en Julio de 1988.

6.3.1 Objetos *SequenceMatcher*

La clase *SequenceMatcher* tiene este constructor:

class `difflib.SequenceMatcher(isjunk=None, a="", b="", autojunk=True)`

El argumento opcional *isjunk* debe ser `None` (que es su valor por defecto) o una función de un solo argumento que reciba un elemento de la secuencia y retorne verdadero si y solo si el elemento es no deseado y deba ser ignorado. Pasar el argumento *isjunk* como `None` es equivalente a pasar `lambda x: False`; en otras palabras, ningún elemento es ignorado. Por ejemplo, pasar:

```
lambda x: x in " \t"
```

si se están comparando líneas como secuencias de caracteres, y no se quiere sincronizar en espacios blancos o tabulaciones.

Los argumentos opcionales *a* y *b* son las secuencias a comparar; ambos tienen como valor por defecto una cadena de texto vacía. Los elementos de ambas secuencias deben ser *hashable*.

El argumento opcional *autojunk* puede ser usado para deshabilitar la heurística automática de elementos no deseados.

Nuevo en la versión 3.2: El parámetro *autojunk*.

Los objetos *SequenceMatcher* reciben tres atributos: *bjunk* es el conjunto de elementos de *b* para los cuales *isjunk* es `True`; *bpopular* es el set de elementos que no son basura considerados populares por la heurística (si no es que fue deshabilitada); *b2j* es un diccionario que mapea elementos de *b* a una lista de posiciones donde estos ocurren. Los tres atributos son reseteados cuando *b* es reseteado mediante `set_seqs()` o `set_seq2()`.

Nuevo en la versión 3.2: Los atributos *bjunk* y *bpopular*.

Los objetos *SequenceMatcher* tienen los siguientes métodos:

set_seqs(a, b)

Establece las dos secuencias a ser comparadas.

SequenceMatcher calcula y almacena información detallada sobre la segunda secuencia, con lo cual si quieres comparar una secuencia contra muchas otras, usa `set_seq2()` para establecer la secuencia común una sola vez y llamar `set_seq1()` repetidamente, una vez por cada una de las otras secuencias.

set_seq1(a)

Establece la primera secuencia a ser comparada. La segunda secuencia a ser comparada no es modificada.

set_seq2 (b)

Establece la segunda secuencia a ser comparada. La primera secuencia a ser comparada no es modificada.

find_longest_match (alo, ahi, blo, bhi)

Encuentra el bloque de coincidencia mas largo en `a[alo:ahi]` y `b[blo:bhi]`.

Si *isjunk* fue omitido o es `None`, `find_longest_match()` retorna `(i, j, k)` tal que `a[i:i+k]` es igual a `b[j:j+k]`, donde `alo <= i <= i+k <= ahi` y `blo <= j <= j+k <= bhi`. Para todo `(i', j', k')` cumpliendo esas condiciones, las condiciones adicionales `k >= k'`, `i <= i'`, y si `i == i'`, `j <= j'` también se cumplen. En otras palabras, de todos los bloques coincidentes máximos, retorna aquel que comienza antes en *a*, y de todos esos bloques coincidentes máximos que comienzan antes en *a*, retorna aquel que comienza antes en *b*.

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

Si se proporcionó *isjunk*, primero se determina el bloque coincidente mas largo como fue indicado anteriormente, pero con la restricción adicional de que no aparezca ningún elemento no deseado en el bloque. Entonces, ese bloque se extiende tan lejos como sea posible haciendo coincidir (solo) elementos no deseados de ambos lados. Por lo tanto, el bloque resultante nunca hará coincidir ningún elemento no deseado, excepto que un elemento no deseado idéntico pase a ser adyacente a una coincidencia interesante.

Este es el mismo ejemplo que el mostrado anteriormente, pero considerando elementos en blanco como no deseados. Esto previene que ' abcd' sea coincidente con 'abcd' en el final de la segunda secuencia directamente. En cambio, sólo el 'abcd' puede coincidir, y coincide con el 'abcd' que se encuentre mas a la izquierda en la segunda secuencia:

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

Si no coincide ningún bloque, esto retorna `(alo, blo, 0)`.

Este método retorna un *named tuple* `Match(a, b, size)`.

get_matching_blocks ()

Retorna una lista de triplas (tuplas de tres elementos) describiendo subsecuencias coincidentes no superpuestas. Cada tripla sigue el formato `(i, j, n)`, y significa que `a[i:i+n] == b[j:j+n]`. Las triplas son monótonamente crecientes en *i* y *j*.

La última tripla es un objeto ficticio (dummy), y tiene el valor `(len(a), len(b), 0)`. Es la única tripla con `n == 0`. Si `(i, j, n)` y `(i', j', n')` son triplas adyacentes en la lista, y la segunda no es el último elemento de la lista, entonces `i+n < i'` o `j+n < j'`; en otras palabras, las triplas adyacentes describen bloques iguales no adyacentes.

```
>>> 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 ()

Retorna una lista de quintuplas (tuplas de cinco elementos) describiendo como convertir *a* en *b*. Cada tupla tiene la forma `(tag, i1, i2, j1, j2)`. En la primer tupla se cumple que `i1 == j1 == 0`, y las tuplas restantes tienen *i1* igual al *i2* de la tupla precedente, y de igual manera, *j1* igual al *j2* de la tupla anterior.

Los valores de *tag* son cadenas de caracteres, con el siguiente significado:

Valor	Significado
'replace'	<code>a[i1:i2]</code> debe ser reemplazado por <code>b[j1:j2]</code> .
'delete'	<code>a[i1:i2]</code> debe ser eliminado. Nótese que en este caso <code>j1 == j2</code> .
'insert'	<code>b[j1:j2]</code> debe ser insertado en <code>a[i1:i1]</code> . Nótese que en este caso <code>i1 == i2</code> .
'equal'	<code>a[i1:i2] == b[j1:j2]</code> (las subsecuencias son iguales).

Por ejemplo:

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7}    a[{}:{}] --> b[{}:{}] {!r:>8} --> {!r}'.format(
...         tag, i1, i2, j1, j2, a[i1:i2], b[j1:j2]))
delete    a[0:1] --> b[0:0]      'q' --> ''
equal     a[1:3] --> b[0:2]      'ab' --> 'ab'
replace   a[3:4] --> b[2:3]      'x' --> 'y'
equal     a[4:6] --> b[3:5]      'cd' --> 'cd'
insert    a[6:6] --> b[5:6]      '' --> 'f'
```

get_grouped_opcodes (*n=3*)

Retorna un *generator* de grupos de hasta *n* líneas de contexto.

Empezando con los grupos retornados por `get_opcodes()`, este método separa grupos con cambios menores y elimina los rangos intermedios que no tienen cambios.

Los grupos son retornados en el mismo formato que `get_opcodes()`.

ratio ()

Retorna una medida de la similitud de las secuencias como un flotante en el rango [0, 1].

Donde T es el número total de elementos en ambas secuencias y M es el número de coincidencias, esto es $2.0 * M / T$. Nótese que esto es 1.0 si las secuencias son idénticas y 0.0 si no tienen nada en común.

Esto es computacionalmente costoso si `get_matching_blocks()` o `get_opcodes()` no fueron ejecutados, in tal caso deberías considerar primero `quick_ratio()` o `real_quick_ratio()` para obtener un límite superior.

Nota: Precaución: El resultado de una llamada a `ratio()` puede depender del orden de los argumentos. Por ejemplo:

```
>>> SequenceMatcher(None, 'tide', 'diet').ratio()
0.25
>>> SequenceMatcher(None, 'diet', 'tide').ratio()
0.5
```

quick_ratio ()

Retorna un límite superior en `ratio()` relativamente rápido.

real_quick_ratio ()

Retorna un límite superior en `ratio()` muy rápido.

Los tres métodos que retornan la proporción de coincidencias con el total de caracteres pueden dar diferentes resultados debido a los distintos niveles de aproximación, a pesar de que `quick_ratio()` y `real_quick_ratio()` son siempre al menos tan grandes como `ratio()`:

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

6.3.2 SequenceMatcher Ejemplos

Este ejemplo compara dos cadenas de texto, considerando los espacios en blanco como caracteres no deseados:

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                       "private Thread currentThread;",
...                       "private volatile Thread currentThread;")
```

`ratio()` retorna un flotante en el rango [0, 1], cuantificando la similitud entre las secuencias. Siguiendo la regla del pulgar, un `ratio()` por encima de 0.6 significa que las secuencias son coincidencias cercanas:

```
>>> print(round(s.ratio(), 3))
0.866
```

Si solamente estás interesado en cuándo las secuencias coinciden, `get_matching_blocks()` es útil:

```
>>> 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
```

Nótese que la última tupla retornada por `get_matching_blocks()` es siempre un objeto ficticio (dummy), `(len(a), len(b), 0)`, y este es el único caso en el cual el último elemento de la tupla (el número de elementos coincidentes) es 0.

Si quieres saber como cambiar la primer secuencia con la segunda, usa `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]
```

Ver también:

- La función `get_close_matches()` en este módulo que muestra lo simple que es el código que construye `SequenceMatcher` puede ser utilizada para hacer un trabajo útil.
- Una receta simple de un controlador de versiones para una aplicación pequeña construida con `SequenceMatcher`.

6.3.3 Objetos Differ

Nótese que los deltas generados por `Differ` no dicen ser diferencias **mínimas**. Todo lo contrario, las diferencias mínimas suelen ser contra-intuitivas, ya que se sincronizan en cualquier lugar posible, a veces coinciden accidentalmente con 100 páginas de diferencia. Restringiendo los puntos de sincronización a coincidencias contiguas se preserva cierta noción de cercanía, con el costo adicional de producir diferencias mas largas.

La clase `Differ` tiene el siguiente constructor:

```
class difflib.Differ (linejunk=None, charjunk=None)
```

Parámetros de palabra clave opcionales `linejunk` y `charjunk` son para funciones de filtrado (o `None`):

linejunk: Una función que acepta una sola cadena de texto como argumento y retorna verdadero si la cadena de texto es un elemento no deseado. Su valor por defecto es `None`, lo que significa que ninguna línea es considerada no deseada.

charjunk: Una función que acepta un solo carácter como argumento (una cadena de caracteres de longitud 1) y retorna verdadero si el carácter es un elemento no deseado. Su valor por defecto es `None`, lo que significa que ningún carácter es considerado no deseado.

Estas funciones de elementos no deseados aceleran la coincidencia para encontrar diferencias y no hacen que se ignoren líneas o caracteres diferentes. Lea la descripción del parámetro *isjunk* en el método `find_longest_match()` para una explicación mas detallada.

Los objetos *Differ* son usados (una vez generados los deltas) mediante un solo método:

compare (*a*, *b*)

Compara dos secuencias de líneas y genera el delta correspondiente (una secuencia de líneas).

Cada secuencia debe contener cadenas de texto individuales de una sola línea terminadas con una línea nueva. Este tipo de secuencias pueden ser obtenidas mediante el método `readlines()` de objetos de tipo archivo. Los delta generados consisten también en cadenas de texto terminadas en nuevas líneas, listas para imprimirse tal cual a través del método `writelines()` de un objeto de tipo archivo.

6.3.4 Ejemplo de *Differ*

Este ejemplo compara dos textos. Primero preparamos los textos, secuencias de cadenas de texto individuales de una sola línea terminadas con una línea nueva (este tipo de secuencias también pueden ser obtenidas mediante el método `readlines()` de objetos de tipo archivo):

```
>>> 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(keepends=True)
>>> 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(keepends=True)
```

Luego instanciamos el objeto *Differ*:

```
>>> d = Differ()
```

Nótese que cuando instanciamos un objeto *Differ* deberíamos pasar funciones para filtrar líneas y caracteres no deseados. Consulte el constructor de *Differ()* para mas detalles.

Finalmente, comparamos las dos:

```
>>> result = list(d.compare(text1, text2))
```

`result` es una lista de cadenas de caracteres, entonces vamos a mostrarlo de una forma elegante:

```
>>> 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']
```

Representado como una sola cadena de caracteres de múltiples líneas se ve así:


```
>>> import sys
>>> sys.stdout.writelines(result)
1. Beautiful is better than ugly.
- 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.
```

6.3.5 Una interfaz de línea de comandos para `difflib`

Este ejemplo muestra como usar `difflib` para crear una herramienta de diferencias. También puedes encontrarla en la distribución estándar de Python como `Tools/scripts/diff.py`.

```
#!/usr/bin/env python3
""" 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, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime,
                              timezone.utc)
    return t.astimezone().isoformat()

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true', default=False,
                        help='Produce a context format diff (default)')
    parser.add_argument('-u', action='store_true', default=False,
                        help='Produce a unified format diff')
    parser.add_argument('-m', action='store_true', default=False,
                        help='Produce HTML side by side diff '
                             '(can use -c and -l in conjunction)')
    parser.add_argument('-n', action='store_true', default=False,
                        help='Produce a ndiff format diff')
    parser.add_argument('-l', '--lines', type=int, default=3,
                        help='Set number of context lines (default 3)')
    parser.add_argument('fromfile')
    parser.add_argument('tofile')
    options = parser.parse_args()

    n = options.lines
    fromfile = options.fromfile
    tofile = options.tofile

    fromdate = file_mtime(fromfile)
    todate = file_mtime(tofile)
```

(continué en la próxima página)

(proviene de la página anterior)

```

with open(fromfile) as ff:
    fromlines = ff.readlines()
with open(tofile) as tf:
    tolines = tf.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)

sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()

```

6.4 textwrap — Envoltura y relleno de texto

Source code: [Lib/textwrap.py](#)

El módulo `textwrap` proporciona algunas funciones de conveniencia, así como `TextWrapper`, la clase que hace todo el trabajo. Si sólo estás envolviendo o rellenando una o dos cadenas de texto, las funciones de conveniencia deberían ser lo suficientemente buenas; de lo contrario, deberías usar una instancia de `TextWrapper` para mayor eficiencia.

`textwrap.wrap(text, width=70, **kwargs)`

Envuelve el párrafo individual en `text` (una cadena) para que cada línea tenga como máximo `width` de caracteres de largo. Retorna una lista de líneas de salida, sin las nuevas líneas finales.

Los argumentos opcionales de las palabras clave corresponden a los atributos de la instancia de `TextWrapper`, documentados a continuación. `width` por defecto es 70.

Ver el método `TextWrapper.wrap()` para más detalles sobre el comportamiento de `wrap()`.

`textwrap.fill(text, width=70, **kwargs)`

Envuelve el único párrafo en `text`, y retorna una sola cadena que contiene el párrafo envuelto. `fill()` es la abreviatura de

```
"\n".join(wrap(text, ...))
```

En particular, `fill()` acepta exactamente los mismos argumentos de palabras clave que `wrap()`.

`textwrap.shorten(text, width, **kwargs)`

Colapsa y trunca el `text` dado para que encaje en el `width` dado.

Primero el espacio blanco en `text` se colapsa (todos los espacios blancos son reemplazados por espacios sencillos). Si el resultado cabe en el `width`, se retorna. En caso contrario, se eliminan suficientes palabras del final para que las palabras restantes más el placeholder encajen dentro de `width`:

```

>>> textwrap.shorten("Hello world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello world!", width=11)

```

(continué en la próxima página)

(proviene de la página anterior)

```
'Hello [...]'
>>> textwrap.shorten("Hello world", width=10, placeholder="...")
'Hello...'
```

Los argumentos opcionales de las palabras clave corresponden a los atributos de la instancia de `TextWrapper`, documentados a continuación. Observe que el espacio en blanco se colapsa antes de pasar el texto a la función `TextWrapper.fill()`, por lo que cambiar el valor de `tabsize`, `expand_tabs`, `drop_whitespace`, y `replace_whitespace` no tendrá ningún efecto.

Nuevo en la versión 3.4.

`textwrap.dedent(text)`

Elimina cualquier espacio en blanco común de cada línea de `text`.

Esto puede utilizarse para hacer que las cadenas con comillas triples se alineen con el borde izquierdo de la pantalla, mientras que se siguen presentando en el código fuente en forma indentada.

Nótese que los tabuladores y los espacios se tratan como espacios en blanco, pero no son iguales: las líneas "hello" y "\thello" se consideran que no tienen un espacio en blanco común.

Las líneas que sólo contienen espacios en blanco se ignoran en la entrada y se normalizan a un solo carácter de nueva línea en la salida.

Por ejemplo:

```
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'
```

`textwrap.indent(text, prefix, predicate=None)`

Añade `prefix` al principio de las líneas seleccionadas en `text`.

Las líneas se separan llamando a `text.splitlines(True)`.

Por defecto, se añade `prefix` a todas las líneas que no consisten únicamente en espacios en blanco (incluyendo cualquier terminación de línea).

Por ejemplo:

```
>>> s = 'hello\n\n \nworld'
>>> indent(s, ' ')
'  hello\n\n \n world'
```

El argumento opcional `predicate` puede ser usado para controlar qué líneas están indentadas. Por ejemplo, es fácil añadir `prefix` incluso a las líneas vacías y de espacio en blanco:

```
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+
+
+ world
```

Nuevo en la versión 3.3.

`wrap()`, `fill()` y `shorten()` funcionan creando una instancia `TextWrapper` y llamando a un solo método en ella. Esa instancia no se reutiliza, por lo que para las aplicaciones que procesan muchas cadenas de texto usando `wrap()` y/o `fill()`, puede ser más eficiente crear su propio objeto `TextWrapper`.

El texto se envuelve preferentemente en espacios en blanco y justo después de los guiones en palabras con guión; sólo entonces se romperán las palabras largas si es necesario, a menos que `TextWrapper.break_long_words` sea falso.

class `textwrap.TextWrapper` (***kwargs*)

El constructor `TextWrapper` acepta un número de argumentos de palabras clave opcionales. Cada argumento de palabra clave corresponde a un atributo de la instancia, por ejemplo

```
wrapper = TextWrapper(initial_indent="* ")
```

es lo mismo que

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

Puedes reutilizar el mismo objeto `TextWrapper` muchas veces, y puedes cambiar cualquiera de sus opciones a través de la asignación directa de atributos de instancia entre usos.

Los atributos de la instancia `TextWrapper` (y los argumentos de las palabras clave para el constructor) son los siguientes:

width

(default: 70) La longitud máxima de las líneas envueltas. Mientras no haya palabras individuales en el texto de entrada más largas que `width`, `TextWrapper` garantiza que ninguna línea de salida será más larga que los caracteres `width`.

expand_tabs

(default: True) Si es verdadero, entonces todos los caracteres de tabulación en `text` serán expandidos a espacios usando el método `expandtabs()` de `text`.

tabsize

(default: 8) Si `expand_tabs` es verdadero, entonces todos los caracteres de tabulación en `text` se expandirán a cero o más espacios, dependiendo de la columna actual y el tamaño de tabulación dado.

Nuevo en la versión 3.3.

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'`).

Nota: Si `expand_tabs` es falso y `replace_whitespace` es verdadero, cada carácter del tabulador será reemplazado por un solo espacio, que *no* es lo mismo que la expansión del tabulador.

Nota: Si `replace_whitespace` es falso, las nuevas líneas pueden aparecer en medio de una línea y causar una salida extraña. Por esta razón, el texto debe ser dividido en párrafos (usando `str.splitlines()` o similar) que se envuelven por separado.

drop_whitespace

(default: True) Si es verdadero, se eliminan los espacios en blanco al principio y al final de cada línea (después de la envoltura pero antes del indentado). Sin embargo, el espacio en blanco al principio del párrafo no se elimina si lo sigue un espacio en blanco. Si el espacio blanco que se deja caer ocupa una línea entera, se deja caer toda la línea.

initial_indent

(default: ' ') Cadena que será preparada para la primera línea de salida envuelta. Cuenta hacia la longitud de la primera línea. La cadena vacía no está indentada.

subsequent_indent

(default: ' ') Cadena que se preparará para todas las líneas de salida envueltas excepto la primera. Cuenta hacia la longitud de cada línea excepto la primera.

fix_sentence_endings

(default: `False`) Si es verdadero, `TextWrapper` intenta detectar los finales de las frases y asegurarse de que las frases estén siempre separadas por dos espacios exactos. Esto es generalmente deseado para el texto en una fuente monoespaciada. Sin embargo, el algoritmo de detección de oraciones es imperfecto: asume que el final de una oración consiste en una letra minúscula seguida de una de `'.'`, `'!'`, o `'?'`, posiblemente seguida de una de `'\"'` o `'\"'`, seguida de un espacio. Un problema de este algoritmo es que no puede detectar la diferencia entre «Dr.» en

```
[...] Dr. Frankenstein's monster [...]
```

y «Spot.» en:

```
[...] See Spot. See Spot run [...]
```

`fix_sentence_endings` es falso por defecto.

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`) Si es verdadero, entonces las palabras más largas que `width` se romperán para asegurar que ninguna línea sea más larga que `width`. Si es falso, las palabras largas no se romperán, y algunas líneas pueden ser más largas que `width`. (Las palabras largas se pondrán en una línea por sí mismas, para minimizar la cantidad en que se excede `width`).

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 inseparable words. Default behaviour in previous versions was to always allow breaking hyphenated words.

max_lines

(default: `None`) Si es `None`, entonces la salida contendrá como máximo `max_lines`, con un *placeholder* que aparecerá al final de la salida.

Nuevo en la versión 3.4.

placeholder

(default: `' [...]'`) Cadena que aparecerá al final del texto de salida si ha sido truncado.

Nuevo en la versión 3.4.

`TextWrapper` también proporciona algunos métodos públicos, análogos a las funciones de conveniencia a nivel de módulo:

wrap (*text*)

Envuelve el párrafo individual en *text* (una cadena) para que cada línea tenga como máximo `width` caracteres de largo. Todas las opciones de envoltura se toman de los atributos de la instancia `TextWrapper`. Retorna una lista de líneas de salida, sin las nuevas líneas finales. Si la salida envuelta no tiene contenido, la lista retornada estará vacía.

fill (*text*)

Envuelve el único párrafo en *text*, y retorna una única cadena que contiene el párrafo envuelto.

6.5 unicodedata — Base de datos Unicode

Este módulo proporciona acceso a la base de datos de caracteres Unicode (UCD), que define las propiedades de todos los caracteres Unicode. Los datos contenidos en esta base de datos se compilan a partir de [UCD versión 12.1.0](http://www.unicode.org/Public/12.1.0/ucd/NameAliases.txt).

El módulo utiliza los mismos nombres y símbolos definidos por el Anexo #44 del estándar Unicode de la «Base de datos de caracteres Unicode». Define las siguientes funciones:

`unicodedata.lookup(name)`

Busca el carácter por su nombre. Si se encuentra un carácter con el nombre proporcionado, retornará el carácter correspondiente. Si no se encuentra, se lanza una excepción `KeyError`.

Distinto en la versión 3.3: Se ha agregado soporte para alias de nombre¹ y secuencias con nombre².

`unicodedata.name(chr[, default])`

Retorna el nombre asignado al carácter `chr` como una cadena de caracteres. Si no se define ningún nombre, se retorna `default` o, si no se proporciona, se lanza una excepción `ValueError`.

`unicodedata.decimal(chr[, default])`

Retorna el valor decimal asignado al carácter `chr` como un entero. Si no se define dicho valor, se retorna `default` o, si no se proporciona, se lanza una excepción `ValueError`.

`unicodedata.digit(chr[, default])`

Retorna el valor del dígito asignado al carácter `chr` como un entero. Si no se define dicho valor, se retorna `default` o, si no se proporciona, se genera una excepción `ValueError`.

`unicodedata.numeric(chr[, default])`

Retorna el valor numérico asignado al carácter `chr` como un flotante. Si no se define dicho valor, se retorna `default` o, si no se proporciona, se lanza una excepción `ValueError`.

`unicodedata.category(chr)`

Retorna la categoría general asignada al carácter `chr` como una cadena de caracteres.

`unicodedata.bidirectional(chr)`

Retorna la clase bidireccional asignada al carácter `chr` como una cadena de caracteres. Si no se define tal valor, se retorna una cadena de caracteres vacía.

`unicodedata.combining(chr)`

Retorna la clase de combinación canónica asignada al carácter `chr` como un entero. Retorna 0 si no se define ninguna clase de combinación.

`unicodedata.east_asian_width(chr)`

Retorna el ancho asignado al carácter `chr` para el este asiático como una cadena de caracteres.

`unicodedata.mirrored(chr)`

Retorna la propiedad reflejada asignada al carácter `chr` como un entero. Retorna 1 si el carácter se ha identificado como un carácter «reflejado» en texto bidireccional y 0 en caso contrario.

`unicodedata.decomposition(chr)`

Retorna el mapeo de descomposición de caracteres asignado al carácter `chr` como una cadena de caracteres. Se retorna una cadena de caracteres vacía en caso de que no se defina tal mapeo.

`unicodedata.normalize(form, unistr)`

Retorna la forma normalizada `form` para la cadena Unicode `unistr`. Los valores válidos para `form` son “NFC”, “NFKC”, “NFD” y “NFKD”.

El estándar Unicode define varias formas de normalización de una cadena Unicode, basándose en la definición de equivalencia canónica y equivalencia de compatibilidad. En Unicode, varios caracteres se pueden expresar de diversas formas. Por ejemplo, el carácter U+00C7 (LETRA C LATINA MAYÚSCULA CON CEDILLA) también se puede expresar con la secuencia U+0043 (LETRA C LATINA MAYÚSCULA) U+0327 (CEDILLA COMBINABLE).

¹ <http://www.unicode.org/Public/12.1.0/ucd/NameAliases.txt>

² <http://www.unicode.org/Public/12.1.0/ucd/NamedSequences.txt>

Para cada carácter, hay dos formas normalizadas: la forma normal C y la forma normal D. La forma normal D (NFD) también se conoce como descomposición canónica y traduce cada carácter a su forma descompuesta. La forma normal C (NFC) primero aplica una descomposición canónica y luego vuelve a componer los caracteres combinados previamente.

Además de las dos anteriores, hay dos formas normalizadas adicionales basadas en la equivalencia de compatibilidad. En Unicode, se admiten ciertos caracteres que normalmente se unificarán con otros caracteres. Por ejemplo, U+2160 (NUMERAL ROMANO UNO) es realmente lo mismo que U+0049 (LETRA LATINA MAYÚSCULA I). Sin embargo, esta característica está soportada por Unicode para ser compatible con los conjuntos de caracteres existentes (por ejemplo, gb2312).

La forma normalizada KD (NFKD) aplicará la descomposición de compatibilidad, es decir, reemplazará todos los caracteres de compatibilidad con sus equivalentes. La forma normalizada KC (NFKC) primero aplica la descomposición de compatibilidad, seguida de la composición canónica.

Incluso si dos cadenas Unicode están normalizadas y parecen iguales para un lector humano, si una tiene caracteres combinados y la otra no, es posible que no se comparen como iguales.

`unicodedata.is_normalized(form, unistr)`

Retorna si la cadena Unicode *unistr* está en la forma normalizada *form*. Los valores válidos para *form* son “NFC”, “NFKC”, “NFD” y “NFKD”.

Nuevo en la versión 3.8.

Además, el módulo expone las siguientes constantes:

`unicodedata.unidata_version`

La versión de la base de datos Unicode usada en este módulo.

`unicodedata.ucd_3_2_0`

Este es un objeto que tiene los mismos métodos que el módulo completo, pero usa la versión 3.2 de la base de datos Unicode en su lugar. Es útil para aplicaciones que requieren esta versión específica de la base de datos Unicode (como IDNA).

Ejemplos:

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
>>> unicodedata.name('/')
'SOLIDUS'
>>> unicodedata.decimal('9')
9
>>> unicodedata.decimal('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category('A') # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional('\u0660') # 'A'rabic, 'N'umber
'AN'
```

Notas al pie

6.6 stringprep — Preparación de cadenas de Internet

Código fuente: [Lib/stringprep.py](#)

Cuando se quiere identificar cosas (como nombres de host) en internet, generalmente es necesario comparar tales identificaciones para «igualdad». La manera en la que esta comparación se ejecuta dependerá del dominio de la aplicación, ej. si tiene o no tiene que distinguir entre mayúsculas y minúsculas. Además, en algunos casos será necesario

restringir las posibles identificaciones, de tal manera que solo se permitan identificadores de caracteres que se puedan «imprimir».

RFC 3454 define el proceso para la «preparación» de cadenas Unicode para protocolos de internet. Antes de pasar cadenas a un cable, se procesan con el proceso de preparación, después del cual tienen una forma normalizada. El RFC define un conjunto de tablas, que pueden ser combinadas en perfiles. Cada perfil debe definir qué tablas usa, y que partes opcionales del proceso `stringprep` son parte del perfil. Un ejemplo de perfil de `stringprep` es `nameprep`, que se usa para nombres de dominios internacionalizados.

El módulo `stringprep` solo expone las tablas de **RFC 3454**. Como estas tablas serían muy grandes para representarlas como diccionarios o listas, el módulo usa la base de datos de los caracteres de Unicode de manera interna. El código fuente del módulo en sí ha sido generado usando la herramienta `mkstringprep.py`

Como resultado, estas tablas son presentadas como funciones, no como estructuras de datos. Hay dos tipos de tablas en el RFC: conjuntos y mapeos. Para un conjunto, `stringprep` proporciona una «función característica», es decir, la función retorna `True` si el parámetro es parte del conjunto. Para los mapas, proporciona una función de mapeado: dada una clave, retorna el valor asociado. Abajo se encuentra una lista con todas las funciones disponibles para este módulo.

`stringprep.in_table_a1 (code)`

Determina si `code` está en la tablaA.1 (Puntos de Código sin asignar en Unicode 3.2).

`stringprep.in_table_b1 (code)`

Determina si `code` está en la tablaB.1 (Generalmente mapeado a nada).

`stringprep.map_table_b2 (code)`

Retorna el valor mapeado para `code` de acuerdo a la tablaB.2 (Mapeo para *case-folding* usado con NFKC).

`stringprep.map_table_b3 (code)`

Retorna el valor mapeado para `code` de acuerdo a tablaB.3 (Mapeo para *case-folding* usado sin normalización).

`stringprep.in_table_c11 (code)`

Determina si `code` está en la tablaC.1.1 (Caracteres de espacio ASCII).

`stringprep.in_table_c12 (code)`

Determina si `code` está en la tablaC.1.2 (Caracteres de espacio no-ASCII).

`stringprep.in_table_c11_c12 (code)`

Determina si `code` está en la tablaC.1 (Caracteres de espacio, unión de C.1.1 y C.1.2).

`stringprep.in_table_c21 (code)`

Determina si `code` está en la tablaC.2.1 (Caracteres de control ASCII).

`stringprep.in_table_c22 (code)`

Determina si `code` está en la tablaC.2.2 (Caracteres de control no ASCII).

`stringprep.in_table_c21_c22 (code)`

Determina si `code` está en la tablaC.2 (Caracteres de control, unión de C.2.1 y C.2.2).

`stringprep.in_table_c3 (code)`

Determina si `code` está en la tablaC.3 (Uso privado).

`stringprep.in_table_c4 (code)`

Determina si `code` está en la tablaC.4 (Puntos de código sin caracteres)

`stringprep.in_table_c5 (code)`

Determina si `code` está en la tablaC.5 (Códigos surrogados).

`stringprep.in_table_c6 (code)`

Determina si `code` está en la tablaC.6 (Inadecuado para texto plano).

`stringprep.in_table_c7 (code)`

Determina si `code` está en la tablaC.7 (Inadecuado para la representación canónica).

`stringprep.in_table_c8 (code)`

Determina si `code` está en la tablaC.8 (Cambia las propiedades de apariencia o están obsoletas).

`stringprep.in_table_c9 (code)`

Determina si *code* está en la tablaC.9 (Caracteres de etiquetado).

`stringprep.in_table_d1 (code)`

Determina si *code* está en la tablaD.1 (Caracteres con propiedad bidireccional «R» o «AL»).

`stringprep.in_table_d2 (code)`

Determina si *code* está en la tablaD.2 (Caracteres con propiedad bidireccional «L»).

6.7 readline — Interfaz readline de GNU

El módulo `readline` define una serie de funciones para facilitar la finalización y lectura/escritura de archivos de historial desde el intérprete de Python. Este módulo se puede usar directamente o mediante el módulo `rlcompleter`, que administra la finalización de identificadores de Python en la solicitud interactiva. Los ajustes realizados con este módulo afectan el comportamiento tanto del aviso interactivo del intérprete como de los avisos ofrecidos por la función incorporada `input()`.

Las combinaciones de teclas de Readline se pueden configurar mediante un archivo de inicialización, generalmente `.inputrc` en su directorio de inicio. Consulte [Readline Init File](#) en el manual de GNU Readline para obtener información sobre el formato y las construcciones permitidas de ese archivo, y las capacidades de la biblioteca Readline en general.

Nota: La API de la biblioteca utilizada por Readline puede implementarse mediante la biblioteca `libedit` en lugar de readline de GNU. En macOS, el módulo `readline` detecta qué biblioteca se está utilizando en tiempo de ejecución.

El archivo de configuración para `libedit` es diferente del readline de GNU. Si carga cadenas de caracteres de configuración mediante programación, puede verificar el texto «libedit» en `readline.__doc__` para diferenciar entre readline de GNU y `libedit`.

Si usa emulación readline `editline/libedit` en macOS, el archivo de inicialización ubicado en su directorio de inicio se llama `.editrc`. Por ejemplo, el siguiente contenido en `~/ .editrc` activa atajos del teclado de *vi* y completado con TAB:

```
python:bind -v
python:bind ^I rl_complete
```

6.7.1 Archivo de inicio

Las siguientes funciones se relacionan con el archivo de inicio y la configuración del usuario:

`readline.parse_and_bind (string)`

Ejecuta la línea de inicio proporcionada en el argumento *string*. Esto llama a la función `rl_parse_and_bind()` de la biblioteca subyacente.

`readline.read_init_file ([filename])`

Ejecuta un archivo de inicialización readline. El nombre de archivo predeterminado es el último nombre de archivo utilizado. Esto llama a la función `rl_read_init_file()` de la biblioteca subyacente.

6.7.2 Búfer de línea

Las siguientes funciones operan en el búfer de línea:

`readline.get_line_buffer()`
Retorna el contenido actual del búfer de línea (`rl_line_buffer` en la biblioteca subyacente).

`readline.insert_text(string)`
Inserta texto en el búfer de línea en la posición del cursor. Esto llama a la función `rl_insert_text()` de la biblioteca subyacente, pero ignora el valor de retorno.

`readline.redisplay()`
Cambia lo que se muestra en la pantalla para reflejar el contenido actual del búfer de línea. Esto llama a la función `rl_redisplay()` de la biblioteca subyacente.

6.7.3 Archivo de historial

Las siguientes funciones operan en un archivo de historial:

`readline.read_history_file([filename])`
Carga un archivo de historial `readline` y lo adjunta a la lista de historial. El nombre de archivo predeterminado es `~/.history`. Esto llama a la función `read_history()` de la biblioteca subyacente.

`readline.write_history_file([filename])`
Guarda la lista de historial en un archivo de historial `readline`, sobrescribiendo cualquier archivo existente. El nombre de archivo predeterminado es `~/.history`. Esto llama a la función `write_history()` de la biblioteca subyacente.

`readline.append_history_file(nelements[, filename])`
Agrega los últimos *nelements* del historial a un archivo. El nombre de archivo predeterminado es `~/.history`. El archivo ya debe existir. Esto llama a la función `append_history()` de la biblioteca subyacente. Esta función solo existe si Python se compiló para una versión de la biblioteca que lo admita.

Nuevo en la versión 3.5.

`readline.get_history_length()`
`readline.set_history_length(length)`
Establece o retorna el número deseado de líneas para guardar en el archivo de historial. La función `write_history_file()` usa este valor para truncar el archivo de historial, llamando a la función `history_truncate_file()` en la biblioteca subyacente. Los valores negativos implican un tamaño de archivo de historial ilimitado.

6.7.4 Lista del historial

Las siguientes funciones operan en una lista de historial global:

`readline.clear_history()`
Borra el historial actual. Esto llama a la función `clear_history()` en la biblioteca subyacente. La función de Python solo existe si Python se compiló para una versión de la biblioteca que lo admita.

`readline.get_current_history_length()`
Retorna el número de elementos actuales en el historial. (Esto es diferente de la función `get_history_length()`, que retorna el número máximo de líneas que se escribirán en un archivo de historial).

`readline.get_history_item(index)`
Retorna el contenido actual de historial en *index*. El índice comienza en 1. Esto llama a la función `history_get()` en la biblioteca subyacente.

`readline.remove_history_item(pos)`
Elimina el objeto del historial definido por su posición del historial. La posición comienza en cero. Esto llama a la función `remove_history()` en la biblioteca subyacente.

`readline.replace_history_item(pos, line)`

Reemplaza el elemento del historial especificado por su posición con *line*. La posición comienza en cero. Esto llama a la función `replace_history_entry()` en la biblioteca subyacente.

`readline.add_history(line)`

Agrega *line* al búfer de historial, como si fuera la última línea escrita. Esto llama a la función `add_history()` en la biblioteca subyacente.

`readline.set_auto_history(enabled)`

Habilita o deshabilita las llamadas automáticas a la función `add_history()` al leer la entrada a través de `readline`. El argumento *enabled* debe ser un valor booleano que cuando es verdadero, habilita el historial automático, y que cuando es falso, desactiva el historial automático.

Nuevo en la versión 3.6.

CPython implementation detail: Auto history is enabled by default, and changes to this do not persist across multiple sessions.

6.7.5 Ganchos (*hooks*) de inicialización

`readline.set_startup_hook([function])`

Establece o elimina la función invocada por la función de retorno `rl_startup_hook` de la biblioteca subyacente. Si se especifica *function*, se utilizará como la nueva función de enlace; Si se omite o es `None`, se elimina cualquier función ya instalada. La función de devolución de llamada se llama sin argumentos justo antes de que `readline` muestre el primer símbolo del sistema.

`readline.set_pre_input_hook([function])`

Establece o elimina la función invocada por la función de retorno `rl_pre_input_hook` de la biblioteca subyacente. Si se especifica *function*, se utilizará como la nueva función de devolución de llamada; Si se omite o es `None`, se elimina cualquier función ya instalada. La función de devolución de llamada se llama sin argumentos después de que se haya visualizado el primer símbolo del sistema y justo antes de que `readline` comience a leer los caracteres ingresados. Esta función solo existe si Python se ha compilado para una versión de la biblioteca que lo admite.

6.7.6 Terminación

Las siguientes funciones se relacionan con la implementación de una función de finalización de palabra personalizada. Esto típicamente es operado por la tecla *Tab* y puede sugerir y completar automáticamente una palabra que se está escribiendo. Por defecto, `Readline` está configurado para ser utilizado por `rlcompleter` para completar los identificadores de Python para el intérprete interactivo. Si el módulo `readline` se va a utilizar con una terminación específica, se debe definir un conjunto de palabras delimitadoras.

`readline.set_completer([function])`

Establece o elimina la función de finalización. Si se especifica *function*, se usará como la nueva función de finalización; Si se omite o es `None`, se elimina cualquier función de finalización ya instalada. La función completa se llama como `function(text, state)`, para *state* en 0, 1, 2, ..., hasta que retorna un valor que no es una cadena de caracteres. Debería retornar las siguientes terminaciones posibles comenzando con *text*.

La función de finalización instalada es invocada por la función de retorno `entry_func` que se pasa a `rl_completion_matches()` en la biblioteca subyacente. La cadena de texto va desde el primer parámetro a la función de retorno `rl_attempted_completion_function` de la biblioteca subyacente.

`readline.get_completer()`

Obtiene la función de finalización o `None` si no se ha definido ninguna función de finalización.

`readline.get_completion_type()`

Obtiene el tipo de finalización que se está intentando. Esto retorna la variable `rl_completion_type` en la biblioteca subyacente como un entero.

`readline.get_begidx()`

`readline.get_endidx()`

Obtiene el índice inicial o final de un contexto de finalización. Estos índices son los argumentos *start* y *end* pasados a la función de retorno `rl_attempted_completion_function` de la biblioteca subyacente.

`readline.set_completer_delims(string)`

`readline.get_completer_delims()`

Define o recupera palabras delimitadoras para completar. Estos determinan el comienzo de la palabra que se considerará para su finalización (el contexto de finalización). Estas funciones acceden a la variable `rl_completer_word_break_characters` desde la biblioteca subyacente.

`readline.set_completion_display_matches_hook([function])`

Establece o elimina la función de visualización de finalización. Si se especifica *function*, se utilizará como una nueva función de visualización de finalización; si se omite o es `None`, se elimina cualquier función de finalización ya instalada. Esto establece o elimina la función de retorno `rl_completion_display_matches_hook` de la biblioteca subyacente. La función de visualización de finalización se llama tal como la `function(substitution, [matches], longest_match_length)` solo una vez cuando se muestran las coincidencias.

6.7.7 Ejemplo

El siguiente ejemplo muestra cómo usar las funciones de lectura y escritura del historial del módulo `readline` para cargar o guardar automáticamente un archivo de historial llamado `.python_history` desde el directorio de inicio del usuario. El siguiente código normalmente debe ejecutarse automáticamente durante una sesión interactiva desde el archivo de usuario `PYTHONSTARTUP`.

```
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

Este código se ejecuta automáticamente cuando Python se ejecuta en modo interactivo (ver [Configuración de Readline](#)).

El siguiente ejemplo logra el mismo objetivo pero administra sesiones interactivas concurrentes, agregando solo el nuevo historial.

```
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history")

try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
```

(continué en la próxima página)

(proviene de la página anterior)

```
readline.append_history_file(new_h_len - prev_h_len, histfile)
atexit.register(save, h_len, histfile)
```

El siguiente ejemplo amplía la clase `code.InteractiveConsole` para administrar el guardado/restauración del historial.

```
import atexit
import code
import os
import readline

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 FileNotFoundError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

6.8 rlcompleter — Función de completado para GNU readline

Código fuente: [Lib/rlcompleter.py](#)

El módulo `rlcompleter` define una función de completado adecuada para el módulo `readline` completando los identificadores y las palabras clave de Python válidas.

Cuando este módulo es importado en una plataforma Unix con el módulo `readline` disponible, una instancia de la clase `Completer` es automáticamente creada y su método `complete()` es fijado como el método de completado de `readline`.

Ejemplo:

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
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.
```

El módulo `rlcompleter` está diseñado para usarse con el modo interactivo de Python. A menos que Python sea ejecutado con la opción `-S`, el módulo es automáticamente importado y configurado (ver [Configuración de Readline](#)).

En plataformas sin `readline`, la clase `Completer` definida por este módulo puede ser usada igualmente para fines personalizados.

6.8.1 Objetos de Completado

Los objetos de completado tienen el siguiente método:

`Completer.complete(text, state)`

Retorna el completado nº *state* para *text*.

Si es invocado para *text* que no incluye un caracter de punto (`'.'`), este completará con nombres actualmente definidos en `__main__`, `builtins` y las palabras clave (tal y como están definidas en el módulo `keyword`).

Si es invocado para un nombre con punto, este tratará de evaluar cualquier cosa sin efectos secundarios obvios (las funciones no serán evaluadas, pero puede generar invocaciones a `__getattr__()` hasta la última parte, y encontrar coincidencias para el resto mediante la función `dir()`, Cualquier excepción ocurrida durante la evaluación de la expresión es cazada, silenciada y se retorna `None`.

Servicios de datos binarios

Los módulos descritos en este capítulo proporcionan algunas operaciones básicas de servicios para la manipulación de datos binarios. Otras operaciones sobre datos binarios específicamente relacionadas con formatos de archivo y protocolos de red están descritas en las secciones relevantes.

Algunas bibliotecas descritas bajo *Servicios de procesamiento de texto* también funcionan o bien sobre formatos binarios compatibles con ASCII (por ejemplo *re*), o bien sobre todos los datos binarios (por ejemplo *difflib*).

Adicionalmente, véase la documentación para los tipos de datos binarios incorporados en Python en *Tipos de secuencias binarias* — *bytes*, *bytearray* y *memoryview*.

7.1 struct — Interpreta bytes como paquetes de datos binarios

Código fuente: [Lib/struct.py](#)

Este módulo realiza conversiones entre valores de Python y estructuras C representadas como objetos *bytes* de Python. Se puede utilizar para el tratamiento de datos binarios almacenados en archivos o desde conexiones de red, entre otras fuentes. Utiliza *Cadenas de Formato* como descripciones compactas del diseño de las estructuras C y la conversión prevista a/desde valores de Python.

Nota: Por defecto, el resultado de empaquetar una estructura C determinada incluye bytes de relleno para mantener la alineación adecuada para los tipos correspondientes en C; del mismo modo, la alineación se tiene en cuenta al desempaquetar. Este comportamiento se elige para que los bytes de una estructura empaquetada se correspondan exactamente con el diseño en memoria de la estructura C correspondiente. Para tratar formatos de datos que sean independientes de la plataforma u omitir bytes de relleno implícitos, utiliza el tamaño y la alineación estándar en lugar del nativo: ver *Orden de Bytes*, *Tamaño y Alineación* para obtener más información.

Varias funciones *struct* (y métodos de *Struct*) toman un argumento *buffer*. Esto hace referencia a los objetos que implementan *bufferobjects* y proporcionan un búfer de lectura o de lectura/escritura. Los tipos más comunes utilizados para ese propósito son *bytes* y *bytearray*, pero muchos otros tipos que se pueden ver como una lista de bytes implementan el protocolo de búfer, para que se puedan leer/rellenar sin necesidad de copiar a partir de un objeto *bytes*.

7.1.1 Funciones y Excepciones

El módulo define la siguiente excepción y funciones:

exception `struct.error`

Excepción lanzada en varias ocasiones; el argumento es una *string* que describe lo que está mal.

`struct.pack(format, v1, v2, ...)`

Retorna un objeto bytes que contiene los valores *v1*, *v2*, ... empaquetado de acuerdo con la cadena de formato *format*. Los argumentos deben coincidir exactamente con los valores requeridos por el formato.

`struct.pack_into(format, buffer, offset, v1, v2, ...)`

Empaqueta los valores *v1*, *v2*, ... de acuerdo con la cadena de formato *format* y escribe los bytes empaquetados en el búfer de escritura *buffer* comenzando en la posición *offset*. Nota: *offset* es un argumento obligatorio.

`struct.unpack(format, buffer)`

Desempaqueta del búfer *buffer* (normalmente empaquetado por `pack(format, ...)`) según la cadena de formato *format*. El resultado es una tupla incluso si contiene un solo elemento. El tamaño del búfer en bytes debe coincidir con el tamaño requerido por el formato, como se refleja en `calcsize()`.

`struct.unpack_from(format, buffer, offset=0)`

Desempaqueta del *buffer* comenzando en la posición *offset*, según la cadena de formato *format*. El resultado es una tupla incluso si contiene un solo elemento. El tamaño del búfer en bytes, comenzando en la posición *offset*, debe tener al menos el tamaño requerido por el formato, como se refleja en `calcsize()`.

`struct.iter_unpack(format, buffer)`

Desempaqueta de manera iterativa desde el búfer *buffer* según la cadena de formato *format*. Esta función retorna un iterador que leerá fragmentos de igual tamaño desde el búfer hasta que se haya consumido todo su contenido. El tamaño del búfer en bytes debe ser un múltiplo del tamaño requerido por el formato, como se refleja en `calcsize()`.

Cada iteración produce una tupla según lo especificado por la cadena de formato.

Nuevo en la versión 3.4.

`struct.calcsize(format)`

Retorna el tamaño de la estructura (y, por lo tanto, del objeto bytes generado por `pack(format, ...)`) correspondiente a la cadena de formato *format*.

7.1.2 Cadenas de Formato

Las cadenas de formato son el mecanismo utilizado para especificar el diseño esperado al empaquetar y desempaquetar datos. Se crean a partir de *Formato de caracteres*, que especifican el tipo de datos que se empaquetan/desempaquetan. Además, hay caracteres especiales para controlar *Orden de Bytes*, *Tamaño* y *Alineación*.

Orden de Bytes, Tamaño y Alineación

Por defecto, los tipos C se representan en el formato nativo y el orden de bytes de la máquina, y se alinean correctamente omitiendo bytes de relleno si es necesario (según las reglas utilizadas por el compilador de C).

Como alternativa, el primer carácter de la cadena de formato se puede utilizar para indicar el orden de bytes, el tamaño y la alineación de los datos empaquetados, según la tabla siguiente:

Caracter	Orden de Bytes	Tamaño	Alineamiento
@	nativo	nativo	nativo
=	nativo	standard	none
<	little-endian	standard	none
>	big-endian	standard	none
!	red (= big-endian)	standard	none

Si el primer carácter no es uno de estos, se asume '@'.

El orden de bytes nativo es big-endian o little-endian, dependiendo del sistema host. Por ejemplo, Intel x86 y AMD64 (x86-64) son little-endian; Motorola 68000 y PowerPC G5 son big-endian; ARM e Intel *Itanium* tienen la propiedad de trabajar con ambos formatos (middle-endian). Utiliza `sys.byteorder` para comprobar la *endianness* («extremidad») de su sistema.

El tamaño y la alineación nativos se determinan mediante la expresión `sizeof` del compilador de C. Esto siempre se combina con el orden de bytes nativo.

El tamaño estándar depende únicamente del carácter de formato; ver la tabla en la sección [Formato de caracteres](#).

Nótese la diferencia entre `'@'` y `'='`: ambos utilizan el orden de bytes nativo, pero el tamaño y la alineación de este último está estandarizado.

The form `'!'` represents the network byte order which is always big-endian as defined in [IETF RFC 1700](#).

No hay manera de indicar el orden de bytes no nativo (forzar el intercambio de bytes); utiliza la elección adecuada de `'<'` o `'>'`.

Notas:

- (1) El relleno solo se agrega automáticamente entre los miembros sucesivos de la estructura. No se agrega ningún relleno al principio o al final de la estructura codificada.
- (2) No se añade ningún relleno cuando se utiliza el tamaño y la alineación no nativos, por ejemplo, con `"<"`, `">"`, `"="` y `"!"`.
- (3) Para alinear el final de una estructura con el requisito de alineación de un tipo determinado, termina el formato con el código de ese tipo con un dos ceros. Véase [Ejemplos](#).

Formato de caracteres

Los caracteres de formato tienen el siguiente significado; la conversión entre los valores C y Python debe ser obvia dados sus tipos. La columna “Tamaño estándar” hace referencia al tamaño del valor empaquetado en bytes cuando se utiliza el tamaño estándar; es decir, cuando la cadena de formato comienza con uno de `'<'`, `'>'`, `'!'` o `'='`. Cuando se utiliza el tamaño nativo, el tamaño del valor empaquetado depende de la plataforma.

Formato	Tipo C	Tipo Python	Tamaño estándar	Notas
x	byte de relleno	sin valor		
c	char	bytes de longitud 1	1	
b	signed char	integer	1	(1), (2)
B	unsigned char	integer	1	(2)
(2)	_Bool	bool	1	(1)
h	short	integer	2	(2)
H	unsigned short	integer	2	(2)
i	int	integer	4	(2)
I	unsigned int	integer	4	(2)
l	long	integer	4	(2)
L	unsigned long	integer	4	(2)
q	long long	integer	8	(2)
Q	unsigned long long	integer	8	(2)
n	ssize_t	integer		(3)
N	size_t	integer		(3)
e	(6)	float	2	(4)
f	float	float	4	(4)
d	double	float	8	(4)
s	char[]	bytes		
p	char[]	bytes		
P	void *	integer		(5)

Distinto en la versión 3.3: Soporte añadido para los formatos `'n'` y `'N'`.

Distinto en la versión 3.6: Soporte añadido para el formato 'e'.

Notas:

- (1) El código de conversión '?' corresponde al tipo `_Bool` definido por C99. Si este tipo no está disponible, se simula mediante un `char`. En el modo estándar, siempre se representa mediante un `byte`.
- (2) Al intentar empaquetar un no entero mediante cualquiera de los códigos de conversión de enteros, si el no entero tiene un método `__index__()`, se llama a ese método para convertir el argumento en un entero antes de empaquetar.

Distinto en la versión 3.2: El uso del método `__index__()` para los no enteros es nuevo en 3.2.

- (3) Los códigos de conversión 'n' y 'N' solo están disponibles para el tamaño nativo (seleccionado como predeterminado o con el carácter de orden de bytes '@'). Para el tamaño estándar, puedes usar cualquiera de los otros formatos enteros que se ajusten a tu aplicación.
- (4) Para los códigos de conversión 'f', 'd' y 'e', la representación empaquetada utiliza el formato IEEE 754 `binary32`, `binary64` o `binary16` (para 'f', 'd' o 'e' respectivamente), independientemente del formato de punto flotante utilizado por la plataforma.
- (5) El carácter de formato 'P' solo está disponible para el orden de bytes nativo (seleccionado como predeterminado o con el carácter de orden de bytes '@'). El carácter de orden de bytes '=' elige utilizar el orden little- o big-endian basado en el sistema host. El módulo `struct` no interpreta esto como un orden nativo, por lo que el formato 'P' no está disponible.
- (6) El tipo IEEE 754 `binary16` «half precision» se introdujo en la revisión de 2008 del [IEEE 754 estándar](#). Tiene un bit de signo, un exponente de 5 bits y una precisión de 11 bits (con 10 bits almacenados explícitamente) y puede representar números entre aproximadamente $6.1e-05$ y $6.5e+04$ con total precisión. Este tipo no es ampliamente compatible con los compiladores de C: en un equipo típico, un `unsigned short` se puede usar para el almacenamiento, pero no para las operaciones matemáticas. Consulte la página de Wikipedia en el [formato de punto flotante de media precisión](#) para obtener más información.

Un carácter de formato puede ir precedido de un número de recuento que repite tantas veces el carácter. Por ejemplo, la cadena de formato '4h' significa exactamente lo mismo que 'hhhh'.

Se omiten los caracteres de espacio entre formatos; sin embargo, un recuento y su formato no deben contener espacios en blanco.

Para el carácter de formato 's', el recuento se interpreta como la longitud de los bytes, no un recuento de repetición como para los otros caracteres de formato; por ejemplo, '10s' significa una sola cadena de 10 bytes, mientras que '10c' significa 10 caracteres. Si no se da un recuento, el valor predeterminado es 1. Para el empaquetado, la cadena es truncada o rellenada con bytes nulos según corresponda para que se ajuste. Para desempaquetar, el objeto bytes resultante siempre tiene exactamente el número especificado de bytes. Como caso especial, '0s' significa una sola cadena vacía (mientras que '0c' significa 0 caracteres).

Al empaquetar un valor `x` utilizando uno de los formatos enteros ('b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q'), si `x` está fuera de un rango válido para ese formato, entonces se lanza la excepción `struct.error`.

Distinto en la versión 3.1: En 3.0, algunos de los formatos enteros ajustaban los valores fuera de rango y lanza `DeprecationWarning` en lugar de `struct.error`.

El carácter de formato 'p' codifica una «cadena de Pascal», lo que significa una cadena de longitud variable corta almacenada en un número *fijo de bytes*, dado por el recuento. El primer byte almacenado es el valor mínimo entre la longitud de la cadena o 255. A continuación se encuentran los bytes de la cadena. Si la cadena pasada a `pack()` es demasiado larga (más larga que la cuenta menos 1), solo se almacenan los bytes iniciales `count-1` de la cadena. Si la cadena es más corta que `count-1`, se rellena con bytes nulos para que se utilicen exactamente los bytes de recuento en total. Tenga en cuenta que para `unpack()`, el carácter de formato 'p' consume bytes `count`, pero que la cadena retornada nunca puede contener más de 255 bytes.

Para el carácter de formato '?', el valor retornado es `True` o `False`. Al empaquetar, se utiliza el valor verdadero del objeto del argumento. Se empaquetará 0 o 1 en la representación *bool* nativa o estándar, y cualquier valor distinto de cero será `True` al desempaquetar.

Ejemplos

Nota: Todos los ejemplos asumen un orden de bytes tamaño y alineación nativos, con una máquina big-endian.

Un ejemplo básico de empaquetado/desempaquetado de tres enteros:

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
b'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', b'\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

Los campos desempaquetados se pueden nombrar asignándolos a variables o ajustando el resultado en una tupla con nombre:

```
>>> record = b'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=b'raymond ', serialnum=4658, school=264, gradelevel=8)
```

El orden de los caracteres de formato puede tener un impacto en el tamaño ya que el relleno necesario para satisfacer los requisitos de alineación es diferente:

```
>>> pack('ci', b'*', 0x12131415)
b'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, b'*')
b'\x12\x13\x14\x15*'
>>> calcsize('ci')
8
>>> calcsize('ic')
5
```

El siguiente formato 'llh01' especifica dos bytes de relleno al final, suponiendo que los tipos *longs* están alineados en los límites de 4 bytes:

```
>>> pack('llh01', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

Esto solo funciona cuando el tamaño y la alineación nativos tienen efecto; el tamaño estándar y la alineación no imponen ninguna alineación.

Ver también:

Módulo `array` Almacenamiento binario empaquetado de datos homogéneos.

Módulo `xdrlib` Empaquetar y desempaquetar datos XDR.

7.1.3 Clases

El módulo `struct` también define el siguiente tipo:

class `struct.Struct` (*format*)

Retorna un nuevo objeto `Struct` que escribe y lee datos binarios según la cadena de formato *format*. Crear un objeto `Struct` una vez y llamar a sus métodos es más eficaz que llamar a las funciones `struct` con el mismo formato, ya que la cadena de formato solo se compila una vez en ese caso.

Nota: Las versiones compiladas de las cadenas de formato más recientes pasadas a `Struct` y las funciones de nivel de módulo se almacenan en caché, por lo que los programas que utilizan solo unas pocas cadenas de formato no necesitan preocuparse por volver a usar una sola instancia `Struct`.

Los objetos `Struct` compilados admiten los siguientes métodos y atributos:

pack (*v1*, *v2*, ...)

Idéntico a la función `pack()`, utilizando el formato compilado. (`len(result)` será igual a *size*.)

pack_into (*buffer*, *offset*, *v1*, *v2*, ...)

Idéntico a la función `pack_into()`, utilizando el formato compilado.

unpack (*buffer*)

Idéntico a la función `unpack()`, utilizando el formato compilado. El tamaño del búfer en bytes debe ser igual a *size*.

unpack_from (*buffer*, *offset*=0)

Idéntico a la función `unpack_from()`, utilizando el formato compilado. El tamaño del búfer en bytes, comenzando en la posición *offset*, debe ser al menos *size*.

iter_unpack (*buffer*)

Idéntico a la función `iter_unpack()`, utilizando el formato compilado. El tamaño del búfer en bytes debe ser un múltiplo de *size*.

Nuevo en la versión 3.4.

format

Cadena de formato utilizada para construir este objeto `Struct`.

Distinto en la versión 3.7: El tipo de cadena de formato es ahora `str` en lugar de `bytes`.

size

El tamaño calculado de la estructura (y, por lo tanto, del objeto bytes generado por el método `pack()`) correspondiente a *format*.

7.2 codecs — Registro de códec y clases base

Código fuente: `Lib/codecs.py`

Este módulo define clases base para códecs Python estándar (codificadores y decodificadores) y proporciona acceso al registro interno de códecs Python, que gestiona el proceso de búsqueda de códec y manejo de errores. La mayoría de los códecs estándar son *codificaciones de texto*, que codifican texto a bytes, pero también se proporcionan códecs que codifican texto a texto y bytes a bytes. Los códecs personalizados pueden codificar y decodificar entre tipos arbitrarios, pero algunas funciones del módulo están restringidas para usarse específicamente con *codificaciones de texto*, o con códecs que codifican para `bytes`.

El módulo define las siguientes funciones para codificar y decodificar con cualquier códec:

`codecs.encode` (*obj*, *encoding*='utf-8', *errors*='strict')

Codifica *obj* utilizando el códec registrado para *encoding*.

Se pueden dar *errors* para establecer el esquema de manejo de errores deseado. El controlador de errores predeterminado es `'estricto'`, lo que significa que los errores de codificación provocan `ValueError` (o una subclase más específica del códec, como `UnicodeEncodeError`). Consulte *Clases Base de Códec* para obtener más información sobre el manejo de errores de códec.

`codecs.decode(obj, encoding='utf-8', errors='strict')`

Decodifica *obj* utilizando el códec registrado para *encoding*.

Se pueden dar *errors* para establecer el esquema de manejo de errores deseado. El controlador de errores predeterminado es `'estricto'`, lo que significa que los errores de decodificación generan `ValueError` (o una subclase más específica de códec, como `UnicodeDecodeError`). Consulte *Clases Base de Códec* para obtener más información sobre el manejo de errores de códec.

Los detalles completos de cada códec también se pueden consultar directamente:

`codecs.lookup(encoding)`

Busca la información de códec en el registro de códec de Python y retorna un objeto `CodecInfo` como se define a continuación.

Las codificaciones se buscan primero en la memoria caché del registro. Si no se encuentran, se explora la lista de funciones de búsqueda registradas. Si no se encuentran objetos `CodecInfo`, se lanza un `LookupError`. De lo contrario, el objeto `CodecInfo` se almacena en la memoria caché y se retorna a quien llama.

class `codecs.CodecInfo` (*encode, decode, streamreader=None, streamwriter=None, incrementalencoder=None, incrementaldecoder=None, name=None*)

Detalles de códec al buscar el registro de códec. Los argumentos del constructor se almacenan en atributos del mismo nombre:

name

El nombre de la codificación.

encode

decode

Las funciones de codificación y decodificación sin estado. Deben ser funciones o métodos que tengan la misma interfaz que los métodos `encode()` y `decode()` de instancias de `Codec` (ver *Codec Interface*). Se espera que las funciones o métodos funcionen en modo sin estado.

incrementalencoder

incrementaldecoder

Clases de codificación y decodificación incremental o funciones de fábrica. Deben proporcionar la interfaz definida por las clases base `IncrementalEncoder` y `IncrementalDecoder`, respectivamente. Los códecs incrementales pueden mantener el estado.

streamwriter

streamreader

Las clases *stream*, tanto *writer* como *reader* o funciones de fábrica. Estos tienen que proporcionar la interfaz definida por las clases base `StreamWriter` y `StreamReader`, respectivamente. Los códecs de flujo pueden mantener el estado.

Para simplificar el acceso a los diversos componentes de códec, el módulo proporciona estas funciones adicionales que utilizan `lookup()` para la búsqueda de códec:

`codecs.getencoder(encoding)`

Busca el códec para la codificación dada y retorna su función de codificador.

Lanza un `LookupError` en caso de que no se encuentre la codificación.

`codecs.getdecoder(encoding)`

Busca el códec para la codificación dada y retorna su función de decodificador.

Lanza un `LookupError` en caso de que no se encuentre la codificación.

`codecs.getincrementalencoder(encoding)`

Busca el códec para la codificación dada y retorna su clase de codificador incremental o función de fábrica.

Lanza un `LookupError` en caso de que no se encuentre la codificación o el códec no admita un codificador incremental.

`codecs.getincrementaldecoder(encoding)`

Busca el códec para la codificación dada y retorna su clase de decodificador incremental o función de fábrica.

Lanza un `LookupError` en caso de que no se encuentre la codificación o el códec no admita un decodificador incremental.

`codecs.getreader(encoding)`

Busca el códec para la codificación dada y retorna su clase `StreamReader` o función de fábrica.

Lanza un `LookupError` en caso de que no se encuentre la codificación.

`codecs.getwriter(encoding)`

Busca el códec para la codificación dada y retorna su clase `StreamWriter` o función de fábrica.

Lanza un `LookupError` en caso de que no se encuentre la codificación.

Los códecs personalizados se ponen a disposición registrando una función de búsqueda de códecs adecuada:

`codecs.register(search_function)`

Registra una función de búsqueda de códec. Se espera que las funciones de búsqueda tomen un argumento, que sea el nombre de codificación en minúsculas, y retornan un objeto `CodecInfo`. En caso de que una función de búsqueda no pueda encontrar una codificación dada, debería retornar `None`.

Nota: El registro de la función de búsqueda no es reversible actualmente, lo que puede causar problemas en algunos casos, como pruebas unitarias o recarga de módulos.

Mientras que la función incorporada `open()` y el módulo asociado `io` son el enfoque recomendado para trabajar con archivos de texto codificados, este módulo proporciona funciones y clases de utilidad adicionales que permiten el uso de una gama más amplia de códecs cuando se trabaja con archivos binarios:

`codecs.open(filename, mode='r', encoding=None, errors='strict', buffering=-1)`

Abre un archivo codificado utilizando el `mode` dado y retorna una instancia de `StreamReaderWriter`, proporcionando codificación/decodificación transparente. El modo de archivo predeterminado es `'r'`, que significa abrir el archivo en modo de lectura.

Nota: Los archivos codificados subyacentes siempre se abren en modo binario. No se realiza la conversión automática de `'\n'` en lectura y escritura. El argumento `mode` puede ser cualquier modo binario aceptable para la función incorporada `open()`; la `'b'` se agrega automáticamente.

`encoding` especifica la codificación que se utilizará para el archivo. Se permite cualquier codificación que codifique y decodifique desde bytes, y los tipos de datos admitidos por los métodos de archivo dependen del códec utilizado.

`errors` puede darse para definir el manejo de errores. El valor predeterminado es `'estricto'`, lo que hace que se genere un `ValueError` en caso de que ocurra un error de codificación.

`buffering` tiene el mismo significado que para la función incorporada `open()`. Su valor predeterminado es `-1`, lo que significa que se utilizará el tamaño predeterminado del búfer.

`codecs.EncodedFile(file, data_encoding, file_encoding=None, errors='strict')`

Retorna una instancia de `StreamRecoder`, una versión envuelta de `file` que proporciona transcodificación transparente. El archivo original se cierra cuando se cierra la versión empaquetada.

Los datos escritos en el archivo empaquetado se decodifican de acuerdo con la `data_encoding` dada y luego se escriben en el archivo original como bytes usando `file_encoding`. Los bytes leídos del archivo original se decodifican según `file_encoding`, y el resultado se codifica utilizando `data_encoding`.

Si no se proporciona `file_encoding`, el valor predeterminado es `data_encoding`.

Se pueden dar `errors` para definir el manejo de errores. Su valor predeterminado es `'estricto'`, lo que hace que se genere `ValueError` en caso de que ocurra un error de codificación.

`codecs.iterencode(iterator, encoding, errors='strict', **kwargs)`

Utiliza un codificador incremental para codificar iterativamente la entrada proporcionada por *iterator*. Esta función es un *generator*. El argumento *errors* (así como cualquier otro argumento de palabra clave) se pasa al codificador incremental.

Esta función requiere que el códec acepte texto en objetos *str* para codificar. Por lo tanto, no admite codificadores de bytes a bytes, como `base64_codec`.

`codecs.iterdecode(iterator, encoding, errors='strict', **kwargs)`

Utiliza un decodificador incremental para decodificar iterativamente la entrada proporcionada por *iterator*. Esta función es un *generator*. El argumento *errors* (así como cualquier otro argumento de palabra clave) se pasa al decodificador incremental.

Esta función requiere que el códec acepte objetos *bytes* para decodificar. Por lo tanto, no admite codificadores de texto a texto como `rot_13`, aunque `rot_13` puede usarse de manera equivalente con `iterencode()`.

El módulo también proporciona las siguientes constantes que son útiles para leer y escribir en archivos dependientes de la plataforma:

```
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
```

Estas constantes definen varias secuencias de bytes, que son marcas de orden de bytes Unicode (BOM) para varias codificaciones. Se utilizan en flujos de datos UTF-16 y UTF-32 para indicar el orden de bytes utilizado, y en UTF-8 como firma Unicode. `BOM_UTF16` es `BOM_UTF16_BE` o `BOM_UTF16_LE` dependiendo del orden de bytes nativo de la plataforma, `BOM` es un alias para `BOM_UTF16`, `BOM_LE` para `BOM_UTF16_LE` y `BOM_BE` para `BOM_UTF16_BE`. Los otros representan la lista de materiales en las codificaciones UTF-8 y UTF-32.

7.2.1 Clases Base de Códec

El módulo `codecs` define un conjunto de clases base que definen las interfaces para trabajar con objetos de códec, y también puede usarse como base para implementaciones de códec personalizadas.

Cada códec tiene que definir cuatro interfaces para que pueda usarse como códec en Python: codificador sin estado, decodificador sin estado, lector de flujo y escritor de flujo. El lector de flujo y los escritores suelen reutilizar el codificador/decodificador sin estado para implementar los protocolos de archivo. Los autores de códec también necesitan definir cómo manejará los errores de codificación y decodificación.

Manejadores de errores

Para simplificar y estandarizar el manejo de errores, los códec pueden implementar diferentes esquemas de manejo de errores al aceptar el argumento de cadena *errors*. Los siguientes valores de cadena están definidos e implementados por todos los códec Python estándar:

Valor	Significado
'strict'	Lanza <code>UnicodeError</code> (o una subclase); Este es el valor predeterminado. Implementado en <code>strictly_errors()</code> .
'ignore'	Ignore los datos mal formados y continúe sin previo aviso. Implementado en <code>ignore_errors()</code> .

Los siguientes controladores de errores solo son aplicables a *codificaciones de texto*:

Valor	Significado
'replace'	Reemplaza con un marcador de reemplazo adecuado; Python utilizará el CARACTER DE REEMPLAZO U+FFFD oficial para los códecs integrados en la decodificación, y “?” en la codificación. Implementado en <code>replace_errors()</code> .
'xmlcharrefreplace'	Reemplaza con la referencia de caracteres XML apropiada (solo para codificación). Implementado en <code>xmlcharrefreplace_errors()</code> .
'backslashreplace'	Reemplaza con secuencias de escape con barra invertida. Implementado en <code>backslashreplace_errors()</code> .
'namereplace'	Reemplazar con secuencias de escape <code>\N{...}</code> (solo para codificación). Implementado en <code>namereplace_errors()</code> .
'surrogateescape'	En la decodificación, reemplaza el byte con código sustituto individual que va desde U+DC80 a U+DCFF. Este código se volverá a convertir en el mismo byte cuando se use el controlador de errores 'sustituto de paisaje' al codificar los datos. (Ver PEP 383 para más información).

Además, el siguiente controlador de errores es específico de los códecs dados:

Valor	Códecs	Significado
'surrogatepass'	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	Permitir codificación y decodificación de códigos sustitutos. Estos códecs normalmente tratan la presencia de sustitutos como un error.

Nuevo en la versión 3.1: Los manejadores de errores 'surrogateescape' y 'surrogatepass'.

Distinto en la versión 3.4: Los controladores de errores 'surrogatepass' ahora funcionan con los códecs utf-16* y utf-32*.

Nuevo en la versión 3.5: El controlador de errores 'namereplace'.

Distinto en la versión 3.5: Los manejadores de errores 'backslashreplace' ahora funcionan con decodificación y traducción.

El conjunto de valores permitidos puede ampliarse registrando un nuevo controlador de errores con nombre:

`codecs.register_error(name, error_handler)`

Registre la función de manejo de errores `error_handler` bajo el nombre `name`. Se invocará el argumento `error_handler` durante la codificación y decodificación en caso de error, cuando `name` se especifica como el parámetro de errores.

Para la codificación, se llamará a `error_handler` con una instancia `UnicodeEncodeError`, que contiene información sobre la ubicación del error. El controlador de errores debe generar esta o una excepción diferente, o retornar una tupla con un reemplazo para la parte no codificable de la entrada y una posición donde la codificación debe continuar. El reemplazo puede ser `str` o `bytes`. Si el reemplazo son bytes, el codificador simplemente los copiará en el búfer de salida. Si el reemplazo es una cadena de caracteres, el codificador codificará el reemplazo. La codificación continúa en la entrada original en la posición especificada. Los valores de posición negativos se tratarán como relativos al final de la cadena de entrada. Si la posición resultante está fuera del límite, se generará `IndexError`.

La decodificación y traducción funcionan de manera similar, excepto que `UnicodeDecodeError` o `UnicodeTranslateError` se pasarán al controlador y el reemplazo del controlador de errores se colocará directamente en la salida.

Los controladores de errores registrados previamente (incluidos los controladores de errores estándar) se pueden buscar por nombre:

`codecs.lookup_error(name)`

Retorna el controlador de errores previamente registrado con el nombre `name`.

Lanza un `LookupError` en caso de que no se pueda encontrar el controlador.

Los siguientes controladores de errores estándar también están disponibles como funciones de nivel de módulo:

`codecs.strict_errors` (*exception*)
 Implementa el manejo de errores 'estricto': cada error de codificación o decodificación genera un `UnicodeError`.

`codecs.replace_errors` (*exception*)
 Implementa el manejo de errores 'reemplazar' (para *codificaciones de texto* solamente): sustituye '?' por errores de codificación (que serán codificados por el códec), y '\ uffff' (el carácter de reemplazo Unicode) por errores de decodificación.

`codecs.ignore_errors` (*exception*)
 Implementa el manejo de errores 'ignorar': los datos mal formados se ignoran y la codificación o decodificación continúa sin previo aviso.

`codecs.xmlcharrefreplace_errors` (*exception*)
 Implementa el manejo de errores 'xmlcharrefreplace' (para codificar con *codificaciones de texto* solamente): el carácter no codificable se reemplaza por una referencia de caracteres XML apropiada.

`codecs.backslashreplace_errors` (*exception*)
 Implementa el manejo de errores 'backslashreplace' (para *codificaciones de texto*): los datos con formato incorrecto se reemplazan por una secuencia de escape con barra invertida.

`codecs.namereplace_errors` (*exception*)
 Implementa el manejo de errores 'namereplace' (para codificar con *codificaciones de texto*): el carácter no codificable se reemplaza por una secuencia de escape `\N{...}`.

Nuevo en la versión 3.5.

Codificación y decodificación sin estado

La clase base `Codec` define estos métodos que también definen las interfaces de función del codificador y decodificador sin estado:

`Codec.encode` (*input*, [*errors*])
 Codifica el objeto *input* y retorna una tupla (objeto de salida, longitud consumida). Por ejemplo *text encoding* convierte un objeto de cadena de caracteres en un objeto de bytes utilizando una codificación de juego de caracteres particular (por ejemplo, "cp1252" o iso-8859-1).

El argumento *errors* define el manejo de errores a aplicar. El valor predeterminado es el manejo *estricto*.

Es posible que el método no almacene estado en la instancia `Codec`. Use *StreamWriter* para códecs que deben mantener el estado para que la codificación sea eficiente.

El codificador debe poder manejar la entrada de longitud cero y retornar un objeto vacío del tipo de objeto de salida en esta situación.

`Codec.decode` (*input*, [*errors*])
 Decodifica el objeto *input* y retorna una tupla (objeto de salida, longitud consumida). Por ejemplo, para un *codificación de texto*, la decodificación convierte un objeto de bytes codificado usando una codificación de juego de caracteres particular en un objeto de cadena de caracteres.

Para codificaciones de texto y códecs de bytes a bytes, *input* debe ser un objeto de bytes o uno que proporcione la interfaz de búfer de solo lectura, por ejemplo, objetos de búfer y archivos mapeados en memoria.

El argumento *errors* define el manejo de errores a aplicar. El valor predeterminado es el manejo *estricto*.

Es posible que el método no almacene estado en la instancia de `Codec`. Use *StreamReader* para códecs que deben mantener el estado para que la decodificación sea eficiente.

El decodificador debe poder manejar la entrada de longitud cero y retornar un objeto vacío del tipo de objeto de salida en esta situación.

Codificación y decodificación incrementales

Las clases *IncrementalEncoder* y *IncrementalDecoder* proporcionan la interfaz básica para la codificación y decodificación incrementales. La codificación/decodificación de la entrada no se realiza con una llamada a la función de codificador/decodificador sin estado, sino con varias llamadas al método *encode()*/*decode()* del codificador incremental/decodificador. El codificador/decodificador incremental realiza un seguimiento del proceso de codificación/decodificación durante las llamadas a métodos.

La salida combinada de las llamadas al método *encode()*/*decode()* es el mismo que si todas las entradas individuales se unieran en una, y esta entrada se codificara/decodificara con codificador/decodificador sin estado.

Objetos IncrementalEncoder

La clase *IncrementalEncoder* se usa para codificar una entrada en varios pasos. Define los siguientes métodos que cada codificador incremental debe definir para ser compatible con el registro de códec Python.

class `codecs.IncrementalEncoder` (*errors*='strict')

Constructor para una clase instancia de *IncrementalEncoder*.

Todos los codificadores incrementales deben proporcionar esta interfaz de constructor. Son libres de agregar argumentos de palabras clave adicionales, pero el registro de códec de Python solo utiliza los definidos aquí.

La clase *IncrementalEncoder* puede implementar diferentes esquemas de manejo de errores al proporcionar el argumento de palabra clave *errors*. Ver *Manejadores de errores* para posibles valores.

El argumento *errors* se asignará a un atributo del mismo nombre. La asignación a este atributo hace posible cambiar entre diferentes estrategias de manejo de errores durante la vida útil del objeto *IncrementalEncoder*.

encode (*object* [, *final*])

Codifica *object* (teniendo en cuenta el estado actual del codificador) y retorna el objeto codificado resultante. Si esta es la última llamada a *encode()* *final* debe ser verdadero (el valor predeterminado es falso).

reset ()

Restablece el codificador al estado inicial. La salida se descarta: llama a *.encode(object, final=True)*, pasando un byte vacío o una cadena de texto si es necesario, para restablecer el codificador y obtener la salida.

getstate ()

Retorna el estado actual del codificador que debe ser un número entero. La implementación debe asegurarse de que 0 sea el estado más común. (Los estados que son más complicados que los enteros se pueden convertir en un entero al empaquetar/serializar el estado y codificar los bytes de la cadena resultante en un entero).

setstate (*state*)

Establece el estado del codificador en *state*. *state* debe ser un estado de codificador retornado por *getstate()*.

Objetos IncrementalDecoder

La clase *IncrementalDecoder* se usa para decodificar una entrada en varios pasos. Define los siguientes métodos que cada decodificador incremental debe definir para ser compatible con el registro de códec Python.

class `codecs.IncrementalDecoder` (*errors*='strict')

Constructor para una instancia de *IncrementalDecoder*.

Todos los decodificadores incrementales deben proporcionar esta interfaz de constructor. Son libres de agregar argumentos de palabras clave adicionales, pero el registro de códec de Python solo utiliza los definidos aquí.

La clase *IncrementalDecoder* puede implementar diferentes esquemas de manejo de errores al proporcionar el argumento de palabra clave *errors*. Ver *Manejadores de errores* para posibles valores.

El argumento *errors* se asignará a un atributo del mismo nombre. La asignación a este atributo hace posible cambiar entre diferentes estrategias de manejo de errores durante la vida útil del objeto *IncrementalDecoder*.

decode (*object* [, *final*])

Decodifica *object* (teniendo en cuenta el estado actual del decodificador) y retorna el objeto decodificado resultante. Si esta es la última llamada a *decode()* *final* debe ser verdadero (el valor predeterminado es falso). Si *final* es verdadero, el decodificador debe decodificar la entrada por completo y debe vaciar todos los búferes. Si esto no es posible (por ejemplo, debido a secuencias de bytes incompletas al final de la entrada), debe iniciar el manejo de errores al igual que en el caso sin estado (lo que podría generar una excepción).

reset ()

Restablece el decodificador al estado inicial.

getstate ()

Retorna el estado actual del decodificador. Debe ser una tupla con dos elementos, el primero debe ser el búfer que contiene la entrada aún sin codificar. El segundo debe ser un número entero y puede ser información de estado adicional. (La implementación debe asegurarse de que 0 sea la información de estado adicional más común). Si esta información de estado adicional es 0, debe ser posible establecer el decodificador en el estado que no tiene entrada almacenada y 0 como información de estado adicional, de modo que alimentar la entrada previamente almacenada en el búfer al decodificador la retorna al estado anterior sin producir ninguna salida. (La información de estado adicional que es más complicada que los enteros se puede convertir en un entero al empaquetar/serializar la información y codificar los bytes de la cadena resultante en un entero).

setstate (*state*)

Establezca el estado del decodificador en *state*. *state* debe ser un estado de decodificador retornado por *getstate()*.

Codificación y decodificación de flujos

Las clases *StreamWriter* y *StreamReader* proporcionan interfaces de trabajo genéricas que se pueden usar para implementar nuevos submódulos de codificación muy fácilmente. Ir a `encodings.utf_8` para ver un ejemplo de cómo se hace esto.

Objetos StreamWriter

La clase *StreamWriter* es una subclase de *Codec* y define los siguientes métodos que cada escritor del flujo debe definir para ser compatible con el registro de códecs Python.

class `codecs.StreamWriter` (*stream*, *errors*='strict')

Constructor para una instancia de *StreamWriter*.

Todos los escritores de flujos deben proporcionar esta interfaz de constructor. Son libres de agregar argumentos de palabras clave adicionales, pero el registro de códecs de Python solo utiliza los definidos aquí.

El argumento *stream* debe ser un objeto tipo archivo abierto para escribir texto o datos binarios, según corresponda para el códec específico.

La clase *StreamWriter* puede implementar diferentes esquemas de manejo de errores al proporcionar el argumento de palabra clave *errors*. Consulte *Manejadores de errores* para ver los controladores de error estándar que puede admitir el códec de flujo subyacente.

El argumento *errors* se asignará a un atributo del mismo nombre. La asignación a este atributo hace posible cambiar entre diferentes estrategias de manejo de errores durante la vida útil del objeto *StreamWriter*.

write (*object*)

Escribe el contenido del objeto codificado en el flujo.

writelines (*list*)

Escribe la lista concatenada de cadenas en el flujo (posiblemente reutilizando el método `write()`). Los códecs de bytes a bytes estándar no admiten este método.

reset ()

Vacía y restablece los búferes de códec utilizados para mantener el estado.

Llamar a este método debería garantizar que los datos en la salida se pongan en un estado limpio que permita agregar datos nuevos sin tener que volver a escanear todo el flujo para recuperar el estado.

Además de los métodos anteriores, la clase `StreamWriter` también debe heredar todos los demás métodos y atributos del flujo subyacente.

Objetos StreamReader

La clase `StreamReader` es una subclase de `Codec` y define los siguientes métodos que cada lector de flujo debe definir para ser compatible con el registro de códecs de Python.

class `codecs.StreamReader` (*stream*, *errors*='strict')

Constructor para una instancia de `StreamReader`.

Todos los lectores de flujo deben proporcionar esta interfaz de constructor. Son libres de agregar argumentos de palabras clave adicionales, pero el registro de códecs de Python solo utiliza los definidos aquí.

El argumento *stream* debe ser un objeto tipo archivo abierto para leer texto o datos binarios, según corresponda para el códec específico.

La clase `StreamReader` puede implementar diferentes esquemas de manejo de errores al proporcionar el argumento de palabra clave *errors*. Consulte [Manejadores de errores](#) para ver los controladores de error estándar que puede admitir el códec de flujo subyacente.

El argumento *errors* se asignará a un atributo del mismo nombre. La asignación a este atributo hace posible cambiar entre diferentes estrategias de manejo de errores durante la vida útil del objeto `StreamReader`.

El conjunto de valores permitidos para el argumento *errors* se puede ampliar con `register_error()`.

read ([*size* [, *chars* [, *firstline*]]])

Decodifica datos del flujo y retorna el objeto resultante.

El argumento *chars* indica el número de puntos de código decodificados o bytes a retornar. El método `read()` nunca retornará más datos de los solicitados, pero podría retornar menos, si no hay suficientes disponibles.

El argumento *size* indica el número máximo aproximado de bytes codificados o puntos de código para leer para la decodificación. El decodificador puede modificar esta configuración según corresponda. El valor predeterminado -1 indica leer y decodificar tanto como sea posible. Este parámetro está diseñado para evitar tener que decodificar archivos grandes en un solo paso.

La bandera *firstline* indica que sería suficiente retornar solo la primera línea, si hay errores de decodificación en las líneas posteriores.

El método debe usar una estrategia de lectura codiciosa, lo que significa que debe leer la mayor cantidad de datos permitidos dentro de la definición de la codificación y el tamaño dado, por ejemplo si las terminaciones de codificación opcionales o los marcadores de estado están disponibles en la transmisión, también deben leerse.

readline ([*size* [, *keepends*]])

Lee una línea del flujo de entrada y retorna los datos decodificados.

size, si se da, se pasa como argumento de tamaño al método `read()` del *stream*.

Si *keepends* es falso, las terminaciones de línea se eliminarán de las líneas retornadas.

readlines ([*sizehint* [, *keepends*]])

Lee todas las líneas disponibles en el flujo de entrada y las retorna como una lista de líneas.

Los finales de línea se implementan utilizando el método `decode()` del códec y se incluyen en las entradas de la lista si `keepends` es verdadero.

`sizehint`, si se proporciona, se pasa como argumento `size` al método `read()` del `stream`.

reset()

Restablece los búferes de códec utilizados para mantener el estado.

Tenga en cuenta que ningún reposicionamiento de flujo debe suceder. Este método está destinado principalmente a poder recuperarse de errores de decodificación.

Además de los métodos anteriores, la clase `StreamReader` también debe heredar todos los demás métodos y atributos del flujo subyacente.

Objetos StreamReaderWriter

La clase `StreamReaderWriter` es una clase de conveniencia que permite envolver flujos que funcionan tanto en modo de lectura como de escritura.

El diseño es tal que uno puede usar las funciones de fábrica retornadas por la función `lookup()` para construir la instancia.

class `codecs.StreamReaderWriter` (*stream*, *Reader*, *Writer*, *errors*='strict')

Crea una instancia de `StreamReaderWriter`. *stream* debe ser un objeto similar a un archivo. *Reader* y *Writer* deben ser funciones o clases de fábrica que proporcionen la interfaz `StreamReader` y `StreamWriter` respectivamente. El manejo de errores se realiza de la misma manera que se define para los lectores y escritores de flujos.

Las instancias `StreamReaderWriter` definen las interfaces combinadas de `StreamReader` y clases `StreamWriter`. Heredan todos los demás métodos y atributos del flujo subyacente.

Objetos StreamRecoder

La clase `StreamRecoder` traduce datos de una codificación a otra, lo que a veces es útil cuando se trata de diferentes entornos de codificación.

El diseño es tal que uno puede usar las funciones de fábrica retornadas por la función `lookup()` para construir la instancia.

class `codecs.StreamRecoder` (*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors*='strict')

Crea una instancia de `StreamRecoder` que implementa una conversión bidireccional: *encode* y *decode* funcionan en el *frontend*: los datos visibles para la llamada de código `read()` y `write()`, mientras que *Reader* y *Writer* funcionan en el *backend* — los datos en *stream*.

Puede usar estos objetos para realizar transcodificaciones transparentes, por ejemplo, de Latin-1 a UTF-8 y viceversa.

El argumento *stream* debe ser un objeto similar a un archivo.

Los argumentos *encode* y *decode* deben cumplir con la interfaz de `Codec`. *Reader* y *Writer* deben ser funciones o clases de fábrica que proporcionen objetos de la interfaz `StreamReader` y `StreamWriter` respectivamente.

El manejo de errores se realiza de la misma manera que se define para los lectores y escritores de flujos.

las instancias `StreamRecoder` definen las interfaces combinadas de las clases `StreamReader` y `StreamWriter`. Heredan todos los demás métodos y atributos del flujo subyacente.

7.2.2 Codificaciones y Unicode

Las cadenas de caracteres se almacenan internamente como secuencias de puntos de código en el rango `0x0 - 0x10FFFF`. (Consulte [PEP 393](#) para obtener más detalles sobre la implementación.) Una vez que se utiliza un objeto de cadena de caracteres fuera de la CPU y la memoria, la *endianness* y cómo se almacenan estos conjuntos como bytes se convierte en un problema. Al igual que con otros códecs, la serialización de una cadena en una secuencia de bytes se conoce como *encoding*, y la recreación de la cadena a partir de la bytes se conoce como *decoding*. Hay una variedad de códecs de serialización de texto diferentes, que se denominan colectivamente como *codificaciones de texto*.

La codificación de texto más simple (llamada `'latin-1'` o `'iso-8859-1'`) asigna los puntos de código `0-255` a los bytes `0x0 - 0xff`, lo que significa que un objeto de cadena de caracteres que contiene puntos de código encima de `U+00FF` no se puede codificar con este códec. Al hacerlo, lanzará un `UnicodeEncodeError` que se parece a lo siguiente (aunque los detalles del mensaje de error pueden diferir): `UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256)`.

Hay otro grupo de codificaciones (las llamadas codificaciones de mapa de caracteres) que eligen un subconjunto diferente de todos los puntos de código Unicode y cómo estos puntos de código se asignan a los bytes `0x0 - 0xff`. Para ver cómo se hace esto, simplemente abra, por ejemplo `encodings/cp1252.py` (que es una codificación que se usa principalmente en Windows). Hay una cadena constante con 256 caracteres que le muestra qué carácter está asignado a qué valor de byte.

Todas estas codificaciones solo pueden codificar 256 de los 1114112 puntos de código definidos en Unicode. Una manera simple y directa que puede almacenar cada punto de código Unicode es almacenar cada punto de código como cuatro bytes consecutivos. Hay dos posibilidades: almacenar los bytes en orden *big endian* o *little endian*. Estas dos codificaciones se denominan `UTF-32-BE` y `UTF-32-LE` respectivamente. Su desventaja es que si por ejemplo usa `UTF-32-BE` en una pequeña máquina endian, siempre tendrá que intercambiar bytes en la codificación y decodificación. `UTF-32` evita este problema: los bytes siempre estarán en *endianness* natural. Cuando estos bytes son leídos por una CPU con una *endianness* diferente, entonces los bytes deben intercambiarse. Para poder detectar la resistencia de una secuencia de bytes `UTF-16` o `UTF-32`, existe la llamada BOM («Marca de orden de bytes», o en inglés *Byte Order Mark*). Este es el carácter Unicode `U+FEFF`. Este carácter puede anteponerse a cada secuencia de bytes `UTF-16` o `UTF-32`. La versión intercambiada de bytes de este carácter (`0xFFFE`) es un carácter ilegal que puede no aparecer en un texto Unicode. Entonces, cuando el primer carácter en una secuencia de bytes `UTF-16` o `UTF-32` parece ser un `U+FFFE`, los bytes deben intercambiarse en la decodificación. Desafortunadamente, el carácter `U+FEFF` tenía un segundo propósito como `ESPACIO SIN QUIEBRE DE ANCHO CERO`: un carácter que no tiene ancho y no permite dividir una palabra. Puede por ejemplo ser usado para dar pistas a un algoritmo de ligadura. Con Unicode 4.0, el uso de `U+FEFF` como `ESPACIO SIN QUIEBRE DE ANCHO CERO` ha quedado en desuso (con `U+2060` (`WORD JOINER`) asumiendo este rol). Sin embargo, el software Unicode aún debe ser capaz de manejar `U+FEFF` en ambos roles: como BOM es un dispositivo para determinar el diseño de almacenamiento de los bytes codificados, y desaparece una vez que la secuencia de bytes ha sido decodificada en una cadena; como un `ESPACIO SIN QUIEBRE DE ANCHO CERO` es un personaje normal que se decodificará como cualquier otro.

Hay otra codificación que puede codificar el rango completo de caracteres Unicode: `UTF-8`. `UTF-8` es una codificación de 8 bits, lo que significa que no hay problemas con el orden de bytes en `UTF-8`. Cada byte en una secuencia de bytes `UTF-8` consta de dos partes: bits marcadores (los bits más significativos) y bits de carga útil. Los bits marcadores son una secuencia de cero a cuatro bits 1 seguidos de un bit 0. Los caracteres Unicode se codifican de esta manera (con `x` siendo bits de carga útil, que cuando se concatenan dan el carácter Unicode):

Rango	Codificación
<code>U-00000000 ... U-0000007F</code>	<code>0xxxxxxx</code>
<code>U-00000080 ... U-000007FF</code>	<code>110xxxxx 10xxxxxx</code>
<code>U-00000800 ... U-0000FFFF</code>	<code>1110xxxx 10xxxxxx 10xxxxxx</code>
<code>U-00010000 ... U-0010FFFF</code>	<code>11110xxx 10xxxxxx 10xxxxxx 10xxxxxx</code>

El bit menos significativo del carácter Unicode es el bit `x` más a la derecha.

Como `UTF-8` es una codificación de 8 bits, no se requiere una lista de materiales y cualquier carácter `U+FEFF` en la cadena decodificada (incluso si es el primer carácter) se trata como un *ESPACIO SIN QUIEBRE DE ANCHO CERO* (*"ZERO WIDTH NO-BREAK SPACE"*).

Sin información externa, es imposible determinar de manera confiable qué codificación se utilizó para codificar una cadena de caracteres. Cada codificación de mapa de caracteres puede decodificar cualquier secuencia de bytes aleatoria. Sin embargo, eso no es posible con UTF-8, ya que las secuencias de bytes UTF-8 tienen una estructura que no permite secuencias de bytes arbitrarias. Para aumentar la confiabilidad con la que se puede detectar una codificación UTF-8, Microsoft inventó una variante de UTF-8 (que Python 2.5 llama "utf-8-sig") para su programa Bloc de notas: Antes de cualquiera de los Unicode los caracteres se escriben en el archivo, se escribe una lista de materiales codificada en UTF-8 (que se ve así como una secuencia de bytes: `0xef`, `0xbb`, `0xbf`). Como es bastante improbable que cualquier archivo codificado del mapa de caracteres comience con estos valores de bytes (que, por ejemplo, se correlacionarán con

LETRA LATINA PEQUEÑA I CON DIAERESIS
SEÑALADO A LA DERECHA DE DOBLE ÁNGULO MARCA DE CITA
SIGNO DE PREGUNTA INVERTIDO

en iso-8859-1), esto aumenta la probabilidad de que una codificación `utf-8-sig` pueda adivinarse correctamente a partir de la secuencia de bytes. Por lo tanto, aquí la lista de materiales no se utiliza para poder determinar el orden de bytes utilizado para generar la secuencia de bytes, sino como una firma que ayuda a adivinar la codificación. Al codificar, el códec `utf-8-sig` escribirá `0xef`, `0xbb`, `0xbf` como los primeros tres bytes del archivo. Al decodificar, `utf-8-sig` omitirá esos tres bytes si aparecen como los primeros tres bytes en el archivo. En UTF-8, se desaconseja el uso de la lista de materiales y, en general, debe evitarse.

7.2.3 Codificaciones estándar

Python viene con una serie de códec integrados, ya sea implementados como funciones C o con diccionarios como tablas de mapeo. La siguiente tabla enumera los códec por nombre, junto con algunos alias comunes y los idiomas para los que probablemente se usa la codificación. Ni la lista de alias ni la lista de idiomas deben ser exhaustivas. Tenga en cuenta que las alternativas de ortografía que solo difieren en el caso o usan un guión en lugar de un guión bajo también son alias válidos; por lo tanto, por ejemplo `'utf-8'` es un alias válido para el códec `'utf_8'`.

CPython implementation detail: Algunas codificaciones comunes pueden omitir la maquinaria de búsqueda de códec para mejorar el rendimiento. CPython solo reconoce estas oportunidades de optimización para un conjunto limitado de alias (sin distinción entre mayúsculas y minúsculas): `utf-8`, `utf8`, `latin-1`, `latin1`, `iso-8859-1`, `iso8859-1`, `mbcs` (solo Windows), `ascii`, `us-ascii`, `utf-16`, `utf16`, `utf-32`, `utf32`, y lo mismo usando guiones bajos en lugar de guiones. El uso de alias alternativos para estas codificaciones puede resultar en una ejecución más lenta.

Distinto en la versión 3.6: Oportunidad de optimización reconocida para `us-ascii`.

Muchos de los juegos de caracteres admiten los mismos idiomas. Varían en caracteres individuales (por ejemplo, si el SIGNO EURO es compatible o no), y en la asignación de caracteres para codificar posiciones. Para los idiomas europeos en particular, generalmente existen las siguientes variantes:

- un conjunto de códigos ISO 8859
- una página de códigos de Microsoft Windows, que generalmente se deriva de un conjunto de códigos 8859, pero reemplaza los caracteres de control con caracteres gráficos adicionales
- una página de códigos EBCDIC de IBM
- una página de códigos de IBM PC, que es compatible con ASCII

Códec	Alias	Lenguajes
<code>ascii</code>	<code>646</code> , <code>us-ascii</code>	Inglés
<code>big5</code>	<code>big5-tw</code> , <code>csbig5</code>	Chino Tradicional
<code>big5hkscs</code>	<code>big5-hkscs</code> , <code>hkscs</code>	Chino Tradicional
<code>cp037</code>	<code>IBM037</code> , <code>IBM039</code>	Inglés
<code>cp273</code>	<code>273</code> , <code>IBM273</code> , <code>csIBM273</code>	Alemán Nuevo en la versión 3.4.
<code>cp424</code>	<code>EBCDIC-CP-HE</code> , <code>IBM424</code>	Hebreo
<code>cp437</code>	<code>437</code> , <code>IBM437</code>	Inglés

Continúa en la página siguiente

Tabla 1 – proviene de la página anterior

Códec	Alias	Lenguajes
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	Europa Occidental
cp720		Árabe
cp737		Griego
cp775	IBM775	Lenguajes bálticos
cp850	850, IBM850	Europa Occidental
cp852	852, IBM852	Europa central y del este
cp855	855, IBM855	Búlgaro, Bielorruso, Macedonio, Ruso, Serbio
cp856		Hebreo
cp857	857, IBM857	Turco
cp858	858, IBM858	Europa Occidental
cp860	860, IBM860	Portugués
cp861	861, CP-IS, IBM861	Islandés
cp862	862, IBM862	Hebreo
cp863	863, IBM863	Canadiense
cp864	IBM864	Árabe
cp865	865, IBM865	Danés, Noruego
cp866	866, IBM866	Ruso
cp869	869, CP-GR, IBM869	Griego
cp874		Tailandés
cp875		Griego
cp932	932, ms932, mskanji, ms-kanji	Japonés
cp949	949, ms949, uhc	Coreano
cp950	950, ms950	Chino Tradicional
cp1006		Urdu
cp1026	ibm1026	Turco
cp1125	1125, ibm1125, cp866u, ruscii	Ucraniano Nuevo en la versión 3.4.
cp1140	ibm1140	Europa Occidental
cp1250	windows-1250	Europa central y del este
cp1251	windows-1251	Búlgaro, Bielorruso, Macedonio, Ruso, Serbio
cp1252	windows-1252	Europa Occidental
cp1253	windows-1253	Griego
cp1254	windows-1254	Turco
cp1255	windows-1255	Hebreo
cp1256	windows-1256	Árabe
cp1257	windows-1257	Lenguajes bálticos
cp1258	windows-1258	Vietnamita
euc_jp	eucjp, ujis, u-jis	Japonés
euc_jis_2004	jisx0213, eucjis2004	Japonés
euc_jisx0213	eucjisx0213	Japonés
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	Coreano
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	Chino simplificado
gbk	936, cp936, ms936	Chino Unificado
gb18030	gb18030-2000	Chino Unificado
hz	hzgb, hz-gb, hz-gb-2312	Chino simplificado

Continúa en la página siguiente

Tabla 1 – proviene de la página anterior

Códec	Aliases	Lenguajes
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	Japonés
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	Japonés
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	Japonés, Coreano, Chino simplificado, Europa occidental, Griego
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	Japonés
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	Japonés
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	Japonés
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	Coreano
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	Europa Occidental
iso8859_2	iso-8859-2, latin2, L2	Europa central y del este
iso8859_3	iso-8859-3, latin3, L3	Esperanto, Maltés
iso8859_4	iso-8859-4, latin4, L4	Lenguajes bálticos
iso8859_5	iso-8859-5, cyrillic	Búlgaro, Bielorruso, Macedonio, Ruso, Serbio
iso8859_6	iso-8859-6, arabic	Árabe
iso8859_7	iso-8859-7, greek, greek8	Griego
iso8859_8	iso-8859-8, hebrew	Hebreo
iso8859_9	iso-8859-9, latin5, L5	Turco
iso8859_10	iso-8859-10, latin6, L6	Lenguajes nórdicos
iso8859_11	iso-8859-11, thai	Lenguajes tailandeses
iso8859_13	iso-8859-13, latin7, L7	Lenguajes bálticos
iso8859_14	iso-8859-14, latin8, L8	Lenguajes Celtas
iso8859_15	iso-8859-15, latin9, L9	Europa Occidental
iso8859_16	iso-8859-16, latin10, L10	Europa sudoriental
johab	cp1361, ms1361	Coreano
koi8_r		Ruso
koi8_t		Tayiko Nuevo en la versión 3.5.
koi8_u		Ucraniano
kz1048	kz_1048, strk1048_2002, rk1048	Kazajo Nuevo en la versión 3.5.
mac_cyrillic	maccyrillic	Búlgaro, Bielorruso, Macedonio, Ruso, Serbio
mac_greek	macgreek	Griego
mac_iceland	maciceland	Islandés
mac_latin2	maclatin2, maccentraleurope	Europa central y del este
mac_roman	macroman, macintosh	Europa Occidental
mac_turkish	macturkish	Turco
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	Kazajo
shift_jis	csshiftjis, shiftjis, sjis, s_jis	Japonés
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	Japonés
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	Japonés
utf_32	U32, utf32	todos los lenguajes
utf_32_be	UTF-32BE	todos los lenguajes
utf_32_le	UTF-32LE	todos los lenguajes
utf_16	U16, utf16	todos los lenguajes
utf_16_be	UTF-16BE	todos los lenguajes
utf_16_le	UTF-16LE	todos los lenguajes

Continúa en la página siguiente

Tabla 1 – proviene de la página anterior

Códec	Alias	Lenguajes
utf_7	U7, unicode-1-1-utf-7	todos los lenguajes
utf_8	U8, UTF, utf8, cp65001	todos los lenguajes
utf_8_sig		todos los lenguajes

Distinto en la versión 3.4: Los codificadores utf-16* y utf-32* ya no permiten codificar puntos de código sustitutos (U+D800 – U+DFFF). Los decodificadores utf-32* ya no decodifican secuencias de bytes que corresponden a puntos de código sustituto.

Distinto en la versión 3.8: cp65001 ahora es un alias de utf_8.

7.2.4 Codificaciones específicas de Python

Varios códecs predefinidos son específicos de Python, por lo que sus nombres de códec no tienen significado fuera de Python. Estos se enumeran en las tablas a continuación según los tipos de entrada y salida esperados (tenga en cuenta que si bien las codificaciones de texto son el caso de uso más común para los códecs, la infraestructura de códecs subyacente admite transformaciones de datos arbitrarias en lugar de solo codificaciones de texto). Para los códecs asimétricos, el significado indicado describe la dirección de codificación.

Codificaciones de texto

Los siguientes códecs proporcionan codificación de *str* a *bytes* y decodificación de *bytes-like object* a *str*, similar a las codificaciones de texto Unicode.

Códec	Alias	Significado
idna		Implementar RFC 3490 , ver también <code>encodings.idna</code> . Solo se admite <code>errors='strict'</code> .
mbcs	ansi, dbcs	Solo Windows: codifique el operando de acuerdo con la página de códigos ANSI (CP_ACP).
oem		Solo Windows: codifique el operando de acuerdo con la página de códigos OEM (CP_OEMCP). Nuevo en la versión 3.6.
palms		Codificación de PalmOS 3.5.
punycode		Implementar RFC 3492 . Los códecs con estado no son compatibles.
raw_unicode_escape		Codificación Latin-1 con <code>\uXXXX</code> y <code>\UXXXXXXXX</code> para otros puntos de código. Las barras invertidas existentes no se escapan de ninguna manera. Se usa en el protocolo Python <i>pickle</i> .
indefinido		Lanza una excepción para todas las conversiones, incluso cadenas vacías. El controlador de errores se ignora.
unicode_escape		Codificación adecuada como contenido de un literal Unicode en código fuente Python codificado en ASCII, excepto que no se escapan las comillas. Decodificar desde el código fuente Latin-1. Tenga en cuenta que el código fuente de Python realmente usa UTF-8 por defecto

Distinto en la versión 3.8: Se elimina el códec «unicode_internal».

Transformaciones Binarias

Los siguientes códecs proporcionan transformaciones binarias: mapeos de *bytes-like object* a *bytes*. No son compatibles con `bytes.decode()` (que solo produce *str* de salida).

Códec	Alias	Significado	Codificador / decodificador
base64_codec ¹	base64, base_64	Convierta el operando a MIME base64 multilínea (el resultado siempre incluye un <code>'\n'</code> final). Distinto en la versión 3.4: acepta cualquier <i>bytes-like object</i> como entrada para codificar y decodificar	<code>base64.encodebytes()</code> / <code>base64.decodebytes()</code>
bz2_codec	bz2	Comprime el operando usando bz2.	<code>bz2.compress()</code> / <code>bz2.decompress()</code>
hex_codec	hex	Convierte el operando en representación hexadecimal, con dos dígitos por byte.	<code>binascii.b2a_hex()</code> / <code>binascii.a2b_hex()</code>
quopri_codec	quopri, quoted-printable, quoted_printable	Convierte el operando a MIME citado imprimible.	<code>quopri.encode()</code> con <code>quotetabs=True</code> / <code>quopri.decode()</code>
uu_codec	uu	Convierte el operando usando uuencode.	<code>uu.encode()</code> / <code>uu.decode()</code>
zlib_codec	zip, zlib	Comprime el operando usando gzip.	<code>zlib.compress()</code> / <code>zlib.decompress()</code>

Nuevo en la versión 3.2: Restauración de las transformaciones binarias.

Distinto en la versión 3.4: Restauración de los alias para las transformaciones binarias.

Transformaciones de texto

El siguiente códec proporciona una transformación de texto: un mapeo de *str* a *str*. No es compatible con *str.encode()* (que solo produce *bytes* de salida).

Códec	Alias	Significado
rot_13	rot13	Retorna el cifrado César (<i>Caesar-cypher</i>) del operando.

Nuevo en la versión 3.2: Restauración de la transformación de texto `rot_13`.

Distinto en la versión 3.4: Restauración del alias `rot13`.

7.2.5 `encodings.idna` — Nombres de dominio internacionalizados en aplicaciones

Este módulo implementa **RFC 3490** (nombres de dominio internacionalizados en aplicaciones) y **RFC 3492** (*Nameprep*: un perfil de *Stringprep* para nombres de dominio internacionalizados (IDN)). Se basa en la codificación `punycode` y *stringprep*.

If you need the IDNA 2008 standard from **RFC 5891** and **RFC 5895**, use the third-party *idna module* <<https://pypi.org/project/idna/>>.

Estas RFC juntas definen un protocolo para admitir caracteres no ASCII en los nombres de dominio. Un nombre de dominio que contiene caracteres no ASCII (como `www.Alliancefrançaise.nu`) se convierte en una codificación compatible con ASCII (ACE, como `www.xn--alliancefranaise-npb.nu`). La forma ACE del

¹ Además de *objetos similares a bytes*, `'base64_codec'` también acepta instancias solo ASCII de *str* para decodificación

nombre de dominio se utiliza en todos los lugares donde el protocolo no permite caracteres arbitrarios, como consultas DNS, campos HTTP *Host*, etc. Esta conversión se lleva a cabo en la aplicación; si es posible invisible para el usuario: la aplicación debe convertir de forma transparente las etiquetas de dominio Unicode a IDNA en el cable, y volver a convertir las etiquetas ACE a Unicode antes de presentarlas al usuario.

Python admite esta conversión de varias maneras: el códec `idna` realiza la conversión entre Unicode y ACE, separando una cadena de entrada en etiquetas basadas en los caracteres separadores definidos en la sección 3.1 de RFC 3490 [RFC 3490#section-3.1](#) y convertir cada etiqueta a ACE según sea necesario, y por el contrario, separar una cadena de bytes de entrada en etiquetas basadas en el separador `.` y convertir cualquier etiqueta ACE encontrada en unicode. Además, el módulo `socket` convierte de forma transparente los nombres de host Unicode a ACE, por lo que las aplicaciones no necesitan preocuparse por convertir los nombres de host ellos mismos cuando los pasan al módulo de socket. Además de eso, los módulos que tienen nombres de host como parámetros de función, como `http.client` y `ftplib`, aceptan nombres de host Unicode (`http.client` y luego también envían un mensaje transparente IDNA *hostname* en el campo *Host* si envía ese campo).

Al recibir nombres de host desde el cable (como en la búsqueda inversa de nombres), no se realiza una conversión automática a Unicode: las aplicaciones que deseen presentar dichos nombres de host al usuario deben decodificarlos en Unicode.

El módulo `encodings.idna` también implementa el procedimiento *nameprep*, que realiza ciertas normalizaciones en los nombres de host, para lograr la insensibilidad a mayúsculas y minúsculas de los nombres de dominio internacionales y unificar caracteres similares. Las funciones *nameprep* se pueden usar directamente si lo desea.

`encodings.idna.nameprep(label)`

Retorna la versión pasada por *nameprep* (o versión *nameprepped*) de *label*. La implementación actualmente asume cadenas de caracteres de consulta, por lo que `AllowUnassigned` es verdadero.

`encodings.idna.ToASCII(label)`

Convierte una etiqueta a ASCII, como se especifica en [RFC 3490](#). Se supone que `UseSTD3ASCIIRules` es falso.

`encodings.idna.ToUnicode(label)`

Convierte una etiqueta a Unicode, como se especifica en [RFC 3490](#).

7.2.6 `encodings.mbc`s — Página de códigos ANSI de Windows

Este módulo implementa la página de códigos ANSI (CP_ACP).

Availability: solo Windows.

Distinto en la versión 3.3: Admite cualquier controlador de errores.

Distinto en la versión 3.2: Antes de 3.2, se ignoraba el argumento *errors*; `'replace'` siempre se usó para codificar e `'ignore'` para decodificar.

7.2.7 `encodings.utf_8_sig` — Códec UTF-8 con firma BOM

Este módulo implementa una variante del códec UTF-8. Al codificar, una lista de materiales codificada en UTF-8 se antepone a los bytes codificados en UTF-8. Para el codificador con estado esto solo se hace una vez (en la primera escritura en el flujo de bytes). En la decodificación, se omitirá una lista de materiales opcional codificada en UTF-8 al comienzo de los datos.

Los módulos descritos en este capítulo proporcionan una variedad de tipos de datos especializados, como fechas y horas, matrices de tipo fijo (*fixed-type arrays*), colas de montículos (*heap queues*), colas de doble extremo (*double-ended queues*) y enumeraciones.

Python también proporciona algunos tipos de datos integrados, concretamente *dict*, *list*, *set* y *frozenset*, y *tuple*. La clase *str* se utiliza para contener cadenas de caracteres Unicode, y las clases *bytes* y *bytearray* se utilizan para contener datos binarios.

En este capítulo se documentan los siguientes módulos:

8.1 `datetime` — Tipos básicos de fecha y hora

Source code: [Lib/datetime.py](#)

El módulo `datetime` proporciona clases para manipular fechas y horas.

Si bien la implementación permite operaciones aritméticas con fechas y horas, su principal objetivo es poder extraer campos de forma eficiente para su posterior manipulación o formateo.

Ver también:

Módulo `calendar` Funciones generales relacionadas a `calendar`.

Módulo `time` Acceso a tiempo y conversiones.

Paquete `dateutil` Biblioteca de terceros con zona horaria ampliada y soporte de análisis.

8.1.1 Objetos conscientes (*aware*) y naïfs (*naive*)

Date and time objects may be categorized as «aware» or «naive» depending on whether or not they include timezone information.

With sufficient knowledge of applicable algorithmic and political time adjustments, such as time zone and daylight saving time information, an **aware** object can locate itself relative to other aware objects. An aware object represents 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 is 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.

El módulo `datetime` solo proporciona una clase concreta `tzinfo`, la clase `timezone`. La clase `timezone` puede representar zonas horarias simples con desplazamientos fijos desde UTC, como UTC o las zonas horarias EST y EDT de América del Norte. La compatibilidad de zonas horarias con niveles de detalle más profundos depende de la aplicación. Las reglas para el ajuste del tiempo en todo el mundo son mas políticas que racionales, cambian con frecuencia y no existe un estándar adecuado para cada aplicación, aparte de UTC.

8.1.2 Constantes

El módulo `datetime` exporta las siguientes constantes:

`datetime.MINYEAR`

El número de año más pequeño permitido en un objeto `date` o `datetime`. `MINYEAR` es '1'.

`datetime.MAXYEAR`

El número de año más grande permitido en un objeto `date` o en `datetime`. `MAXYEAR` es '9999'.

8.1.3 Tipos disponibles

class `datetime.date`

Una fecha naïf (*naive*) idealizada, suponiendo que el calendario gregoriano actual siempre estuvo, y siempre estará, vigente. Atributos: `year`, `month`, y `day`.

class `datetime.time`

Un tiempo idealizado, independiente de cualquier día en particular, suponiendo que cada día tenga exactamente 24* 60* 60 segundos. (Aquí no hay noción de «segundos intercalares».) Atributos: `hour`, `minute`, `second`, `microsecond`, y `tzinfo`.

class `datetime.datetime`

Una combinación de una fecha y una hora. Atributos: `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`, y `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).

¹ Es decir, si ignoramos los efectos de la relatividad

class `datetime.timezone`

Una clase que implementa la clase de base abstracta `tzinfo` como un desplazamiento fijo desde el UTC.

Nuevo en la versión 3.2.

Los objetos de este tipo son inmutables.

Relaciones de subclase:

```
object
  timedelta
  tzinfo
    timezone
  time
  date
    datetime
```

Propiedades comunes

Las clases `date`, `datetime`, `time`, y `timezone` comparten estas características comunes:

- Los objetos de este tipo son inmutables.
- Los objetos de este tipo son *hashable*, lo que significa que pueden usarse como claves de diccionario.
- Los objetos de este tipo admiten el *pickling* eficiente a través del módulo `pickle`.

Determinando si un objeto es Consciente (*Aware*) o Naïf (*Naive*)

Los objetos del tipo `date` son siempre naïf (*naive*).

Un objeto de tipo `time` o `datetime` puede ser consciente (*aware*) o naïf (*naive*).

A `datetime` object `d` is aware if both of the following hold:

1. `d.tzinfo` no es `None`
2. `d.tzinfo.utcoffset(d)` no retorna `None`

De lo contrario, `d` es naïf (*naive*).

A `time` object `t` is aware if both of the following hold:

1. `t.tzinfo` no es `None`
2. `t.tzinfo.utcoffset(None)` no retorna `None`.

De lo contrario, `t` es naïf (*naive*).

La distinción entre los objetos consciente (*aware*) y naïf (*naive*) no se aplica a `timedelta`.

8.1.4 Objetos `timedelta`

El objeto `timedelta` representa una duración, la diferencia entre dos fechas u horas.

class `datetime.timedelta` (`days=0`, `seconds=0`, `microseconds=0`, `milliseconds=0`, `minutes=0`,
`hours=0`, `weeks=0`)

Todos los argumentos son opcionales y predeterminados a 0. Los argumentos pueden ser enteros o flotantes, y pueden ser positivos o negativos.

Solo `days`, `seconds` y `microseconds` se almacenan internamente. Los argumentos se convierten a esas unidades:

- Un milisegundo se convierte a 1000 microsegundos.
- Un minuto se convierte a 60 segundos.
- Una hora se convierte a 3600 segundos.

- Una semana se convierte a 7 días.

y los días, segundos y microsegundos se normalizan para que la representación sea única, con

- $0 \leq \text{microsegundos} < 1000000$
- $0 \leq \text{segundos} < 3600 \cdot 24$ (el número de segundos en un día)
- $-999999999 \leq \text{days} \leq 999999999$

El siguiente ejemplo ilustra cómo cualquier argumento además de *days*, *seconds* y *microseconds* se «fusionan» y normalizan en esos tres atributos resultantes:

```
>>> from datetime import timedelta
>>> delta = timedelta(
...     days=50,
...     seconds=27,
...     microseconds=10,
...     milliseconds=29000,
...     minutes=5,
...     hours=8,
...     weeks=2
... )
>>> # Only days, seconds, and microseconds remain
>>> delta
datetime.timedelta(days=64, seconds=29156, microseconds=10)
```

Si algún argumento es flotante y hay microsegundos fraccionarios, los microsegundos fraccionarios que quedan de todos los argumentos se combinan y su suma se redondea al microsegundo más cercano utilizando el desempate de medio redondeo a par. Si ningún argumento es flotante, los procesos de conversión y normalización son exactos (no se pierde información).

If the normalized value of days lies outside the indicated range, *OverflowError* is raised.

Tenga en cuenta que la normalización de los valores negativos puede ser sorprendente al principio. Por ejemplo:

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Atributos de clase:

`timedelta.min`

The most negative *timedelta* object, `timedelta(-999999999)`.

`timedelta.max`

El objeto más positivo de la *timedelta*, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

`timedelta.resolution`

La diferencia más pequeña posible entre los objetos no iguales *timedelta* `timedelta(microseconds=1)`.

Tenga en cuenta que, debido a la normalización, `timedelta.max > -timedelta.min`. `-timedelta.max` no es representable como un objeto *timedelta*.

Atributos de instancia (solo lectura):

Atributo	Valor
<code>days</code>	Entre -999999999 y 999999999 inclusive
<code>seconds</code>	Entre 0 y 86399 inclusive
<code>microseconds</code>	Entre 0 y 999999 inclusive

Operaciones soportadas:

Operación	Resultado
$t1 = t2 + t3$	Suma de $t2$ y $t3$. Después $t1 - t2 == t3$ y $t1 - t3 == t2$ son verdaderos. (1)
$t1 = t2 - t3$	La suma de $t2$ y $t3$. Después $t1 == t2 - t3$ y $t2 == t1 + t3$ son verdaderos. (1)(6)
$t1 = t2 * i$ o $t1 = i * t2$	Delta multiplied by an integer. Afterwards $t1 // i == t2$ is true, provided $i \neq 0$.
	En general, $t1 * i == t1 * (i-1) + t1$ es verdadero. (1)
$t1 = t2 * f$ o $t1 = f * t2$	Delta multiplicado por un número decimal. El resultado se redondea al múltiplo mas cercano de <code>timedelta.resolution</code> usando redondeo de medio a par.
$f = t2 / t3$	División (3) de la duración total $t2$ por unidad de intervalo $t3$. Retorna un objeto <code>float</code> .
$t1 = t2 / f$ o $t1 = t2 / i$	Delta dividido por un número decimal o un entero. El resultado se redondea al múltiplo más cercano de <code>timedelta.resolution</code> usando redondeo de medio a par.
$t1 = t2 // i$ o $t1 = t2 // t3$	El piso (<i>floor</i>) se calcula y el resto (si lo hay) se descarta. En el segundo caso, se retorna un entero. (3)
$t1 = t2 \% t3$	El resto se calcula como un objeto <code>timedelta</code> . (3)
$q, r = \text{divmod}(t1, t2)$	Calcula el cociente y el resto: $q = t1 // t2$ (3) y $r = t1 \% t2$. q es un entero y r es un objeto <code>timedelta</code> .
$+t1$	Retorna un objeto <code>timedelta</code> con el mismo valor. (2)
$-t1$	equivalente a <code>timedelta(-t1.days, -t1.seconds, -t1.microseconds)</code> , y a $t1 * -1$. (1)(4)
$\text{abs}(t)$	equivalente a $+t$ cuando $t.days \geq 0$, y a $-t$ cuando $t.days < 0$. (2)
$\text{str}(t)$	Retorna una cadena de caracteres en la forma <code>[D] day[s],][H]H:MM:SS[.UUUUUU]</code> , donde D es negativo para negativo t . (5)
$\text{repr}(t)$	Retorna una representación de cadena del objeto <code>timedelta</code> como una llamada de constructor con valores de atributos canónicos.

Notas:

- (1) Esto es exacto pero puede desbordarse.
- (2) Esto es exacto pero no puede desbordarse.
- (3) División por 0 genera `ZeroDivisionError`.
- (4) `-timedelta.max` no es representable como un objeto `timedelta`.
- (5) Las representaciones de cadena de caracteres de los objetos `timedelta` se normalizan de manera similar a su representación interna. Esto conduce a resultados algo inusuales para `timedeltas` negativos. Por ejemplo:

```
>>> timedelta(hours=-5)
datetime.timedelta(days=-1, seconds=68400)
>>> print(_)
-1 day, 19:00:00
```

- (6) La expresión $t2 - t3$ siempre será igual a la expresión $t2 + (-t3)$ excepto cuando $t3$ es igual a `timedelta.max`; en ese caso, el primero producirá un resultado mientras que el segundo se desbordará.

Además de las operaciones enumeradas anteriormente, los objetos `timedelta` admiten ciertas sumas y restas con objetos `date` y `datetime` (ver más abajo).

Distinto en la versión 3.2: La división de piso y la división verdadera de un objeto `timedelta` por otro `timedelta` ahora son compatibles, al igual que las operaciones restantes y la función `divmod()`. La división verdadera y multiplicación de un objeto `timedelta` por un objeto flotante ahora son compatibles.

Comparaciones de los objetos `timedelta` son compatibles, con algunas limitaciones.

Las comparaciones `==` o `!=` Siempre retornan `bool`, sin importar el tipo de objeto comparado:

```
>>> from datetime import timedelta
>>> delta1 = timedelta(seconds=57)
>>> delta2 = timedelta(hours=25, seconds=2)
>>> delta2 != delta1
```

(continué en la próxima página)

(proviene de la página anterior)

```
True
>>> delta2 == 5
False
```

Para todas las demás comparaciones (como `<` y `>`), cuando un objeto `timedelta` se compara con un objeto de un tipo diferente, se genera `TypeError`:

```
>>> delta2 > delta1
True
>>> delta2 > 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'datetime.timedelta' and 'int'
```

En contextos booleanos, un objeto `timedelta` se considera verdadero si y solo si no es igual a `timedelta(0)`.

Métodos de instancia:

`timedelta.total_seconds()`

Retorna el número total de segundos contenidos en la duración. Equivalente a `td / timedelta(segundos=1)`. Para unidades de intervalo que no sean segundos, use la forma de división directamente (por ejemplo, `td / timedelta(microseconds=1)`).

Tenga en cuenta que para intervalos de tiempo muy largos (más de 270 años en la mayoría de las plataformas) este método perderá precisión de microsegundos.

Nuevo en la versión 3.2.

Ejemplos de uso: `timedelta`

Ejemplos adicionales de normalización:

```
>>> # Components of another_year add up to exactly 365 days
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)
>>> year == another_year
True
>>> year.total_seconds()
31536000.0
```

Ejemplos de `timedelta` aritmética:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> ten_years = 10 * year
>>> ten_years
datetime.timedelta(days=3650)
>>> ten_years.days // 365
10
>>> nine_years = ten_years - year
>>> nine_years
datetime.timedelta(days=3285)
>>> three_years = nine_years // 3
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 3)
```

8.1.5 Objeto `date`

El objeto `date` representa una fecha (año, mes y día) en un calendario idealizado, el calendario gregoriano actual se extiende indefinidamente en ambas direcciones.

El 1 de enero del año 1 se llama día número 1, el 2 de enero del año 1 se llama día número 2, y así sucesivamente.²

class `datetime.date` (*year, month, day*)

Todos los argumentos son obligatorios. Los argumentos deben ser enteros, en los siguientes rangos:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= day <= number of days in the given month and year`

Si se proporciona un argumento fuera de esos rangos, `ValueError` se genera.

Otros constructores, todos los métodos de clase:

classmethod `date.today()`

Retorna la fecha local actual.

Esto es equivalente a `date.fromtimestamp(time.time())`.

classmethod `date.fromtimestamp(timestamp)`

Retorna la fecha local correspondiente a la marca de tiempo POSIX, tal como la retorna `time.time()`.

Esto puede generar `OverflowError`, si la marca de tiempo está fuera del rango de valores admitidos por la plataforma C `localtime()`, y `OSError` en `localtime()` falla. Es común que esto se restrinja a años desde 1970 hasta 2038. Tenga en cuenta que en los sistemas que no son POSIX que incluyen segundos bisiestos en su noción de marca de tiempo, los segundos bisiestos son ignorados por `fromtimestamp()`.

Distinto en la versión 3.3: Se genera `OverflowError` en lugar de `ValueError` si la marca de tiempo está fuera del rango de valores admitidos por la plataforma C `localtime()`. Se genera `OSError` en lugar de `ValueError` cuando `localtime()` falla.

classmethod `date.fromordinal(ordinal)`

Retorna la fecha correspondiente al ordinal gregoriano proléptico, donde el 1 de enero del año 1 tiene el ordinal 1.

`ValueError` se genera a menos que `1 <= ordinal <= date.max.toordinal().()`. Para cualquier fecha *d*, `date.fromordinal(d.toordinal()) == d`.

classmethod `date.fromisoformat(date_string)`

Retorna `date` correspondiente a una `date_string` dada en el formato YYYY-MM-DD:

```
>>> from datetime import date
>>> date.fromisoformat('2019-12-04')
datetime.date(2019, 12, 4)
```

Este es el inverso de `date.isoformat()`. Solo admite el formato AAAA-MM-DD.

Nuevo en la versión 3.7.

classmethod `date.fromisocalendar(year, week, day)`

Retorna `date` correspondiente a la fecha del calendario ISO especificada por año, semana y día. Esta es la inversa de la función `date.isocalendar()`.

Nuevo en la versión 3.8.

Atributos de clase:

`date.min`

La fecha representable más antigua, `date(MINYEAR, 1, 1)`.

² Esto coincide con la definición del calendario «proléptico gregoriano» en el libro de *Dershowitz y Reingold Cálculos calendáricos*, donde es el calendario base para todos los cálculos. Consulte el libro sobre algoritmos para convertir entre ordinales gregorianos prolépticos y muchos otros sistemas de calendario.

`date.max`

La última fecha representable, `date(MAXYEAR, 12, 31)`.

`date.resolution`

La menor diferencia entre objetos de fecha no iguales, `timedelta(days=1)`.

Atributos de instancia (solo lectura):

`date.year`

Entre `MINYEAR` y `MAXYEAR` inclusive.

`date.month`

Entre 1 y 12 inclusive.

`date.day`

Entre 1 y el número de días en el mes dado del año dado.

Operaciones soportadas:

Operación	Resultado
<code>date2 = date1 + timedelta</code>	<code>date2</code> es <code>timedelta.days</code> días eliminados de <code>date1</code> . (1)
<code>date2 = date1 - timedelta</code>	Calcula <code>date2</code> tal que <code>date2 + timedelta == date1</code> . (2)
<code>timedelta = date1 - date2</code>	(3)
<code>date1 < date2</code>	<code>date1</code> se considera menor que <code>date2</code> cuando <code>date1</code> precede a <code>date2</code> en el tiempo. (4)

Notas:

- (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) `timedelta.seconds` y `timedelta.microseconds` son ignorados.
- (3) Esto es exacto y no puede desbordarse. `timedelta.seconds` y `timedelta.microseconds` son 0, y `date2 + timedelta == date1`.
- (4) En otras palabras, `date1 < date2` si y solo si `date1.toordinal() < date2.toordinal()`. La comparación de fechas plantea `TypeError` si el otro elemento comparado no es también un objeto `date`. Sin embargo, se retorna `NotImplemented` si el otro elemento comparado tiene un atributo `timetuple()`. Este enlace ofrece a otros tipos de objetos de fecha la posibilidad de implementar una comparación de tipos mixtos. Si no, cuando un objeto `date` se compara con un objeto de un tipo diferente, `TypeError` se genera a menos que la comparación sea `==` or `!=`. Los últimos casos retorna `False` o `True`, respectivamente.

En contextos booleanos, todos los objetos `date` se consideran verdaderos.

Métodos de instancia:

`date.replace(year=self.year, month=self.month, day=self.day)`

Retorna una fecha con el mismo valor, a excepción de aquellos parámetros dados nuevos valores por cualquier argumento de palabra clave especificado.

Ejemplo:

```
>>> from datetime import date
>>> d = date(2002, 12, 31)
>>> d.replace(day=26)
datetime.date(2002, 12, 26)
```

`date.timetuple()`

Retorna una `time.struct_time` como la que retorna `time.localtime()`.

Las horas, minutos y segundos son 0, y el indicador DST es -1.

`d.timetuple()` es equivalente a:

```
time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))
```

donde `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` es el número de día dentro del año actual que comienza con 1 para el 1 de enero.

`date.toordinal()`

Retorna el ordinal gregoriano proléptico de la fecha, donde el 1 de enero del año 1 tiene el ordinal 1. Para cualquiera `date` object `d`, `date.fromordinal(d.toordinal()) == d`.

`date.weekday()`

Retorna el día de la semana como un número entero, donde el lunes es 0 y el domingo es 6. Por ejemplo, `date(2002, 12, 4).weekday() == 2`, un miércoles. Ver también `isoweekday()`.

`date.isoweekday()`

Retorna el día de la semana como un número entero, donde el lunes es 1 y el domingo es 7. Por ejemplo, `date(2002, 12, 4).isoweekday() == 3`, un miércoles. Ver también `weekday()`, `isocalendar()`.

`date.isocalendar()`

Retorna una 3-tupla, (año ISO, número de semana ISO, día de la semana ISO).

El calendario ISO es una variante amplia utilizada del calendario gregoriano.³

El año ISO consta de 52 o 53 semanas completas, y donde una semana comienza un lunes y termina un domingo. La primera semana de un año ISO es la primera semana calendario (gregoriana) de un año que contiene un jueves. Esto se llama semana número 1, y el año ISO de ese jueves es el mismo que el año gregoriano.

Por ejemplo, 2004 comienza en jueves, por lo que la primera semana del año ISO 2004 comienza el lunes 29 de diciembre de 2003 y termina el domingo 4 de enero de 2004

```
>>> from datetime import date
>>> date(2003, 12, 29).isocalendar()
(2004, 1, 1)
>>> date(2004, 1, 4).isocalendar()
(2004, 1, 7)
```

`date.isoformat()`

Retorna una cadena de caracteres que representa la fecha en formato ISO 8601, AAAA-MM-DD:

```
>>> from datetime import date
>>> date(2002, 12, 4).isoformat()
'2002-12-04'
```

Este es el inverso de `date.fromisoformat()`.

`date.__str__()`

Para una fecha `d`, `str(d)` es equivalente a `d.isoformat()`.

`date.ctime()`

Retorna una cadena de caracteres que representa la fecha:

```
>>> from datetime import date
>>> date(2002, 12, 4).ctime()
'Wed Dec 4 00:00:00 2002'
```

`d.ctime()` es equivalente a:

³ Consulte la guía de *R. H. van Gent's guide to the mathematics of the ISO 8601 calendar* para una buena explicación.

```
time.ctime(time.mktime(d.timetuple()))
```

en plataformas donde la función nativa C `ctime()` (donde `time.ctime()` llama, pero que `date.ctime()` no se llama) se ajusta al estándar C.

`date.strftime(format)`

Retorna una cadena de caracteres que representa la fecha, controlada por una cadena de formato explícito. Los códigos de formato que se refieren a horas, minutos o segundos verán valores 0. Para obtener una lista completa de las directivas de formato, consulte *Comportamiento strftime() y strptime()*.

`date.__format__(format)`

Igual que `date.strftime()`. Esto hace posible especificar una cadena de formato para un objeto `date` en literales de cadena con formato y cuando se usa `str.format()`. Para obtener una lista completa de las directivas de formato, consulte *Comportamiento strftime() y strptime()*.

Ejemplos de uso: date

Ejemplo de contar días para un evento:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
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
```

Más ejemplos de trabajo con `date`:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)

>>> # Methods related to formatting string output
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%Y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> d.ctime()
'Mon Mar 11 00:00:00 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'

>>> # Methods for to extracting 'components' under different calendars
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002          # year
3             # month
11            # day
```

(continué en la próxima página)

(proviene de la página anterior)

```

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 )

>>> # A date object is immutable; all operations produce a new object
>>> d.replace(year=2005)
datetime.date(2005, 3, 11)

```

8.1.6 Objetos `datetime`

El objeto `datetime` es un único objeto que contiene toda la información de un objeto `date` y un objeto `time`.

Como un objeto `date`, `datetime` asume el calendario gregoriano actual extendido en ambas direcciones; como un objeto `time`, `datetime` supone que hay exactamente 3600*24 segundos en cada día.

Constructor:

class `datetime.datetime` (*year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0*)

Se requieren los argumentos *year*, *month* y *day*. *tzinfo* puede ser `None`, o una instancia de una subclase `tzinfo`. Los argumentos restantes deben ser enteros en los siguientes rangos:

- `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`,
- `fold` in `[0, 1]`.

Si se proporciona un argumento fuera de esos rangos, `ValueError` se genera.

Nuevo en la versión 3.6: Se agregó el argumento `fold`.

Otros constructores, todos los métodos de clase:

classmethod `datetime.today()`
Retorna la fecha y hora local actual, con *tzinfo* `None`.

Equivalente a:

```
datetime.fromtimestamp(time.time())
```

Ver también `now()`, `fromtimestamp()`.

Este método es funciona como `now()`, pero sin un parámetro `tz`.

classmethod `datetime.now(tz=None)`

Retorna la fecha y hora local actual.

Si el argumento opcional `tz` es `None` o no se especifica, es como `today()`, pero, si es posible, proporciona más precisión de la que se puede obtener al pasar por `time.time()` marca de tiempo (por ejemplo, esto puede ser posible en plataformas que suministran la función C `gettimeofday()`).

Si `tz` no es `None`, debe ser una instancia de una subclase `tzinfo`, y la fecha y hora actuales se convierten en la zona horaria de `tz`.

Esta función es preferible a `today()` y `utcnow()`.

classmethod `datetime.utcnow()`

Retorna la fecha y hora UTC actual, con `tzinfo` `None`.

Esto es como `now()`, pero retorna la fecha y hora UTC actual, como un objeto naïf (*naive*): `datetime`. Se puede obtener una fecha y hora UTC actual consciente (*aware*) llamando a `datetime.now(tz=timezone.utc)`. Ver también `now()`.

Advertencia: Debido a que los objetos naïfs (*naive*) de `datetime` son tratados por muchos métodos de `datetime` como horas locales, se prefiere usar fechas y horas conscientes (*aware*) para representar las horas en UTC. Como tal, la forma recomendada de crear un objeto que represente la hora actual en UTC es llamando a `datetime.now(tz=timezone.utc)`.

classmethod `datetime.fromtimestamp(timestamp, tz=None)`

Retorna la fecha y hora local correspondiente a la marca de tiempo POSIX, tal como la retorna `time.time()`. Si el argumento opcional `tz` es `None` o no se especifica, la marca de tiempo se convierte a la fecha y hora local de la plataforma, y el objeto retornado `datetime` es naïf (*naive*).

Si `tz` no es `None`, debe ser una instancia de una subclase `tzinfo`, y la fecha y hora actuales se convierten en la zona horaria de `tz`.

`fromtimestamp()` puede aumentar `OverflowError`, si la marca de tiempo está fuera del rango de valores admitidos por la plataforma C `localtime()` o `gmtime()`, y `OSError` en `localtime()` o `gmtime()` falla. Es común que esto se restrinja a los años 1970 a 2038. Tenga en cuenta que en los sistemas que no son POSIX que incluyen segundos bisiestos en su noción de marca de tiempo, los segundos bisiestos son ignorados por `fromtimestamp()`, y luego es posible tener dos marcas de tiempo que difieren en un segundo que producen objetos idénticos `datetime`. Se prefiere este método sobre `utcfromtimestamp()`.

Distinto en la versión 3.3: Se genera `OverflowError` en lugar de `ValueError` si la marca de tiempo está fuera del rango de valores admitidos por la plataforma C `localtime()` o `gmtime()`. genera `OSError` en lugar de la función `ValueError` en `localtime()` o error `gmtime()`.

Distinto en la versión 3.6: `fromtimestamp()` puede retornar instancias con `fold` establecido en 1.

classmethod `datetime.utcfromtimestamp(timestamp)`

Retorna el UTC `datetime` correspondiente a la marca de tiempo POSIX, con `tzinfo` `None`. (El objeto resultante es naïf (*naive*).)

Esto puede generar `OverflowError`, si la marca de tiempo está fuera del rango de valores admitidos por la plataforma C `gmtime()`, y error en `OSError` en `gmtime()`. Es común que esto se restrinja a los años entre 1970 a 2038.

Para conocer un objeto `datetime`, llama a `fromtimestamp()`:

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

En las plataformas compatibles con POSIX, es equivalente a la siguiente expresión:

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

excepto que la última fórmula siempre admite el rango de años completo: entre `MINYEAR` y `MAXYEAR` inclusive.

Advertencia: Debido a que los objetos naïf (*naïve*) de `datetime` son tratados por muchos métodos de `datetime` como horas locales, se prefiere usar fechas y horas conscientes para representar las horas en UTC. Como tal, la forma recomendada de crear un objeto que represente una marca de tiempo específica en UTC es llamando a `datetime.fromtimestamp(timestamp, tz=timzone.utc)`.

Distinto en la versión 3.3: Se genera `OverflowError` en lugar de `ValueError` si la marca de tiempo está fuera del rango de valores admitidos por la plataforma C `gmtime()`. genera `OSError` en lugar de `ValueError` en el error de `gmtime()`.

classmethod `datetime.fromordinal(ordinal)`

Se genera `datetime` correspondiente al ordinal del proleptico gregoriano, donde el 1 de enero del año 1 tiene ordinal 1. `ValueError` se genera a menos que $1 \leq \text{ordinal} \leq \text{datetime.max.toordinal}()$. La hora, minuto, segundo y microsegundo del resultado son todos 0, y `tzinfo` es `None`.

classmethod `datetime.combine(date, time, tzinfo=self.tzinfo)`

Se genera un nuevo objeto `datetime` cuyos componentes de fecha son iguales a los dados en el objeto `date`, y cuyos componentes de tiempo son iguales a los dados en el objeto `time`. Si se proporciona el argumento `tzinfo`, su valor se usa para establecer el atributo `tzinfo` del resultado; de lo contrario, se usa el atributo `tzinfo` del argumento `time`.

Para cualquier objeto de `datetime` `d`, `d == datetime.combine(d.date(), d.time(), d.tzinfo)`. Si la fecha es un objeto de `datetime`, se ignoran sus componentes de tiempo y `tzinfo`.

Distinto en la versión 3.6: Se agregó el argumento `tzinfo`.

classmethod `datetime.fromisoformat(date_string)`

Retorna `datetime` correspondiente a `date_string` en uno de los formatos emitidos por `date.isoformat()` y `datetime.isoformat()`.

Específicamente, esta función admite cadenas de caracteres en el formato:

```
YYYY-MM-DD[*HH[:MM[:SS[.fff[fff]]]]][+HH:MM[:SS[.ffffff]]]]
```

donde `*` puede coincidir con cualquier carácter individual.

Prudencia: Esto *no* admite el *parsing* de cadenas de caracteres arbitrarias ISO 8601; solo está pensando cómo la operación inversa de `datetime.isoformat()`. Un *parseador* ISO 8601 mas completo, `dateutil.parser.isoparse` está disponible en el paquete de terceros `dateutil`.

Ejemplos:

```
>>> from datetime import datetime
>>> datetime.fromisoformat('2011-11-04')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('2011-11-04T00:05:23')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283+00:00')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000, tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('2011-11-04T00:05:23+04:00')
datetime.datetime(2011, 11, 4, 0, 5, 23,
    tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
```

Nuevo en la versión 3.7.

classmethod `datetime.fromisocalendar(year, week, day)`

Retorna `datetime` correspondiente a la fecha del calendario ISO especificada por año, semana y día. Los componentes que no son de fecha de fecha y hora se rellenan con sus valores predeterminados normales. Esta es la inversa de la función `datetime.isocalendar()`.

Nuevo en la versión 3.8.

classmethod `datetime.strptime(date_string, format)`

Retorna *datetime* correspondiente a *date_string*, analizado según *format*.

Esto es equivalente a:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

Se genera *ValueError* se genera si *date_string* y el formato no pueden ser analizados por *time.strptime()* o si retorna un valor que no es una tupla de tiempo. Para obtener una lista completa de las directivas de formato, consulte *Comportamiento strftime()* y *strptime()*.

Atributos de clase:

`datetime.min`

La primera fecha representable *datetime*, `datetime(MINYEAR, 1, 1, tzinfo=None)`.

`datetime.max`

La última fecha representable *datetime*, `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`.

`datetime.resolution`

La diferencia más pequeña posible entre objetos no iguales *datetime*, `timedelta(microseconds=1)`.

Atributos de instancia (solo lectura):

`datetime.year`

Entre *MINYEAR* y *MAXYEAR* inclusive.

`datetime.month`

Entre 1 y 12 inclusive.

`datetime.day`

Entre 1 y el número de días en el mes dado del año dado.

`datetime.hour`

En range(24).

`datetime.minute`

En range(60).

`datetime.second`

En range(60).

`datetime.microsecond`

En range(1000000).

`datetime.tzinfo`

El objeto pasó como argumento *tzinfo* al constructor *datetime*, o None si no se pasó ninguno.

`datetime.fold`

En [0, 1]. Se usa para desambiguar los tiempos de pared durante un intervalo repetido. (Se produce un intervalo repetido cuando los relojes se retrotraen al final del horario de verano o cuando el desplazamiento UTC para la zona actual se reduce por razones políticas). El valor 0 (1) representa el anterior (posterior) de los dos momentos con la misma representación de tiempo real transcurrido (*wall time*).

Nuevo en la versión 3.6.

Operaciones soportadas:

Operación	Resultado
<code>datetime2 = datetime1 + timedelta</code>	(1)
<code>datetime2 = datetime1 - timedelta</code>	(2)
<code>timedelta = datetime1 - datetime2</code>	(3)
<code>datetime1 < datetime2</code>	Compara <i>datetime</i> to <i>datetime</i> . (4)

- (1) `datetime2` es una duración de `timedelta` eliminada de `datetime1`, avanzando en el tiempo si `timedelta.days > 0`, o hacia atrás si `timedelta.days < 0`. El resultado tiene el mismo `tzinfo` como el atributo `datetime` y `datetime2 - datetime1 == timedelta`. `OverflowError` se genera si `datetime2.year` sería menor que `MINYEAR` o mayor que `MAXYEAR`. Tenga en cuenta que no se realizan ajustes de zona horaria, incluso si la entrada es un objeto consciente.
- (2) Calcula el `datetime` tal que `datetime2 + timedelta == datetime1`. En cuanto a la adición, el resultado tiene el mismo atributo `tzinfo` que la fecha y hora de entrada, y no se realizan ajustes de zona horaria, incluso si la entrada es consciente.
- (3) La resta de `datetime` de la `datetime` se define solo si ambos operandos son naïf(*naive*), o si ambos son conscientes(*aware*). Si uno es consciente y el otro es naïf, `TypeError` aparece.

Si ambos son naïf (*naive*), o ambos son conscientes (*aware*) y tienen el mismo atributo `tzinfo`, los atributos `tzinfo` se ignoran y el resultado es un objeto de `timedelta` `*t* tal que datetime2 + t == datetime1. No se realizan ajustes de zona horaria en este caso.`

Si ambos son conscientes (*aware*) y tienen atributos diferentes `tzinfo`, `a-b` actúa como si primero `a` y `b` se convirtieran primero en fechas naïf (*naive*) UTC. El resultado es `(a.replace(tzinfo = None) - a.utcoffset()) - (b.replace(tzinfo = None) - b.utcoffset())` excepto que la implementación nunca se desborda.

- (4) `datetime1` se considera menor que `datetime2` cuando `datetime1` precede `datetime2` en el tiempo.

Si uno de los elementos comparados es naïf (*naive*) y el otro lo sabe, se genera un `TypeError` si se intenta una comparación de órdenes. Para las comparaciones de igualdad, las instancias naïf nunca son iguales a las instancias conscientes (*aware*).

Si ambos comparados son conscientes (*aware*) y tienen el mismo atributo `tzinfo`, el atributo común `tzinfo` se ignora y se comparan las fechas base. Si ambos elementos comparados son conscientes y tienen atributos diferentes `tzinfo`, los elementos comparados se ajustan primero restando sus compensaciones UTC (obtenidas de `self.utcoffset()`).

Distinto en la versión 3.3: Las comparaciones de igualdad entre las instancias conscientes (*aware*) y naïf (*naive*) `datetime` no generan `TypeError`.

Nota: Para evitar que la comparación vuelva al esquema predeterminado de comparación de direcciones de objetos, la comparación de fecha y hora normalmente genera `TypeError` si el otro elemento comparado no es también un objeto `datetime`. Sin embargo, `NotImplemented` se retorna si el otro elemento comparado tiene un atributo `timetuple()`. Este enlace ofrece a otros tipos de objetos de fecha la posibilidad de implementar una comparación de tipos mixtos. Si no, cuándo un objeto `datetime` se compara con un objeto de un tipo diferente, se genera `TypeError` a menos que la comparación sea `==` o `!=`. Los últimos casos retornan `False` o `True`, respectivamente.

Métodos de instancia:

`datetime.date()`

Retorna el objeto `date` con el mismo año, mes y día.

`datetime.time()`

Retorna el objeto `time` con la misma hora, minuto, segundo, microsegundo y doblado(*fold*). `tzinfo` es `None`. Ver también método `timetz()`.

Distinto en la versión 3.6: El valor de plegado (*fold value*) se copia en el objeto `time` retornado.

`datetime.timetz()`

Retorna el objeto `time` con los mismos atributos de hora, minuto, segundo, microsegundo, pliegue y `tzinfo`. Ver también método `time()`.

Distinto en la versión 3.6: El valor de plegado (*fold value*) se copia en el objeto `time` retornado.

`datetime.replace(year=self.year, month=self.month, day=self.day, hour=self.hour, minute=self.minute, second=self.second, microsecond=self.microsecond, tzinfo=self.tzinfo, *, fold=0)`

Retorna una fecha y hora con los mismos atributos, a excepción de aquellos atributos a los que se les asignan

nuevos valores según los argumentos de palabras clave especificados. Tenga en cuenta que `tzinfo = None` se puede especificar para crear una fecha y hora naíf (*naive*) a partir de una fecha y hora consciente (*aware*) sin conversión de datos de fecha y hora.

Nuevo en la versión 3.6: Se agregó el argumento `fold`.

`datetime.astimezone(tz=None)`

Retorna un objeto `datetime` con el atributo nuevo `tzinfo tz`, ajustando los datos de fecha y hora para que el resultado sea la misma hora UTC que `self`, pero en hora local `tz`.

Si se proporciona, `tz` debe ser una instancia de una subclase `tzinfo`, y sus métodos `utcoffset()` y `dst()` no deben retornar `None`. Si `self` es naíf, se supone que representa la hora en la zona horaria del sistema.

Si se llama sin argumentos (o con `tz=None`), se asume la zona horaria local del sistema para la zona horaria objetivo. El atributo `.tzinfo` de la instancia de fecha y hora convertida se establecerá en una instancia de `timezone` con el nombre de zona y el desplazamiento obtenido del sistema operativo.

Si `self.tzinfo` es `tz`, `self.astimezone(tz)` es igual a `self`: no se realiza ningún ajuste de datos de fecha u hora. De lo contrario, el resultado es la hora local en la zona horaria `tz`, que representa la misma hora UTC que `self`: después de `astz = dt.astimezone(tz)`, `astz - astz.utcoffset()` tendrá los mismos datos de fecha y hora que `dt - dt.utcoffset()`.

Si simplemente desea adjuntar un objeto de zona horaria `tz` a una fecha y hora `dt` sin ajustar los datos de fecha y hora, use `dt.replace(tzinfo = tz)`. Si simplemente desea eliminar el objeto de zona horaria de una fecha y hora `dt` sin conversión de datos de fecha y hora, use `dt.replace(tzinfo = None)`.

Tenga en cuenta que el método predeterminado `tzinfo.fromutc()` se puede reemplazar en una subclase `tzinfo` para afectar el resultado retornado por `astimezone()`. `astimezone()` ignora los casos de error, actúa como:

```
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)
```

Distinto en la versión 3.3: `tz` ahora puede ser omitido.

Distinto en la versión 3.6: El método `astimezone()` ahora se puede invocar en instancias naíf (*naive*) que se supone representan la hora local del sistema.

`datetime.utcoffset()`

Si `tzinfo` es `None`, retorna `None`, de lo contrario retorna `self.tzinfo.utcoffset(self)`, y genera una excepción si este último no retorna `None` o un objeto `timedelta` con magnitud inferior a un día.

Distinto en la versión 3.7: El desfase UTC no está restringido a un número entero de minutos.

`datetime.dst()`

Si `tzinfo` es `None`, retorna `None`, de lo contrario retorna `self.tzinfo.utcoffset(self)`, y genera una excepción si este último no retorna `None` o un objeto `timedelta` con magnitud inferior a un día.

Distinto en la versión 3.7: El desfase DST no está restringido a un número entero de minutos.

`datetime.tzname()`

Si `tzinfo` es `None`, retorna `None`, de lo contrario retorna `self.tzinfo.tzname(self)`, genera una excepción si este último no retorna `None` o un objeto de cadena de caracteres,

`datetime.timetuple()`

Retorna una `time.struct_time` como la que retorna `time.localtime()`.

`d.timetuple()` es equivalente a:

```
time.struct_time((d.year, d.month, d.day,
                  d.hour, d.minute, d.second,
                  d.weekday(), yday, dst))
```

donde `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` es el número de día dentro del año actual que comienza con 1 para el 1 de enero. El indicador `tm_isdst` del resultado se establece de acuerdo con el método `dst()`: `tzinfo` es `None` o `dst()` retorna `None`, `tm_isdst` se establece en `-1`; si no `dst()` retorna un valor distinto de cero, `tm_isdst` se establece en `1`; de lo contrario `tm_isdst` se establece en `0`.

`datetime.utctimetuple()`

Si `datetime` instancia `d` es naïf (*naive*), esto es lo mismo que `d.timetuple()` excepto que `tm_isdst` se fuerza a `0` independientemente de lo que retorna `d.dst()`. El horario de verano nunca está en vigencia durante un horario UTC.

Si `d` es consciente (*aware*), `d` se normaliza a la hora UTC, restando `d.utcoffset()`, y se retorna `time.struct_time` para la hora normalizada. `tm_isdst` se fuerza a `0`. Tenga en cuenta que un `OverflowError` puede aparecer si `d*.year` fue `MINYEAR` o `MAXYEAR` y el ajuste UTC se derrama durante el límite de un año.

Advertencia: Debido a que los objetos naïf (*naive*) de `datetime` son tratados por muchos métodos de `datetime` como horas locales, se prefiere usar fechas y horas conscientes (*aware*) para representar las horas en UTC; como resultado, el uso de `utctimetuple` puede dar resultados engañosos. Si tiene una `datetime` naïf que representa UTC, use `datetime.replace(tzinfo=timezone.utc)` para que sea consciente, en cuyo punto se puede usar `datetime.timetuple()`.

`datetime.toordinal()`

Retorna el ordinal gregoriano proléptico de la fecha. Lo mismo que `self.date().toordinal()`.

`datetime.timestamp()`

Retorna la marca de tiempo (*timestamp*) POSIX correspondiente a la instancia `datetime`. El valor de retorno es `float` similar al retornado por `time.time()`.

Naive `datetime` instances are assumed to represent local time and this method relies on the platform `C mktime()` function to perform the conversion. Since `datetime` supports wider range of values than `mktime()` on many platforms, this method may raise `OverflowError` for times far in the past or far in the future.

Para las instancias de `datetime`, el valor de retorno se calcula como:

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

Nuevo en la versión 3.3.

Distinto en la versión 3.6: El método `timestamp()` utiliza el atributo `fold` para desambiguar los tiempos durante un intervalo repetido.

Nota: No hay ningún método para obtener la marca de tiempo (*timestamp*) POSIX directamente de una instancia naïf `datetime` que representa la hora UTC. Si su aplicación utiliza esta convención y la zona horaria de su sistema no está configurada en UTC, puede obtener la marca de tiempo POSIX al proporcionar `tzinfo = timezone.utc`:

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

o calculando la marca de tiempo (*timestamp*) directamente:

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```


`datetime.weekday()`

Retorna el día de la semana como un entero, donde el lunes es 0 y el domingo es 6. Lo mismo que `self.date().weekday()`. Ver también `isoweekday()`.

`datetime.isoweekday()`

Retorna el día de la semana como un número entero, donde el lunes es 1 y el domingo es 7. Lo mismo que `self.date().isoweekday()`. Ver también `weekday()`, `isocalendar()`.

`datetime.isocalendar()`

Retorna una 3-tupla, (*ISO year*, *ISO week number*, *ISO weekday*). Lo mismo que `self.date().isocalendar()`.

`datetime.isoformat(sep='T', timespec='auto')`

Retorna una cadena de caracteres representando la fecha y la hora en formato ISO 8601:

- `YYYY-MM-DDTHH:MM:SS.ffffff`, si *microsecond* no es 0
- `YYYY-MM-DDTHH:MM:SS`, si *microsecond* es 0

Si `utcoffset()` no retorna None, se agrega una cadena de caracteres dando el desplazamiento UTC:

- `YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`, si *microsecond* no es 0
- `YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.ffffff]]`, si *microsecond* es 0

Ejemplos:

```
>>> from datetime import datetime, timezone
>>> datetime(2019, 5, 18, 15, 17, 8, 132263).isoformat()
'2019-05-18T15:17:08.132263'
>>> datetime(2019, 5, 18, 15, 17, tzinfo=timezone.utc).isoformat()
'2019-05-18T15:17:00+00:00'
```

El argumento opcional *sep* (default 'T') es un separador de un carácter, ubicado entre las porciones de fecha y hora del resultado. Por ejemplo:

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     """A time zone with an arbitrary, constant -06:39 offset."""
...     def utcoffset(self, dt):
...         return timedelta(hours=-6, minutes=-39)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
>>> datetime(2009, 11, 27, microsecond=100, tzinfo=TZ()).isoformat()
'2009-11-27T00:00:00.000100-06:39'
```

El argumento opcional *timespec* especifica el número de componentes adicionales del tiempo a incluir (el valor predeterminado es 'auto'). Puede ser uno de los siguientes:

- 'auto': Igual que 'seconds' si *microsecond* es 0, igual que 'microseconds' de lo contrario.
- 'hours': Incluye el *hour* en el formato de dos dígitos HH.
- 'minutes': Incluye *hour* y *minute* en formato "HH:MM".
- 'seconds': Incluye *hour*, *minute*, y *second* en formato HH:MM:SS.
- 'milliseconds': Incluye tiempo completo, pero trunca la segunda parte fraccionaria a milisegundos. Formato HH:MM:SS.sss.
- 'microseconds': Incluye tiempo completo en formato "HH:MM:SS.ffff".

Nota: Los componentes de tiempo excluidos están truncados, no redondeados.

`ValueError` generará un argumento inválido *timespec*:

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

Nuevo en la versión 3.6: Se agregó el argumento *timespec*.

`datetime.__str__()`

Para una instancia de la *datetime* *d*, `str(d)` es equivalente a `d.isoformat(' ')`.

`datetime.ctime()`

Retorna una cadena de caracteres que representa la fecha y la hora:

```
>>> from datetime import datetime
>>> datetime(2002, 12, 4, 20, 30, 40).ctime()
'Wed Dec  4 20:30:40 2002'
```

La cadena de salida *no* incluirá información de zona horaria, independientemente de si la entrada es consciente (*aware*) o naíf (*naive*).

`d.ctime()` es equivalente a:

```
time.ctime(time.mktime(d.timetuple()))
```

en plataformas donde la función nativa C `ctime()` (que `time.ctime()` invoca, pero que `datetime.ctime()` no invoca) se ajusta al estándar C.

`datetime.strftime(format)`

Retorna una cadena de caracteres que representa la fecha y la hora, controlada por una cadena de formato explícito. Para obtener una lista completa de las directivas de formato, consulte [Comportamiento strftime\(\)](#) y [strptime\(\)](#).

`datetime.__format__(format)`

Igual que `date.strftime()`. Esto hace posible especificar una cadena de formato para un objeto *date* en literales de cadena con formato y cuando se usa `str.format()`. Para obtener una lista completa de las directivas de formato, consulte [Comportamiento strftime\(\)](#) y [strptime\(\)](#).

Ejemplos de uso: datetime

Ejemplos de trabajo con objetos *datetime*:

```
>>> from datetime import datetime, date, time, timezone

>>> # 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()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043) # GMT +1
>>> datetime.now(timezone.utc)
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060, tzinfo=datetime.timezone.utc)

>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
```

(continué en la próxima página)

(proviene de la página anterior)

```

datetime.datetime(2006, 11, 21, 16, 30)

>>> # 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 a 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.'
```

El siguiente ejemplo define una subclase `tzinfo` que captura la información de zona horaria de *Kabul*, Afganistán, que utilizó +4 UTC hasta 1945 y +4:30 UTC a partir de entonces:

```

from datetime import timedelta, datetime, tzinfo, timezone

class KabulTz(tzinfo):
    # Kabul used +4 until 1945, when they moved to +4:30
    UTC_MOVE_DATE = datetime(1944, 12, 31, 20, tzinfo=timezone.utc)

    def utcoffset(self, dt):
        if dt.year < 1945:
            return timedelta(hours=4)
        elif (1945, 1, 1, 0, 0) <= dt.timetuple()[5] < (1945, 1, 1, 0, 30):
            # An ambiguous ("imaginary") half-hour range representing
            # a 'fold' in time due to the shift from +4 to +4:30.
            # If dt falls in the imaginary range, use fold to decide how
            # to resolve. See PEP495.
            return timedelta(hours=4, minutes=(30 if dt.fold else 0))
        else:
            return timedelta(hours=4, minutes=30)

    def fromutc(self, dt):
        # Follow same validations as in datetime.tzinfo
        if not isinstance(dt, datetime):
            raise TypeError("fromutc() requires a datetime argument")
        if dt.tzinfo is not self:
            raise ValueError("dt.tzinfo is not self")
```

(continué en la próxima página)

(proviene de la página anterior)

```

    # A custom implementation is required for fromutc as
    # the input to this function is a datetime with utc values
    # but with a tzinfo set to self.
    # See datetime.astimezone or fromtimestamp.
    if dt.replace(tzinfo=timezone.utc) >= self.UTC_MOVE_DATE:
        return dt + timedelta(hours=4, minutes=30)
    else:
        return dt + timedelta(hours=4)

def dst(self, dt):
    # Kabul does not observe daylight saving time.
    return timedelta(0)

def tzname(self, dt):
    if dt >= self.UTC_MOVE_DATE:
        return "+04:30"
    return "+04"

```

Uso de KabulTz desde arriba

```

>>> tz1 = KabulTz()

>>> # Datetime before the change
>>> dt1 = datetime(1900, 11, 21, 16, 30, tzinfo=tz1)
>>> print(dt1.utcoffset())
4:00:00

>>> # Datetime after the change
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=tz1)
>>> print(dt2.utcoffset())
4:30:00

>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(timezone.utc)
>>> dt3
datetime.datetime(2006, 6, 14, 8, 30, tzinfo=datetime.timezone.utc)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=KabulTz())
>>> dt2 == dt3
True

```

8.1.7 Objetos `time`

Un objeto `time` representa a una hora del día (local), independiente de cualquier día en particular, y está sujeto a ajustes a través de un objeto `tzinfo`.

class `datetime.time` (*hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0*)

Todos los argumentos son opcionales. `tzinfo` puede ser `None`, o una instancia de una subclase `tzinfo`. Los argumentos restantes deben ser enteros en los siguientes rangos:

- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold` in `[0, 1]`.

Si se proporciona un argumento fuera de esos rangos, se genera `ValueError`. Todo predeterminado a 0 excepto `tzinfo`, que por defecto es `None`.

Atributos de clase:

`time.min`

El primero representable `time`, “time (0, 0, 0, 0)“.

`time.max`

El último representable `time`, `time (23, 59, 59, 999999)`.

`time.resolution`

La diferencia más pequeña posible entre los objetos no iguales `time`, `timedelta (microseconds=1)`, aunque tenga en cuenta que la aritmética en objetos `time` no es compatible.

Atributos de instancia (solo lectura):

`time.hour`

En `range (24)`.

`time.minute`

En `range (60)`.

`time.second`

En `range (60)`.

`time.microsecond`

En `range (1000000)`.

`time.tzinfo`

El objeto pasado como argumento `tzinfo` al constructor de la clase `time`, o `None` si no se pasó ninguno.

`time.fold`

En `[0, 1]`. Se usa para desambiguar los tiempos de pared durante un intervalo repetido. (Se produce un intervalo repetido cuando los relojes se retrotraen al final del horario de verano o cuando el desplazamiento UTC para la zona actual se reduce por razones políticas). El valor 0 (1) representa el anterior (posterior) de los dos momentos con la misma representación de tiempo real transcurrido (*wall time*).

Nuevo en la versión 3.6.

Los objetos `time` admiten la comparación de `time` con `time`, donde *a* se considera menor que *b* cuando *a* precede a *b* en el tiempo. Si un elemento comparado es naíf (*naive*) y el otro lo sabe se genera `TypeError` si se intenta una comparación de orden. Para las comparaciones de igualdad, las instancias naíf nunca son iguales a las instancias conscientes (*aware*).

Si ambos elementos comparados son conscientes (*aware*) y tienen el mismo atributo `tzinfo`, el atributo común `tzinfo` se ignora y se comparan los tiempos base. Si ambos elementos comparados son conscientes y tienen atributos diferentes `tzinfo`, los elementos comparados se ajustan primero restando sus compensaciones UTC (obtenidas de `self.utcoffset()`). Para evitar que las comparaciones de tipos mixtos vuelvan a la comparación predeterminada por dirección de objeto, cuando un objeto `time` se compara con un objeto de un tipo diferente, se genera `TypeError` a menos que la comparación es `==` o `!=`. Los últimos casos retornan `False` o `True`, respectivamente.

Distinto en la versión 3.3: Las comparaciones de igualdad entre las instancias conscientes (*aware*) y naífs (*naive*) `time` no generan `TypeError`.

En contextos booleanos, un objeto `time` siempre se considera verdadero.

Distinto en la versión 3.5: Antes de Python 3.5, un objeto `time` se consideraba falso si representaba la medianoche en UTC. Este comportamiento se consideró oscuro y propenso a errores y se ha eliminado en Python 3.5. Ver [bpo-13936](#) para más detalles.

Otro constructor:

classmethod `time.fromisoformat (time_string)`

Retorna una `time` correspondiente a `time_string` en uno de los formatos emitidos por `time.isoformat ()`. Específicamente, esta función admite cadenas de caracteres en el formato:

`HH[:MM[:SS[.fff[fff]]]] [+HH:MM[:SS[.fffff]]]`

Prudencia: Esto *no* admite el *parsing* de cadenas arbitrarias ISO 8601. Solo pretende ser la operación inversa de `time.isoformat()`.

Ejemplos:

```
>>> from datetime import time
>>> time.fromisoformat('04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('04:23:01.000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01+04:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone(datetime.
↳timedelta(seconds=14400)))
```

Nuevo en la versión 3.7.

Métodos de instancia:

`time.replace(hour=self.hour, minute=self.minute, second=self.second, microsecond=self.microsecond, tzinfo=self.tzinfo, *, fold=0)`

Retorna una `time` con el mismo valor, excepto para aquellos atributos a los que se les otorgan nuevos valores según los argumentos de palabras clave especificados. Tenga en cuenta que `tzinfo = None` se puede especificar para crear una `time` naíf (*naive*) desde un consciente (*aware*) `time`, sin conversión de los datos de tiempo.

Nuevo en la versión 3.6: Se agregó el argumento `fold`.

`time.isoformat(timespec='auto')`

Retorna una cadena que representa la hora en formato ISO 8601, una de:

- HH:MM:SS.ffffff, si `microsecond` no es 0
- HH:MM:SS, si `microsecond` es 0
- HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]], si `utcoffset()` no retorna None
- HH:MM:SS+HH:MM[:SS[.ffffff]], si `microsecond` es 0 y `utcoffset()` no retorna None

El argumento opcional `timespec` especifica el número de componentes adicionales del tiempo a incluir (el valor predeterminado es `'auto'`). Puede ser uno de los siguientes:

- `'auto'`: Igual que `'seconds'` si `microsecond` es 0, igual que `'microseconds'` de lo contrario.
- `'hours'`: Incluye el `hour` en el formato de dos dígitos HH.
- `'minutes'`: Incluye `hour` y `minute` en formato “HH:MM”.
- `'seconds'`: Incluye `hour`, `minute`, y `second` en formato HH:MM:SS.
- `'milliseconds'`: Incluye tiempo completo, pero trunca la segunda parte fraccionaria a milisegundos. Formato HH:MM:SS.sss.
- `'microseconds'`: Incluye tiempo completo en formato “HH:MM:SS.ffffff”.

Nota: Los componentes de tiempo excluidos están truncados, no redondeados.

`ValueError` generará un argumento inválido `timespec`.

Ejemplo:

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec=
↳'minutes')
'12:34'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

Nuevo en la versión 3.6: Se agregó el argumento *timespec*.

`time.__str__()`

Durante un tiempo *t*, `str(t)` es equivalente a `t.isoformat()`.

`time.strftime(format)`

Retorna una cadena que representa la hora, controlada por una cadena de formato explícito. Para obtener una lista completa de las directivas de formato, consulte [Comportamiento strftime\(\)](#) y [strptime\(\)](#).

`time.__format__(format)`

Igual que `time.strftime()`. Esto permite especificar una cadena de formato para un objeto *time* en cadenas de caracteres literales con formato y cuando se usa `str.format()`. Para obtener una lista completa de las directivas de formato, consulte [Comportamiento strftime\(\)](#) y [strptime\(\)](#).

`time.utcoffset()`

Si *tzinfo* es *None*, retorna *None*, sino retorna `self.tzinfo.utcoffset()` (*None*), y genera una excepción si este último no retorna *None* o un objeto de *timedelta* con magnitud inferior a un día.

Distinto en la versión 3.7: El desfase UTC no está restringido a un número entero de minutos.

`time.dst()`

Si *tzinfo* es *None*, retorna *None*, sino retorna `self.tzinfo.utcoffset()` (*None*), y genera una excepción si este último no retorna *None*, o un objeto de *timedelta* con magnitud inferior a un día.

Distinto en la versión 3.7: El desfase DST no está restringido a un número entero de minutos.

`time.tzname()`

Si *tzinfo* es *None*, retorna *None*, sino retorna `self.tzinfo.tzname()` (*None*), o genera una excepción si este último no retorna *None* o un objeto de cadena.

Ejemplos de uso: time

Ejemplos de trabajo con el objeto *time*:

```
>>> from datetime import time, tzinfo, timedelta
>>> class TZ1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self, dt):
...         return "+01:00"
...     def __repr__(self):
...         return f"{self.__class__.__name__}()"
...
>>> t = time(12, 10, 30, tzinfo=TZ1())
>>> t
datetime.time(12, 10, 30, tzinfo=TZ1())
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'+01:00'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 +01:00'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 'The {} is {:%H:%M}'.format("time", t)
'The time is 12:10.'
```

8.1.8 Objetos `tzinfo`

`class datetime.tzinfo`

Esta es una clase base abstracta, lo que significa que esta clase no debe ser instanciada directamente. Defina una subclase de `tzinfo` para capturar información sobre una zona horaria particular.

Una instancia (de una subclase concreta) `tzinfo` se puede pasar a los constructores para objetos de `datetime` y `time`. Los últimos objetos ven sus atributos como en la hora local, y el objeto `tzinfo` admite métodos que revelan el desplazamiento de la hora local desde UTC, el nombre de la zona horaria y el desplazamiento DST, todo en relación con un objeto de fecha u hora pasó a ellos.

Debe derivar una subclase concreta y (al menos) proporcionar implementaciones de los métodos estándar `tzinfo` que necesitan los métodos `datetime` que utiliza. El módulo `datetime` proporciona `timezone`, una subclase concreta simple de `tzinfo` que puede representar zonas horarias con desplazamiento fijo desde UTC como UTC o Norte América EST y EDT.

Requisito especial para el *pickling*: La subclase `tzinfo` debe tener un método `__init__()` que se pueda invocar sin argumentos; de lo contrario, se puede hacer *pickling* pero posiblemente no se vuelva a despegar. Este es un requisito técnico que puede ser relajado en el futuro.

Una subclase concreta de `tzinfo` puede necesitar implementar los siguientes métodos. Exactamente qué métodos son necesarios depende de los usos de los objetos conscientes(*aware*) `datetime`. En caso de duda, simplemente implemente todos ellos.

`tzinfo.utcoffset(dt)`

Retorna el desplazamiento de la hora local desde UTC, como un objeto `timedelta` que es positivo al este de UTC. Si la hora local es al oeste de UTC, esto debería ser negativo.

Esto representa el desplazamiento *total* de UTC; por ejemplo, si un objeto `tzinfo` representa ajustes de zona horaria y DST, `utcoffset()` debería retornar su suma. Si no se conoce el desplazamiento UTC, retorna `None`. De lo contrario, el valor devuelto debe ser un objeto de `timedelta` estrictamente entre `-timedelta(hours = 24)` y `timedelta(hours = 24)` (la magnitud del desplazamiento debe ser inferior a un día) La mayoría de las implementaciones de `utcoffset()` probablemente se parecerán a una de estas dos:

```
return CONSTANT # fixed-offset class
return CONSTANT + self.dst(dt) # daylight-aware class
```

Si `utcoffset()` no retorna `None`, `dst()` no debería retornar `None` tampoco.

La implementación por defecto de `utcoffset()` genera `NotImplementedError`.

Distinto en la versión 3.7: El desfase UTC no está restringido a un número entero de minutos.

`tzinfo.dst(dt)`

Retorna el ajuste del horario de verano (DST), como un objeto de `timedelta` o `None` si no se conoce la información de DST.

Retorna `timedelta(0)` si el horario de verano no está en vigor. Si DST está en vigor, retorna el desplazamiento como un objeto `timedelta` (consulte `utcoffset()` para más detalles). Tenga en cuenta que el desplazamiento DST, si corresponde, ya se ha agregado al desplazamiento UTC retornado por `utcoffset()`, por lo que no es necesario consultar `dst()` a menos que esté interesado en obtener información DST por separado. Por ejemplo, `datetime.timetuple()` llama a su `tzinfo` del atributo `dst()` para determinar cómo se debe establecer el indicador `tm_isdst`, y `tzinfo.fromutc()` llama `dst()` para tener en cuenta los cambios de horario de verano al cruzar zonas horarias.

Una instancia `tz` de una subclase `tzinfo` que modela los horarios estándar y diurnos debe ser coherente en este sentido:

```
tz.utcoffset(dt) - tz.dst(dt)
```

debe retornar el mismo resultado para cada *datetime* *dt* con *dt.tzinfo == tz*. Para las subclases sanas *tzinfo*, esta expresión produce el «desplazamiento estándar» de la zona horaria, que no debe depender de la fecha o la hora, sino solo de la ubicación geográfica. La implementación de *datetime.astimezone()* se basa en esto, pero no puede detectar violaciones; es responsabilidad del programador asegurarlo. Si una subclase *tzinfo* no puede garantizar esto, puede anular la implementación predeterminada de *tzinfo.fromutc()* para que funcione correctamente con *astimezone()* independientemente.

La mayoría de las implementaciones de *dst()* probablemente se parecerán a una de estas dos:

```
def dst(self, dt):
    # a fixed-offset class: doesn't account for DST
    return timedelta(0)
```

o:

```
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.

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

La implementación predeterminada de *dst()* genera *NotImplementedError*.

Distinto en la versión 3.7: El desfase DST no está restringido a un número entero de minutos.

tzinfo.tzname(dt)

Retorna el nombre de zona horaria correspondiente al objeto *datetime* *dt*, como una cadena de caracteres. El módulo *datetime* no define nada sobre los nombres de cadena, y no hay ningún requisito de que signifique algo en particular. Por ejemplo, «GMT», «UTC», «-500», «-5:00», «EDT», «US/Eastern», «America/New York» son todas respuestas válidas. Retorna *None* si no se conoce un nombre de cadena. Tenga en cuenta que este es un método en lugar de una cadena fija principalmente porque algunas subclases *tzinfo* desearán retornar diferentes nombres dependiendo del valor específico de *dt* pasado, especialmente si la clase *tzinfo* es contable para el horario de verano.

La implementación predeterminada de *tzname()* genera *NotImplementedError*.

Estos métodos son llamados por un objeto *datetime* o *time*, en respuesta a sus métodos con los mismos nombres. El objeto de *datetime* se pasa a sí mismo como argumento, y un objeto *time* pasa a *None* como argumento. Los métodos de la subclase *tzinfo* deben, por lo tanto, estar preparados para aceptar un argumento *dt* de *None*, o de clase *datetime*.

Cuando se pasa *None*, corresponde al diseñador de la clase decidir la mejor respuesta. Por ejemplo, retornar *None* es apropiado si la clase desea decir que los objetos de tiempo no participan en los protocolos *tzinfo*. Puede ser más útil que *utcoffset(None)* retorne el desplazamiento UTC estándar, ya que no existe otra convención para descubrir el desplazamiento estándar.

Cuando se pasa un objeto *datetime* en respuesta a un método *datetime*, *dt.tzinfo* es el mismo objeto que *self.tzinfo*. Los métodos pueden confiar en esto, a menos que el código del usuario llame *tzinfo* métodos directamente. La intención es que los métodos *tzinfo* interpreten *dt* como si estuvieran en la hora local, y no necesiten preocuparse por los objetos en otras zonas horarias.

Hay un método más *tzinfo* que una subclase puede desear anular:

tzinfo.fromutc(dt)

Esto se llama desde la implementación predeterminada *datetime.astimezone()*. Cuando se llama desde eso, *dt.tzinfo* es *self*, y los datos de fecha y hora de *dt* deben considerarse como una hora UTC. El propósito de *fromutc()* es ajustar los datos de fecha y hora, retornando una fecha y hora equivalente en la hora local de *self*.

La mayoría de las subclases `tzinfo` deberían poder heredar la implementación predeterminada `fromutc()` sin problemas. Es lo suficientemente fuerte como para manejar zonas horarias de desplazamiento fijo y zonas horarias que representan tanto el horario estándar como el horario de verano, y esto último incluso si los tiempos de transición DST difieren en años diferentes. Un ejemplo de una zona horaria predeterminada `fromutc()`, la implementación puede que no se maneje correctamente en todos los casos es aquella en la que el desplazamiento estándar (desde UTC) depende de la fecha y hora específicas que pasan, lo que puede suceder por razones políticas. Las implementaciones predeterminadas de `astimezone()` y `fromutc()` pueden no producir el resultado que desea si el resultado es una de las horas a horcajadas en el momento en que cambia el desplazamiento estándar.

Código de omisión para casos de error, el valor predeterminado `fromutc()` la implementación actúa como

```
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
```

En el siguiente archivo `tzinfo_examples.py` hay algunos ejemplos de clases `tzinfo`:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)

# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
# timezones where UTC offset and/or the DST rules had
# changed in the past.)
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 fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond,
                        tzinfo=self, fold=fold)

    def utcoffset(self, dt):
```

(continué en la próxima página)

(proviene de la página anterior)

```

        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    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) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# 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) 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, 2)
# 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)
# 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

def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967, return

```

(continué en la próxima página)

(proviene de la página anterior)

```

# start = end for no DST.
if 2006 < year:
    dststart, dstend = DSTSTART_2007, DSTEND_2007
elif 1986 < year < 2007:
    dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
elif 1966 < year < 1987:
    dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
else:
    return (datetime(year, 1, 1), ) * 2

start = first_sunday_on_or_after(dststart.replace(year=year))
end = first_sunday_on_or_after(dstend.replace(year=year))
return start, end

```

```
class USTimeZone(tzinfo):
```

```

    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
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)
        if start + HOUR <= dt < end - HOUR:
            # DST is in effect.
            return HOUR
        if end - HOUR <= dt < end:
            # Fold (an ambiguous hour): use dt.fold to disambiguate.
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # Gap (a non-existent hour): reverse the fold rule.
            return HOUR if dt.fold else ZERO
        # DST is off.
        return ZERO

    def fromutc(self, dt):
        assert dt.tzinfo is self

```

(continué en la próxima página)

(proviene de la página anterior)

```

start, end = us_dst_range(dt.year)
start = start.replace(tzinfo=self)
end = end.replace(tzinfo=self)
std_time = dt + self.stdoftime
dst_time = std_time + HOUR
if end <= dst_time < end + HOUR:
    # Repeated hour
    return std_time.replace(fold=1)
if std_time < start or dst_time >= end:
    # Standard time
    return std_time
if start <= std_time < end - HOUR:
    # Daylight saving time
    return dst_time

Eastern = USTimeZone(-5, "Eastern", "EST", "EDT")
Central = USTimeZone(-6, "Central", "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific = USTimeZone(-8, "Pacific", "PST", "PDT")

```

Tenga en cuenta que hay sutilezas inevitables dos veces al año en una subclase `tzinfo` que representa tanto el horario estándar como el horario de verano, en los puntos de transición DST. Para mayor concreción, considere *US Eastern* (UTC -0500), donde EDT comienza el minuto después de 1:59 (EST) el segundo domingo de marzo y termina el minuto después de 1:59 (EDT) el primer domingo de noviembre

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

Cuando comienza el horario de verano (la línea de «inicio»), tiempo real transcurrido (*wall time*) salta de 1:59 a 3:00. Un tiempo de pared de la forma 2:MM realmente no tiene sentido ese día, por lo que `astimezone` (Eastern) no entregará un resultado con `hour == 2` el día en que comienza el horario de verano. Por ejemplo, en la transición de primavera de 2016, obtenemos

```

>>> from datetime import datetime, timezone
>>> from tzinfo_examples import HOUR, Eastern
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT

```

Cuando finaliza el horario de verano (la línea «final»), hay un problema potencialmente peor: hay una hora que no se puede deletrear sin ambigüedades en el tiempo de la pared local: la última hora del día. En el Este, esos son los tiempos de la forma 5:MM UTC en el día en que termina el horario de verano. El reloj de pared local salta de 1:59 (hora del día) a la 1:00 (hora estándar) nuevamente. Horas locales de la forma 1:MM son ambiguas. `astimezone()` imita el comportamiento del reloj local al mapear dos horas UTC adyacentes en la misma hora local. En el ejemplo oriental, los tiempos UTC de la forma 5:MM y 6:MM se correlacionan con 1:MM cuando se convierten en oriental, pero los tiempos anteriores tienen el atributo `fold` establecido en 0 y los tiempos posteriores configúrelo en 1. Por ejemplo, en la transición alternativa de 2016, obtenemos:

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

Tenga en cuenta que las instancias `datetime` que difieren solo por el valor del atributo `fold` se consideran iguales en las comparaciones.

Las aplicaciones que no pueden soportar ambigüedades de tiempo real (*wall time*) deben verificar explícitamente el valor del atributo `fold` o evitar el uso de las subclases híbridas `tzinfo`; no existen ambigüedades cuando se utiliza `timezone`, o cualquier otra subclase de clase offset `tzinfo` (como una clase que representa solo *EST* (desplazamiento fijo -5 horas), o solo EDT (desplazamiento fijo -4 horas)).

Ver también:

dateutil.tz El módulo `datetime` tiene una clase básica `timezone` (para manejar compensaciones fijas arbitrarias desde UTC) y su atributo `timezone.utc` (una instancia de zona horaria UTC).

La biblioteca `dateutil.tz` trae la *IANA timezone database* (también conocida como la base de datos *Olson*) a Python, y se recomienda su uso.

IANA timezone database La base de datos de zonas horarias (a menudo llamada *tz*, *tzdata* o *zoneinfo*) contiene código y datos que representan el historial de la hora local de muchos lugares representativos de todo el mundo. Se actualiza periódicamente para reflejar los cambios realizados por los cuerpos políticos en los límites de la zona horaria, las compensaciones UTC y las reglas de horario de verano.

8.1.9 Objetos `timezone`

La clase `timezone` es una subclase de `tzinfo`, cada una de las cuales representa una zona horaria definida por un desplazamiento fijo desde UTC.

Los objetos de esta clase no se pueden usar para representar la información de zona horaria en los lugares donde se usan diferentes desplazamientos en diferentes días del año o donde se han realizado cambios históricos en la hora civil.

class `datetime.timezone` (*offset*, *name=None*)

El argumento *offset* debe especificarse como un objeto de `timedelta` que representa la diferencia entre la hora local y UTC. Debe estar estrictamente entre `-timedelta(horas = 24)` y `timedelta(horas = 24)`, de lo contrario `ValueError` se genera.

El argumento *name* es opcional. Si se especifica, debe ser una cadena de caracteres que se utilizará como el valor retornado por el método `datetime.timezone.tzname()`.

Nuevo en la versión 3.2.

Distinto en la versión 3.7: El desfase UTC no está restringido a un número entero de minutos.

`timezone.utcoffset` (*dt*)

Retorna el valor fijo especificado cuando se construye la instancia `timezone`.

El argumento *dt* se ignora. El valor de retorno es una instancia de `timedelta` igual a la diferencia entre la hora local y UTC.

Distinto en la versión 3.7: El desfase UTC no está restringido a un número entero de minutos.

`timezone.tzname` (*dt*)

Retorna el valor fijo especificado cuando se construye la instancia `timezone`.

Si no se proporciona *name* en el constructor, el nombre retornado por `tzname(dt)` se genera a partir del valor del `offset` de la siguiente manera. Si *offset* es `timedelta(0)`, el nombre es «UTC», de lo contrario es una cadena en el formato `UTC ±`, donde \pm es el signo de `offset`, HH y MM son dos dígitos de `offset.hours` y `offset.minutes` respectivamente.

Distinto en la versión 3.6: El nombre generado a partir de `offset = timedelta(0)` ahora es simple “UTC”, no “UTC+00:00”.

`timezone.dst(dt)`

Siempre retorna `None`.

`timezone.fromutc(dt)`

Retorna `dt + offset`. El argumento *dt* debe ser una instancia consciente (*aware*) `datetime`, con *tzinfo* establecido en “self”.

Atributos de clase:

`timezone.utc`

La zona horaria UTC, `timezone(timedelta(0))`.

8.1.10 Comportamiento `strftime()` y `strptime()`

`date`, `datetime`, y `time` los objetos admiten un método `strftime(format)`, para crear una cadena que represente el tiempo bajo el control de una cadena de caracteres de formato explícito.

Por el contrario, el método de clase `datetime.strptime()` crea un objeto `datetime` a partir de una cadena que representa una fecha y hora y una cadena de formato correspondiente.

La siguiente tabla proporciona una comparación de alto nivel de `strftime()` versus `strptime()`:

	<code>strftime</code>	<code>strptime</code>
Uso	Convierte objetos en una cadena de caracteres de acuerdo con un formato dado	<i>parsear</i> una cadena en un objeto <code>datetime</code> con el formato correspondiente
Tipo de método	Método de instancia	Método de clase
Método de	<code>date</code> ; <code>datetime</code> ; <code>time</code>	<code>datetime</code>
Firma	<code>strftime(format)</code>	<code>strptime(date_string, format)</code>

Códigos de formato `strftime()` y `strptime()`

La siguiente es una lista de todos los códigos de formato que requiere el estándar 1989 C, y estos funcionan en todas las plataformas con una implementación estándar C.

Directiva	Significado	Ejemplo	Notas
%a	Día de la semana como nombre abreviado según la configuración regional.	<i>Sun, Mon, ..., Sat</i> (<i>en_US</i>); <i>So, Mo, ..., Sa</i> (<i>de_DE</i>)	(1)
%A	Día de la semana como nombre completo de la localidad.	<i>Sunday, Monday, ..., Saturday</i> (<i>en_US</i>); <i>Sonntag, Montag, ..., Samstag</i> (<i>de_DE</i>)	(1)
%w	Día de la semana como un número decimal, donde 0 es domingo y 6 es sábado.	0, 1, ..., 6	
%d	Día del mes como un número decimal rellenado con ceros.	01, 02, ..., 31	(9)
%b	Mes como nombre abreviado según la configuración regional.	<i>Jan, Feb, ..., Dec</i> (<i>en_US</i>); <i>Jan, Feb, ..., Dez</i> (<i>de_DE</i>)	(1)
%B	Mes como nombre completo según la configuración regional.	<i>January, February, ..., December</i> (<i>en_US</i>); <i>Januar, Februar, ..., Dezember</i> (<i>de_DE</i>)	(1)
%m	Mes como un número decimal rellenado con ceros.	01, 02, ..., 12	(9)
%y	Año sin siglo como un número decimal rellenado con ceros.	00, 01, ..., 99	(9)
%Y	Año con siglo como número decimal.	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(2)
%H	Hora (reloj de 24 horas) como un número decimal rellenado con ceros.	00, 01, ..., 23	(9)
%I	Hora (reloj de 12 horas) como un número decimal rellenado con ceros.	01, 02, ..., 12	(9)
%p	El equivalente de la configuración regional de AM o PM.	AM, PM (<i>en_US</i>); am, pm (<i>de_DE</i>)	(1), (3)
%M	Minuto como un número decimal rellenado con ceros.	00, 01, ..., 59	(9)
%S	Segundo como un número decimal rellenado con ceros.	00, 01, ..., 59	(4), (9)
%f	Microsegundo como un número decimal, rellenado con ceros a la izquierda.	000000, 000001, ..., 999999	(5)
210		Capítulo 8. Tipos de datos	
%z	Desplazamiento (offset) UTC en la forma +HHMM[SS[.SS]]	(vacío), +0000, -0400, +1030, +063415, -030712.345216	(6)

Se incluyen varias directivas adicionales no requeridas por el estándar C89 por conveniencia. Todos estos parámetros corresponden a valores de fecha ISO 8601.

Directiva	Significado	Ejemplo	Notas
%G	ISO 8601 year with century representing the year that contains the greater part of the ISO week (%V).	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(8)
%u	ISO 8601 weekday as a decimal number where 1 is Monday.	1, 2, ..., 7	
%V	ISO 8601 semana como un número decimal con lunes como primer día de la semana. La semana 01 es la semana que contiene el 4 de enero.	01, 02, ..., 53	(8), (9)

Es posible que no estén disponibles en todas las plataformas cuando se usan con el método `strptime()`. Las directivas ISO 8601 año e ISO 8601 semana no son intercambiables con las directivas de número de año y semana anteriores. Llamar a `strptime()` con directivas ISO 8601 incompletas o ambiguas generará un `ValueError`.

El conjunto completo de códigos de formato admitidos varía entre plataformas, porque Python llama a la función de la biblioteca C de la plataforma `strptime()`, y las variaciones de la plataforma son comunes. Para ver el conjunto completo de códigos de formato admitidos en su plataforma, consulte la documentación de `strptime(3)`.

Nuevo en la versión 3.6: %G, %u y %V fueron añadidos.

Detalle técnico

En términos generales, `d.strptime(fmt)` actúa como el módulo `time.time.strptime(fmt, d.timetuple())` aunque no todos los objetos admiten el método `timetuple()`.

Para el método de clase `datetime.strptime()`, el valor predeterminado es `1900-01-01T00:00:00.000`: cualquier componente no especificado en la cadena de formato se extraerá del valor predeterminado.⁴

Usar `datetime.strptime(date_string, format)` es equivalente a:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

excepto cuando el formato incluye componentes de sub-segundos o información de compensación de zona horaria, que son compatibles con `datetime.strptime` pero son descartados por `time.strptime`.

Para objetos de `time`, los códigos de formato para año, mes y día no deben usarse, ya que los objetos de `time` no tienen tales valores. Si se usan de todos modos, 1900 se sustituye por el año y 1 por el mes y el día.

Para los objetos `date`, los códigos de formato para horas, minutos, segundos y microsegundos no deben usarse, ya que los objetos `date` no tienen tales valores. Si se usan de todos modos, 0 se sustituye por ellos.

Por la misma razón, el manejo de cadenas de formato que contienen puntos de código Unicode que no se pueden representar en el conjunto de caracteres del entorno local actual también depende de la plataforma. En algunas plataformas, estos puntos de código se conservan intactos en la salida, mientras que en otros `strptime` puede generar `UnicodeError` o retornar una cadena vacía.

Notas:

- (1) Debido a que el formato depende de la configuración regional actual, se debe tener cuidado al hacer suposiciones sobre el valor de salida. Los ordenamientos de campo variarán (por ejemplo, «mes/día/año» versus «día/mes/año»), y la salida puede contener caracteres Unicode codificados utilizando la codificación predeterminada de la configuración regional (por ejemplo, si la configuración regional actual es `ja_JP`, la codificación predeterminada podría ser cualquiera de `eucJP`, “SJIS” o `utf-8`; use `locale.getlocale()` para determinar la codificación de la configuración regional actual).
- (2) El método `strptime()` puede analizar años en el rango completo [1, 9999], pero los años < 1000 deben llenarse desde cero hasta un ancho de 4 dígitos.

Distinto en la versión 3.2: En versiones anteriores, el método `strptime()` estaba restringido a años ≥ 1900 .

⁴ Si se pasa `datetime.strptime('29 de febrero', '%b %d')` fallará ya que 1900 no es un año bisiesto.

Distinto en la versión 3.3: En la versión 3.2, el método `strptime()` estaba restringido a años ≥ 1000 .

- (3) Cuando se usa con el método `strptime()`, la directiva `%p` solo afecta el campo de hora de salida si se usa la directiva `%I` para analizar la hora.
- (4) A diferencia del módulo `time`, el módulo `datetime` no admite segundos intercalares.
- (5) Cuando se usa con el método `strptime()`, la `%f` directiva acepta de uno a seis dígitos y cero *pads* a la derecha. `%f` es una extensión del conjunto de caracteres de formato en el estándar C (pero implementado por separado en objetos de fecha y hora y, por lo tanto, siempre disponible).
- (6) Para un objeto naíf (*naive*), los códigos de formato `%z` y `%Z` se reemplazan por cadenas vacías.

Para un objeto consciente (*aware*)

`%z` `utcoffset()` se transforma en una cadena de la forma `±HHMM[SS[.ffffff]]`, donde “HH” es una cadena de 2 dígitos que da el número de horas de desplazamiento UTC, “MM” es una cadena de 2 dígitos que da el número de minutos de desplazamiento UTC, “SS” es una cadena de 2 dígitos que da el número de segundos de desplazamiento UTC y `ffffff` es una cadena de 6 dígitos que da el número de microsegundos de desplazamiento UTC. La parte `ffffff` se omite cuando el desplazamiento es un número entero de segundos y tanto la parte `ffffff` como la parte `SS` se omiten cuando el desplazamiento es un número entero de minutos. Por ejemplo, si `utcoffset()` retorna `timedelta(hours=-3, minutes=-30)`, `%z` se reemplaza con la cadena `'-0330'`.

Distinto en la versión 3.7: El desfase UTC no está restringido a un número entero de minutos.

Distinto en la versión 3.7: Cuando la directiva `%z` se proporciona al método `strptime()`, las compensaciones UTC pueden tener dos puntos como separador entre horas, minutos y segundos. Por ejemplo, `'+01:00:00'` se analizará como una compensación de una hora. Además, proporcionar `'Z'` es idéntico a `'+00:00'`.

`%Z` Si `tzname()` retorna `None`, `%Z` se reemplaza por una cadena de caracteres vacía. De lo contrario, `%Z` se reemplaza por el valor retornado, que debe ser una cadena.

Distinto en la versión 3.2: Cuando la directiva `%z` se proporciona al método `strptime()`, se generará un objeto consciente `datetime`. El `tzinfo` del resultado se establecerá en una instancia `timezone`.

- (7) Cuando se usa con el método `strptime()`, `%U` y `%W` solo se usan en los cálculos cuando se especifican el día de la semana y el año calendario (`%Y`).
- (8) Similar a `%U` y `%W`, `%V` solo se usa en cálculos cuando el día de la semana y el año ISO (`%G`) se especifican en `strptime()` cadena de formato. También tenga en cuenta que `%G` y `%Y` no son intercambiables.
- (9) Cuando se usa con el método `strptime()`, el cero inicial es opcional para los formatos `%d`, `%m`, `%H`, `%I`, `%M`, `%S`, `%J`, `%U`, `%W` y `%V`. El formato `%y` requiere un cero a la izquierda.

Pie de notas

8.2 calendar — Funciones generales relacionadas con el calendario

Código fuente: [Lib/calendar.py](#)

Este módulo te permite generar calendarios como el programa Unix `cal`, y proporciona funciones útiles adicionales relacionadas con el calendario. Por defecto, estos calendarios tienen el lunes como el primer día de la semana, y el domingo como el último (la convención europea). Use `setfirstweekday()` para establecer el primer día de la semana en domingo (6) o en cualquier otro día de la semana. Los parámetros que especifican fechas se indican como enteros. Para la funcionalidad relacionada, consulta también los módulos `datetime` y `time`.

Las funciones y clases definidas en este módulo utilizan un calendario idealizado, el calendario gregoriano actual extendido indefinidamente en ambas direcciones. Esto coincide con la definición del calendario «Gregoriano pro-léptico» en el libro «Calendrical Calculations» de Dershowitz y Reingold, donde es el calendario base para todos los

cálculos. Los años cero y negativos se interpretan según lo prescrito por la norma ISO 8601. El año 0 es 1 A. C., el año -1 es 2 a. C., y así sucesivamente.

class `calendar.Calendar` (*firstweekday=0*)

Crea un objeto `Calendar`. *firstweekday* es un entero que especifica el primer día de la semana. 0 es lunes (por defecto), 6 es domingo.

Un objeto `Calendar` proporciona varios métodos que se pueden utilizar para preparar los datos del calendario para dar formato. Esta clase no hace ningún formato en sí. Este es el trabajo de las subclases.

Las instancias de `Calendar` tienen los siguientes métodos:

iterweekdays ()

Retorna un iterador para los números del día de la semana que se usará durante una semana. El primer valor del iterador será el mismo que el valor de la propiedad *firstweekday*.

itermonthdates (*year, month*)

Retorna un iterador para el mes *month* (1–12) en el año *year*. Este iterador retornará todos los días (como objetos `datetime.date`) para el mes y todos los días antes del inicio del mes o después del final del mes que se requieren para obtener una semana completa.

itermonthdays (*year, month*)

Retorna un iterador para el mes *month* en el año *year* similar a `itermonthdates()`, pero no restringido por el intervalo `datetime.date`. Los días retornados serán simplemente el día de los números del mes. Para los días fuera del mes especificado, el número de día es 0.

itermonthdays2 (*year, month*)

Retorna un iterador para el mes *month* del año *year* similar a `itermonthdates()`, pero no restringido por el rango `datetime.date`. Los días retornados serán tuplas que consisten en un número de día del mes y un número de día de la semana.

itermonthdays3 (*year, month*)

Retorna un iterador para el mes *month* del año *year* similar a `itermonthdates()`, pero no restringido por el rango `datetime.date`. Los días retornados serán tuplas que consisten en un año, un mes y un día del mes.

Nuevo en la versión 3.7.

itermonthdays4 (*year, month*)

Retorna un iterador para el mes *month* del año *year* similar a `itermonthdates()`, pero no restringido por el rango `datetime.date`. Los días retornados serán tuplas que consisten en un año, un mes, un día del mes y un día de la semana.

Nuevo en la versión 3.7.

monthdatescalendar (*year, month*)

Retorna una lista de las semanas del mes *month* del año *year* como semanas completas. Las semanas son listas de siete objetos `datetime.date`.

monthdays2calendar (*year, month*)

Retorna una lista de las semanas del mes *month* del año *year* como semanas completas. Las semanas son listas de siete tuplas de números de días y números de días de la semana.

monthdayscalendar (*year, month*)

Retorna una lista de las semanas del mes *month* del año *year* como semanas completas. Las semanas son listas de números de siete días.

yeardatescalendar (*year, width=3*)

Retorna los datos del año especificado listos para ser formateados. El valor de retorno es una lista de filas de mes. Cada fila de mes contiene hasta *width* meses (por defecto hasta 3). Cada mes contiene entre 4 y 6 semanas y cada semana contiene 1–7 días. Los días son objetos `datetime.date`.

yeardays2calendar (*year, width=3*)

Retorna los datos del año especificado listos para ser formateados (similar a `yeardatescalendar()`). Las entradas en las listas de la semana son tuplas de números de días y números de días de la semana. Los números de los días fuera de este mes son cero.

yeardayscalendar (*year*, *width=3*)

Retorna los datos del año especificado listos para ser formateados (similar a `yeardatescalendar()`). Las entradas en las listas de la semana son números de día. Los números de día fuera de este mes son cero.

class `calendar.TextCalendar` (*firstweekday=0*)

Esta clase puede ser usada para generar calendarios de texto simple.

Las instancias de `TextCalendar` tienen los siguientes métodos:

formatmonth (*theyear*, *themonth*, *w=0*, *l=0*)

Retorna el calendario de un mes en una cadena de varias líneas. Si se proporciona *w*, especifica el ancho de las columnas de fecha, que están centradas. Si se proporciona *l*, especifica el número de líneas que se utilizarán cada semana. Depende del primer día de la semana como se especifica en el constructor o se establece por el método `setfirstweekday()`.

prmonth (*theyear*, *themonth*, *w=0*, *l=0*)

Imprime el calendario de un mes como lo retorna `formatmonth()`.

formatyear (*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

Retorna un calendario de *m* columnas para todo un año como una cadena de varias líneas. Los parámetros opcionales *w*, *l* y *c* son para el ancho de la columna de la fecha, las líneas por semana y el número de espacios entre las columnas del mes, respectivamente. Depende del primer día de la semana como se especifica en el constructor o se establece por el método `setfirstweekday()`. El primer año para el que se puede generar un calendario depende de la plataforma.

pryear (*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

Imprime el calendario de un año entero como lo retorna `formatyear()`.

class `calendar.HTMLCalendar` (*firstweekday=0*)

Esta clase puede utilizarse para generar calendarios HTML.

Las instancias de `HTMLCalendar` tienen los siguientes métodos:

formatmonth (*theyear*, *themonth*, *withyear=True*)

Retorna el calendario de un mes como una tabla HTML. Si *withyear* es verdadero, el año será incluido en el encabezado, de lo contrario sólo se usará el nombre del mes.

formatyear (*theyear*, *width=3*)

Retorna el calendario de un año como una tabla HTML. *width* (por defecto a 3) especifica el número de meses por fila.

formatyearpage (*theyear*, *width=3*, *css='calendar.css'*, *encoding=None*)

Retorna el calendario de un año como una página HTML completa. *width* (por defecto a 3) especifica el número de meses por fila. *css* es el nombre de la hoja de estilo en cascada que se debe usar. *None* puede ser pasada si no se debe usar una hoja de estilo. *encoding* especifica la codificación a ser usada para la salida (por defecto a la codificación por defecto del sistema).

`HTMLCalendar` tiene los siguientes atributos que puedes sobrescribir para personalizar las clases CSS utilizadas por el calendario:

cssclasses

Una lista de clases CSS utilizadas para cada día de la semana. La lista de clases predeterminada es:

```
cssclasses = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"]
```

se pueden añadir más estilos para cada día:

```
cssclasses = ["mon text-bold", "tue", "wed", "thu", "fri", "sat", "sun red  
↪"]
```

Ten en cuenta que la longitud de esta lista debe ser de siete elementos.

cssclass_noday

La clase CSS para un día de la semana que ocurre en el mes anterior o siguiente.

Nuevo en la versión 3.7.

cssclasses_weekday_head

Una lista de clases CSS utilizadas para los nombres de los días de la semana en la fila del encabezado. El valor por defecto es el mismo que `cssclasses`.

Nuevo en la versión 3.7.

cssclass_month_head

La clase de CSS del mes (usada por `formatmonthname()`). El valor por defecto es "month".

Nuevo en la versión 3.7.

cssclass_month

La clase de CSS para la tabla de todo el mes (usada por `formatmonth()`). El valor por defecto es "month".

Nuevo en la versión 3.7.

cssclass_year

La clase de CSS para la tabla de tablas de todo el año (usada por `formatyear()`). El valor por defecto es "year".

Nuevo en la versión 3.7.

cssclass_year_head

La clase de CSS para el encabezado de la tabla para todo el año (usado por `formatyear()`). El valor por defecto es "year".

Nuevo en la versión 3.7.

Nótese que aunque la denominación de los atributos de clase descritos anteriormente es singular (por ejemplo, `cssclass_month` `cssclass_noday`), uno puede reemplazar la clase CSS única con una lista de clases CSS separadas por espacios, por ejemplo:

```
"text-bold text-red"
```

Aquí hay un ejemplo de cómo `HTMLCalendar` puede ser personalizado:

```
class CustomHTMLCal(calendar.HTMLCalendar):
    cssclasses = [style + " text-nowrap" for style in
                  calendar.HTMLCalendar.cssclasses]
    cssclass_month_head = "text-center month-head"
    cssclass_month = "text-center month"
    cssclass_year = "text-italic lead"
```

class `calendar.LocaleTextCalendar` (*firstweekday=0, locale=None*)

Esta subclase de `TextCalendar` se le puede pasar un nombre de configuración regional en el constructor y retornará los nombres de los meses y días de la semana en la configuración regional especificada. Si esta configuración regional incluye una codificación, todas las cadenas que contengan los nombres de los meses y días de la semana serán retornadas como Unicode.

class `calendar.LocaleHTMLCalendar` (*firstweekday=0, locale=None*)

Esta subclase de `TextCalendar` se le puede pasar un nombre de configuración regional en el constructor y retornará los nombres de los meses y días de la semana en la configuración regional especificada. Si esta configuración regional incluye una codificación, todas las cadenas que contengan los nombres de los meses y días de la semana serán retornadas como Unicode.

Nota: Los métodos `formatweekday()` y `formatmonthname()` de estas dos clases cambian temporalmente la configuración regional actual al *locale* dado. Debido a que la configuración regional actual es un ajuste de todo el proceso, no son seguros para los hilos.

Para los calendarios de texto simples este módulo proporciona las siguientes funciones.

`calendar.setfirstweekday(weekday)`

Establece el día de la semana (el 0 es el lunes, el 6 es el domingo) para empezar cada semana. Los valores `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, y `SUNDAY` se proporcionan por conveniencia. Por ejemplo, para fijar el primer día de la semana en domingo:

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

`calendar.firstweekday()`

Retorna la configuración actual para el día de la semana para empezar cada semana.

`calendar.isleap(year)`

Retorna *True* si *year* es un año bisiesto, si no *False*.

`calendar.leapdays(y1, y2)`

Retorna el número de años bisiestos en el rango de *y1* a *y2* (exclusivo), donde *y1* y *y2* son años.

Esta función opera para rangos que abarcan un cambio de siglo.

`calendar.weekday(year, month, day)`

Retorna el día de la semana (0 es lunes) para *year* (1970`--...`), **month** (`1-12`), *day* (1-31).

`calendar.weekheader(n)`

Retorna un encabezado con nombres abreviados de los días de la semana. *n* especifica el ancho en caracteres de un día de la semana.

`calendar.monthrange(year, month)`

Retorna el día de la semana del primer día del mes y el número de días del mes, para el *year* y *month* especificados.

`calendar.monthcalendar(year, month)`

Returns a matrix representing a month's calendar. Each row represents a week; days outside of the month are represented by zeros. Each week begins with Monday unless set by `setfirstweekday()`.

`calendar.prmonth(theyear, themonth, w=0, l=0)`

Imprime el calendario de un mes según lo retorna `month()`.

`calendar.month(theyear, themonth, w=0, l=0)`

Retorna el calendario de un mes en una cadena de varias líneas usando el `formatmonth()` de la clase `TextCalendar`.

`calendar.prcal(year, w=0, l=0, c=6, m=3)`

Imprime el calendario de todo un año como lo retorna `calendar()`.

`calendar.calendar(year, w=2, l=1, c=6, m=3)`

Retorna un calendario de 3 columnas para un año entero como una cadena de varias líneas usando el `formatyear()` de la clase `TextCalendar`.

`calendar.timegm(tuple)`

Una función no relacionada pero útil que toma una tupla de tiempo como la retornada por la función `gmtime()` en el módulo `time`, y retorna el valor correspondiente a la marca de tiempo (*timestamp*) Unix, asumiendo una época de 1970, y la codificación POSIX. De hecho, `time.gmtime()` y `timegm()` son el inverso de cada uno de ellos.

El módulo `calendar` exporta los siguientes atributos de datos:

`calendar.day_name`

Un arreglo que representa los días de la semana en la configuración regional actual.

`calendar.day_abbr`

Un vector que representa los días abreviados de la semana en la configuración regional actual.

`calendar.month_name`

Un vector que representa los meses del año en la configuración regional actual. Esto sigue la convención normal de que enero es el mes número 1, por lo que tiene una longitud de 13 y `month_name[0]` es la cadena vacía.

`calendar.month_abbrev`

Una matriz que representa los meses abreviados del año en la configuración regional actual. Esto sigue la convención normal de que enero es el mes número 1, por lo que tiene una longitud de 13 y `month_abbrev[0]` es la cadena vacía.

Ver también:

Módulo `datetime` Interfaz orientada a objetos para fechas y horas con una funcionalidad similar a la del módulo `time`.

Módulo `time` Funciones de bajo nivel relacionadas con el tiempo.

8.3 collections — Tipos de datos contenedor

Source code: [Lib/collections/__init__.py](#)

Este módulo implementa tipos de datos de contenedores especializados que proporcionan alternativas a los contenedores integrados de uso general de Python, `dict`, `list`, `set`, and `tuple`.

<code>namedtuple()</code>	función <i>factory</i> para crear subclases de <i>tuplas</i> con campos con nombre
<code>deque</code>	contenedor similar a una lista con <i>appends</i> y <i>pops</i> rápidos en ambos extremos
<code>ChainMap</code>	clase similar a <i>dict</i> para crear una vista única de múltiples <i>mapeados</i>
<code>Counter</code>	subclase de <i>dict</i> para contar objetos <i>hashables</i>
<code>OrderedDict</code>	subclase de <i>dict</i> que recuerda las entradas de la orden que se agregaron
<code>defaultdict</code>	subclase de <i>dict</i> que llama a una función de <i>factory</i> para suministrar valores faltantes
<code>UserDict</code>	envoltura alrededor de los objetos de diccionario para facilitar subclasificaciones <i>dict</i>
<code>UserList</code>	envoltura alrededor de los objetos de lista para facilitar la subclasificación de un <i>list</i>
<code>UserString</code>	envoltura alrededor de objetos de cadena para facilitar la subclasificación de <i>string</i>

Deprecated since version 3.3, will be removed in version 3.10: Trasladado *Colecciones Clases Base Abstractas* al módulo `collections.abc`. Para compatibilidad hacia atrás, continúan siendo visibles en este módulo a través de Python 3.9.

8.3.1 Objetos ChainMap

Nuevo en la versión 3.3.

Una clase `ChainMap` se proporciona para vincular rápidamente una serie de *mappings* de modo que puedan tratarse como una sola unidad. Suele ser mucho más rápido que crear un diccionario nuevo y ejecutar varias llamadas a `update()`.

La clase se puede utilizar para simular ámbitos anidados y es útil para crear plantillas.

class `collections.ChainMap(*maps)`

Un `ChainMap` agrupa varios diccionarios u otros *mappings* para crear una vista única y actualizable. Si no se especifican *maps*, se proporciona un solo diccionario vacío para que una nueva cadena siempre tenga al menos un *mapeo*.

Las asignaciones subyacentes se almacenan en una lista. Esa lista es pública y se puede acceder a ella o actualizarla usando el atributo `maps`. No hay otro estado.

Las búsquedas buscan los mapeos subyacentes sucesivamente hasta que se encuentra una clave. Por el contrario, las escrituras, actualizaciones y eliminaciones solo operan en el primer *mapeo*.

Un `ChainMap` incorpora los mapeos subyacentes por referencia. Entonces, si una de los mapeos subyacentes se actualiza, esos cambios se reflejarán en `ChainMap`.

Se admiten todos los métodos habituales de un diccionario. Además, hay un atributo `maps`, un método para crear nuevos sub contextos y una propiedad para acceder a todos menos al primer mapeo:

maps

Una lista de mapeos actualizable por el usuario. La lista está ordenada desde la primera búsqueda hasta la última búsqueda. Es el único estado almacenado y se puede modificar para cambiar los mapeos que se buscan. La lista siempre debe contener al menos un mapeo.

new_child (*m=None*)

Retorna un nuevo *ChainMap* conteniendo un nuevo mapa seguido de todos los mapas de la instancia actual. Si se especifica *m*, se convierte en el nuevo mapa al principio de la lista de asignaciones; si no se especifica, se usa un dict vacío, de modo que una llamada a `d.new_child()` es equivalente a: `ChainMap({}, *d.maps)`. Este método se utiliza para crear sub contextos que se pueden actualizar sin alterar los valores en ninguna de los mapeos padre.

Distinto en la versión 3.4: Se agregó el parámetro opcional *m*.

parents

Propiedad que retorna un nuevo *ChainMap* conteniendo todos los mapas de la instancia actual excepto el primero. Esto es útil para omitir el primer mapa en la búsqueda. Los casos de uso son similares a los de `nonlocal` la palabra clave usada en *alcances anidados*. Los casos de uso también son paralelos a los de la función incorporada `super()`. Una referencia a `d.parents` es equivalente a: `ChainMap(*d.maps[1:])`.

Tenga en cuenta que el orden de iteración de a *ChainMap()* se determina escaneando los mapeos del último al primero:

```
>>> baseline = {'music': 'bach', 'art': 'rembrandt'}
>>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
>>> list(ChainMap(adjustments, baseline))
['music', 'art', 'opera']
```

Esto da el mismo orden que una serie de llamadas a `dict.update()` comenzando con el último mapeo:

```
>>> combined = baseline.copy()
>>> combined.update(adjustments)
>>> list(combined)
['music', 'art', 'opera']
```

Ver también:

- La clase *MultiContext* en el paquete de Enthought llamado *CodeTools* tiene opciones para admitir la escritura en cualquier mapeo de la cadena.
- Django's *Context class* for templating is a read-only chain of mappings. It also features pushing and popping of contexts similar to the `new_child()` method and the `parents` property.
- La *receta de Contextos Anidados* tiene opciones para controlar si las escrituras y otras mutaciones se aplican solo al primer mapeo o a cualquier mapeo en la cadena.
- Una *versión de solo lectura muy simplificada de Chainmap*.

Ejemplos y recetas ChainMap

Esta sección muestra varios enfoques para trabajar con mapas encadenados.

Ejemplo de simulación de la cadena de búsqueda interna de Python:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

Ejemplo de dejar que los argumentos de la línea de comandos especificados por el usuario tengan prioridad sobre las variables de entorno que, a su vez, tienen prioridad sobre los valores predeterminados:


```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k: v for k, v in vars(namespace).items() if v is not None}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

Patrones de ejemplo para usar la clase `ChainMap` para simular contextos anidados:

```
c = ChainMap()           # Create root context
d = c.new_child()        # Create nested child context
e = c.new_child()        # Child of c, independent from d
e.maps[0]                # Current context dictionary -- like Python's locals()
e.maps[-1]               # Root context -- like Python's globals()
e.parents                # Enclosing context chain -- like Python's nonlocals

d['x'] = 1                # Set value in current context
d['x']                   # Get first key in the chain of contexts
del d['x']                # Delete from current context
list(d)                  # All nested values
k in d                   # Check all nested values
len(d)                   # Number of nested values
d.items()                # All nested items
dict(d)                  # Flatten into a regular dictionary
```

La clase `ChainMap` solo realiza actualizaciones (escrituras y eliminaciones) en el primer mapeo de la cadena, mientras que las búsquedas buscarán en la cadena completa. Sin embargo, si se desean escrituras y eliminaciones profundas, es fácil crear una subclase que actualice las llaves que se encuentran más profundas en la cadena:

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'      # update an existing key two levels down
>>> d['snake'] = 'red'        # new keys get added to the topmost dict
>>> del d['elephant']         # remove an existing key one level down
>>> d                         # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

8.3.2 Objetos Counter

Se proporciona una herramienta de contador para respaldar recuentos rápidos y convenientes. Por ejemplo:

```
>>> # 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*])

Una clase *Counter* es una subclase *dict* para contar objetos hashables. Es una colección donde los elementos se almacenan como llaves de diccionario y sus conteos se almacenan como valores de diccionario. Se permite que los conteos sean cualquier valor entero, incluidos los conteos de cero o negativos. La clase *Counter* es similar a los *bags* o multiconjuntos en otros idiomas.

Los elementos se cuentan desde un *iterable* o se inicializan desde otro *mapeo* (o contador):

```
>>> 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
```

Los objetos Counter tienen una interfaz de diccionario, excepto que retornan un conteo de cero para los elementos faltantes en lugar de levantar una *KeyError*:

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon'] # count of a missing element is_
↪ zero
0
```

Establecer un conteo en cero no elimina un elemento de un contador. Utilice `del` para eliminarlo por completo:

```
>>> c['sausage'] = 0 # counter entry with a zero count
>>> del c['sausage'] # del actually removes the entry
```

Nuevo en la versión 3.1.

Distinto en la versión 3.7: Como subclase de *dict*, *Counter* heredó la capacidad de recordar el orden de inserción. Las operaciones matemáticas en objetos *Counter* también preserva el orden. Los resultados se ordenan cuando se encuentra un elemento por primera vez en el operando izquierdo y luego según el orden encontrado en el operando derecho.

Los objetos Counter admiten tres métodos más allá de los disponibles para todos los diccionarios:

elements()

Retorna un iterador sobre los elementos que se repiten tantas veces como su conteo. Los elementos se retornan en el orden en que se encontraron por primera vez. Si el conteo de un elemento es menor que uno, *elements()* lo ignorará.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common (*[n]*)

Retorna una lista de los *n* elementos mas comunes y sus conteos, del mas común al menos común. Si se omite *n* o None, *most_common()* retorna *todos* los elementos del contador. Los elementos con conteos iguales se ordenan en el orden en que se encontraron por primera vez:

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

subtract (*[iterable-or-mapping]*)

Los elementos se restan de un *iterable* o de otro *mapeo* (o contador). Como *dict.update()* pero resta los conteos en lugar de reemplazarlos. Tanto las entradas como las salidas pueden ser cero o negativas.

```
>>> 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})
```

Nuevo en la versión 3.2.

Los métodos de diccionario habituales están disponibles para objetos *Counter* excepto dos que funcionan de manera diferente para los contadores.

fromkeys (*iterable*)

Este método de clase no está implementado para objetos *Counter*.

update (*[iterable-or-mapping]*)

Los elementos se cuentan desde un *iterable* o agregados desde otro *mapeo* (o contador). Como *dict.update()* pero agrega conteos en lugar de reemplazarlos. Además, se espera que el *iterable* sea una secuencia de elementos, no una secuencia de parejas (*llave, valor*).

Patrones comunes para trabajar con objetos *Counter*:

```
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                       # remove zero and negative counts
```

Se proporcionan varias operaciones matemáticas para combinar objetos *Counter* para producir multiconjuntos (contadores que tienen conteos mayores que cero). La suma y la resta combinan contadores sumando o restando los conteos de los elementos correspondientes. La Intersección y unión retornan el mínimo y el máximo de conteos correspondientes. Cada operación puede aceptar entradas con conteos con signo, pero la salida excluirá los resultados con conteos de cero o menos.

```
>>> 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})
```

La suma y resta unaria son atajos para agregar un contador vacío o restar de un contador vacío.

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

Nuevo en la versión 3.3: Se agregó soporte para operaciones unarias de adición, resta y multiconjunto en su lugar (*in-place*).

Nota: Los `Counters` se diseñaron principalmente para trabajar con números enteros positivos para representar conteos continuos; sin embargo, se tuvo cuidado de no excluir innecesariamente los casos de uso que necesitan otros tipos o valores negativos. Para ayudar con esos casos de uso, esta sección documenta el rango mínimo y las restricciones de tipo.

- La clase `Counter` en sí misma es una subclase de diccionario sin restricciones en sus llaves y valores. Los valores están pensados para ser números que representan conteos, pero *podría* almacenar cualquier cosa en el campo de valor.
- El método `most_common()` solo requiere que los valores se puedan ordenar.
- Para operaciones en su lugar (*in-place*) como `c[key] += 1`, el tipo de valor solo necesita admitir la suma y la resta. Por lo tanto, las fracciones, flotantes y decimales funcionarían y se admiten valores negativos. Lo mismo ocurre con `update()` y `subtract()` que permiten valores negativos y cero para las entradas y salidas.
- Los métodos de multiconjuntos están diseñados solo para casos de uso con valores positivos. Las entradas pueden ser negativas o cero, pero solo se crean salidas con valores positivos. No hay restricciones de tipo, pero el tipo de valor debe admitir la suma, la resta y la comparación.
- El método `elements()` requiere conteos enteros. Ignora los conteos de cero y negativos.

Ver también:

- Clase `Bag` en `Smalltalk`.
- Entrada de Wikipedia para [Multiconjuntos](#).
- Tutorial de [multiconjuntos de C++](#) con ejemplos.
- Para operaciones matemáticas en multiconjuntos y sus casos de uso, consulte *Knuth, Donald. The Art of Computer Programming Volume II, Sección 4.6.3, Ejercicio 19*.
- Para enumerar todos los distintos multiconjuntos de un tamaño dado sobre un conjunto dado de elementos, consulte `itertools.combinations_with_replacement()`:

```
map(Counter, combinations_with_replacement('ABC', 2)) # --> AA AB AC BB BC CC
```

8.3.3 Objetos deque

class `collections.deque` (`[iterable[, maxlen]]`)

Retorna un nuevo objeto deque inicializado de izquierda a derecha (usando `append()`) con datos de *iterable*. Si no se especifica *iterable*, el nuevo deque estará vacío.

Los deques son una generalización de pilas y colas (el nombre se pronuncia “baraja”, *deck* en inglés, y es la abreviatura de “cola de dos extremos”, *double-ended queue* en inglés). Los Deques admiten hilos seguros, `appends` y `pops` eficientes en memoria desde cualquier lado del deque con aproximadamente el mismo rendimiento $O(1)$ en cualquier dirección.

Aunque los objetos `list` admiten operaciones similares, están optimizados para operaciones rápidas de longitud fija e incurrir en costos de movimiento de memoria $O(n)$ para operaciones `pop(0)` y `insert(0, v)` que cambian tanto el tamaño como la posición de la representación de datos subyacente.

Si no se especifica *maxlen* o es `None`, los deque pueden crecer hasta una longitud arbitraria. De lo contrario, el deque está limitado a la longitud máxima especificada. Una vez que un deque de longitud limitada esta lleno, cuando se agregan nuevos elementos, se descarta el número correspondiente de elementos del extremo opuesto. Los deque de longitud limitada proporcionan una funcionalidad similar al filtro `tail` en Unix. También son útiles para rastrear transacciones y otros grupos de datos donde solo la actividad más reciente es de interés.

Los objetos deque admiten los siguientes métodos:

append (*x*)

Agregue *x* al lado derecho del deque.

appendleft (*x*)

Agregue *x* al lado izquierdo del deque.

clear ()

Retire todos los elementos del deque dejándolo con longitud 0.

copy ()

Crea una copia superficial del deque.

Nuevo en la versión 3.5.

count (*x*)

Cuenta el número de elementos deque igual a *x*.

Nuevo en la versión 3.2.

extend (*iterable*)

Extienda el lado derecho del deque agregando elementos del argumento iterable.

extendleft (*iterable*)

Extienda el lado izquierdo del deque agregando elementos de *iterable*. Tenga en cuenta que la serie de `appends` a la izquierda da como resultado la inversión del orden de los elementos en el argumento iterable.

index (*x*[, *start*[, *stop*]])

Retorna la posición de *x* en el deque (en o después del índice *start* y antes del índice *stop*). Retorna la primera coincidencia o lanza `ValueError` si no se encuentra.

Nuevo en la versión 3.5.

insert (*i*, *x*)

Ingresa *x* en el deque en la posición *i*.

Si la inserción causara que un deque limitado crezca más allá de *maxlen*, se lanza un `IndexError`.

Nuevo en la versión 3.5.

pop ()

Elimina y retorna un elemento del lado derecho del deque. Si no hay elementos presentes, lanza un `IndexError`.

popleft ()

Elimina y retorna un elemento del lado izquierdo del deque. Si no hay elementos presentes, lanza un `IndexError`.

remove (*value*)

Elimina la primera aparición de *value*. Si no se encuentra, lanza un `ValueError`.

reverse ()

Invierte los elementos del deque en su lugar (*in-place*) y luego retorna `None`.

Nuevo en la versión 3.2.

rotate (*n=1*)

Gira el deque *n* pasos a la derecha. Si *n* es negativo, lo gira hacia la izquierda.

Cuando el deque no está vacío, girar un paso hacia la derecha equivale a `d.appendleft(d.pop())`, y girar un paso hacia la izquierda equivale a `d.append(d.popleft())`.

Los objetos deque también proporcionan un atributo de solo lectura:

maxlen

Tamaño máximo de un deque o None si no está limitado.

Nuevo en la versión 3.1.

Además de lo anterior, los deques admiten iteración, pickling, `len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, prueba de pertenencia con el operador `in`, y referencias de subíndices como `d[0]` para acceder al primer elemento. El acceso indexado es $O(1)$ en ambos extremos, pero se ralentiza hasta $O(n)$ en el medio. Para un acceso aleatorio rápido, use listas en su lugar.

A partir de la versión 3.5, los deques admiten `__add__()`, `__mul__()`, y `__imul__()`.

Ejemplo:

```
>>> 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'

>>> 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'])
```

Recetas deque

Esta sección muestra varios enfoques para trabajar con deques.

Los deques de longitud limitada proporcionan una funcionalidad similar al filtro `tail` en Unix:

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

Otro enfoque para usar deques es mantener una secuencia de elementos agregados recientemente haciendo `append` a la derecha y `popping` a la izquierda:

```
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 / n
```

Un `scheduler round-robin` se puede implementar con iteradores de entrada almacenados en `deque`. Los valores son producidos del iterador activo en la posición cero. Si ese iterador está agotado, se puede eliminar con `popleft()`; de lo contrario, se puede volver en ciclos al final con el método `rotate()`

```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    iterators = deque(map(iter, iterables))
    while iterators:
        try:
            while True:
                yield next(iterators[0])
                iterators.rotate(-1)
            except StopIteration:
                # Remove an exhausted iterator.
                iterators.popleft()
```

El método `rotate()` proporciona una forma de implementar eliminación y rebanado de `deque`. Por ejemplo, una implementación pura de Python de `del d[n]` se basa en el método `rotate()` para colocar los elementos que se van a extraer:

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

Para implementar el rebanado de un `deque`, use un enfoque similar aplicando `rotate()` para traer un elemento objetivo al lado izquierdo del deque. Elimine las entradas antiguas con `popleft()`, agregue nuevas entradas con `extend()`, y luego invierta la rotación. Con variaciones menores en ese enfoque, es fácil implementar manipulaciones de pila de estilo hacia adelante como `dup`, `drop`, `swap`, `over`, `pick`, `rot`, y `roll`.

8.3.4 Objetos defaultdict

class collections.defaultdict ([*default_factory*[, ...]])

Retorna un nuevo objeto similar a un diccionario. *defaultdict* es una subclase de la clase incorporada *dict*. Anula un método y agrega una variable de instancia de escritura. La funcionalidad restante es la misma que para la clase *dict* y no está documentada aquí.

El primer argumento proporciona el valor inicial para el atributo *default_factory*; por defecto es *None*. Todos los argumentos restantes se tratan de la misma forma que si se pasaran al constructor *dict*, incluidos los argumentos de palabras clave.

Los objetos *defaultdict* admiten el siguiente método además de las operaciones estándar de *dict*:

__missing__ (*key*)

Si el atributo *default_factory* es *None*, lanza una excepción *KeyError* con la *llave* como argumento.

Si *default_factory* no es *None*, se llama sin argumentos para proporcionar un valor predeterminado para la *llave* dada, este valor se inserta en el diccionario para la *llave* y se retorna.

Si llamar a *default_factory* lanza una excepción, esta excepción se propaga sin cambios.

Este método es llamado por el método *__getitem__*() de la clase *dict* cuando no se encuentra la llave solicitada; todo lo que retorna o lanza es retornado o lanzado por *__getitem__*() .

Tenga en cuenta que *__missing__*() no se llama para ninguna operación aparte de *__getitem__*() . Esto significa que *get*() , como los diccionarios normales, retornará *None* por defecto en lugar de usar *default_factory*.

los objetos *defaultdict* admiten la siguiente variable de instancia:

default_factory

Este atributo es utilizado por el método *__missing__*() ; se inicializa desde el primer argumento al constructor, si está presente, o en *None*, si está ausente.

Ejemplos defaultdict

Usando *list* como *default_factory*, es fácil agrupar una secuencia de pares llave-valor en un diccionario de listas:

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Cuando se encuentra cada llave por primera vez, no está ya en el mapping; por lo que una entrada se crea automáticamente usando la función *default_factory* que retorna una *list* vacía. La operación *list.append()* luego adjunta el valor a la nueva lista. Cuando se vuelven a encontrar llaves, la búsqueda procede normalmente (retornando la lista para esa llave) y la operación *list.append()* agrega otro valor a la lista. Esta técnica es más simple y rápida que una técnica equivalente usando *dict.setdefault()*:

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Establecer *default_factory* en *int* hace que *defaultdict* sea útil para contar (como un bag o multiconjunto en otros idiomas):


```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

Cuando se encuentra una letra por primera vez, falta en el mapping, por lo que la función `default_factory` llama a `int()` para proporcionar una cuenta predeterminada de cero. La operación de incremento luego acumula el conteo de cada letra.

La función `int()` que siempre retorna cero es solo un caso especial de funciones constantes. Una forma más rápida y flexible de crear funciones constantes es utilizar una función lambda que pueda proporcionar cualquier valor constante (no solo cero):

```
>>> def constant_factory(value):
...     return lambda: value
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

Establecer `default_factory` en `set` hace que `defaultdict` sea útil para construir un diccionario de conjuntos:

```
>>> 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)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.3.5 `namedtuple()` Funciones *Factory* para Tuplas y Campos con Nombres

Las tuplas con nombre asignan significado a cada posición en una tupla y permiten un código más legible y auto-documentado. Se pueden usar donde se usen tuplas regulares y agregan la capacidad de acceder a los campos por nombre en lugar del índice de posición.

`collections.namedtuple` (*typename*, *field_names*, *, *rename=False*, *defaults=None*, *module=None*)

Retorna una nueva subclase de tupla llamada *typename*. La nueva subclase se utiliza para crear objetos tipo tupla que tienen campos accesibles mediante búsqueda de atributos, además de ser indexables e iterables. Las instancias de la subclase también tienen un docstring útil (con *typename* y *field_names*) y un método útil `__repr__()` que lista el contenido de la tupla en un formato de `nombre=valor`.

Los *nombres de campo* son una secuencia de cadenas como `['x', 'y']`. Alternativamente, *nombres de campo* puede ser una sola cadena con cada nombre de campo separado por espacios en blanco y/o comas, por ejemplo `'x y'` or `'x, y'`.

Se puede usar cualquier identificador de Python válido para un *fieldname*, excepto para los nombres que comienzan con un guión bajo. Los identificadores válidos constan de letras, dígitos y guiones bajos, pero no comienzan con un dígito o guion bajo y no pueden ser *keyword* como `class`, `for`, `return`, `global`, `pass`, o `raise`.

Si *rename* es verdadero, los nombres de campo no válidos se reemplazan automáticamente con nombres posicionales. Por ejemplo, `['abc', 'def', 'ghi', 'abc']` se convierte en `['abc', '_1', 'ghi', '_3']`, eliminando la palabra clave `def` y el nombre de campo duplicado `abc`.

defaults pueden ser `None` o un *iterable* de los valores predeterminados. Dado que los campos con un valor predeterminado deben venir después de cualquier campo sin un valor predeterminado, los *defaults* se aplican a los parámetros situados más a la derecha. Por ejemplo, si los nombres de campo son `['x', 'y', 'z']` y

los valores predeterminados son ``(1, 2)`, entonces `x` será un argumento obligatorio, y tendrá el valor predeterminado de 1, y `z` el valor predeterminado de 2.

Si se define *module*, el atributo `__module__` de la tupla nombrada se establece en ese valor.

Las instancias de tuplas con nombre no tienen diccionarios por instancia, por lo que son livianas y no requieren más memoria que las tuplas normales.

Distinto en la versión 3.1: Se agregó soporte para *rename*.

Distinto en la versión 3.6: Los parámetros *verbose* y *rename* se convirtieron en *argumentos de solo palabra clave*.

Distinto en la versión 3.6: Se agregó el parámetro *module*.

Distinto en la versión 3.7: Se eliminaron el parámetro *verbose* y el atributo `_source`.

Distinto en la versión 3.7: Se agregaron el parámetro *defaults* y el atributo `_field_defaults`.

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> 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)
```

Las tuplas con nombre son especialmente útiles para asignar nombres de campo a las tuplas de resultado retornadas por los módulos *csv* o *sqlite3*:

```
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)
```

Además de los métodos heredados de las tuplas, las tuplas con nombre admiten tres métodos adicionales y dos atributos. Para evitar conflictos con los nombres de campo, los nombres de método y atributo comienzan con un guión bajo.

classmethod `somenamedtuple._make(iterable)`

Método de clase que crea una nueva instancia a partir de una secuencia existente o iterable.

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

`somenamedtuple._asdict()`

Retorna un nuevo *dict* que asigna los nombres de los campos a sus valores correspondientes:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
{'x': 11, 'y': 22}
```

Distinto en la versión 3.1: Retorna un *OrderedDict* en lugar de un *dict* regular.

Distinto en la versión 3.8: Retorna un *dict* normal en lugar de un *OrderedDict*. A partir de Python 3.7, se garantiza el orden de los diccionarios normales. Si se requieren las características adicionales de *OrderedDict*, la corrección sugerida es emitir el resultado al tipo deseado: `OrderedDict(nt._asdict())`.

`somenamedtuple._replace(**kwargs)`

Retorna una nueva instancia de la tupla nombrada reemplazando los campos especificados con nuevos valores:

```
>>> 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`

Tupla de cadenas que lista los nombres de los campos. Útil para la introspección y para crear nuevos tipos de tuplas con nombre a partir de tuplas con nombre existentes.

```
>>> 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)
```

`somenamedtuple._field_defaults`

Diccionario de nombres de campos mapeados a valores predeterminados.

```
>>> Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
>>> Account._field_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=0)
```

Para recuperar un campo cuyo nombre está almacenado en una cadena, use la función `getattr()`:

```
>>> getattr(p, 'x')
11
```

Para convertir un diccionario en una tupla con nombre, use el operador de doble estrella (como se describe en `unpacking-arguments`):

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

Dado que una tupla con nombre es una clase Python normal, es fácil agregar o cambiar la funcionalidad con una subclase. A continuación, se explica cómo agregar un campo calculado y un formato de impresión de ancho fijo:

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
```

(continué en la próxima página)

(proviene de la página anterior)

```

...     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

```

La subclase que se muestra arriba establece `__slots__` a una tupla vacía. Esto ayuda a mantener bajos los requisitos de memoria al evitar la creación de diccionarios de instancia.

La subclasificación no es útil para agregar campos nuevos almacenados. En su lugar, simplemente cree un nuevo tipo de tupla con nombre a partir del atributo `__fields__`:

```
>>> Point3D = namedtuple('Point3D', Point.__fields + ('z',))
```

Los docstrings se pueden personalizar realizando asignaciones directas a los campos `__doc__`:

```

>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'

```

Distinto en la versión 3.5: Los docstrings de propiedad se pueden escribir.

Ver también:

- Consulte [typing.NamedTuple](#) para ver una forma de agregar sugerencias de tipo para tuplas con nombre. También proporciona una notación elegante usando la palabra clave `class`:

```

class Component(NamedTuple):
    part_number: int
    weight: float
    description: Optional[str] = None

```

- Vea [types.SimpleNamespace\(\)](#) para un espacio de nombres mutable basado en un diccionario subyacente en lugar de una tupla.
- El módulo [dataclasses](#) proporciona un decorador y funciones para agregar automáticamente métodos especiales generados a clases definidas por el usuario.

8.3.6 Objetos `OrderedDict`

Los diccionarios ordenados son como los diccionarios normales, pero tienen algunas capacidades adicionales relacionadas con las operaciones de ordenado. Se han vuelto menos importantes ahora que la clase incorporada `dict` ganó la capacidad de recordar el orden de inserción (este nuevo comportamiento quedó garantizado en Python 3.7).

Aún quedan algunas diferencias con `dict`:

- El `dict` normal fue diseñado para ser muy bueno en operaciones de mapeo. El seguimiento del pedido de inserción era secundario.
- La `OrderedDict` fue diseñada para ser buena para reordenar operaciones. La eficiencia del espacio, la velocidad de iteración y el rendimiento de las operaciones de actualización fueron secundarios.
- Algorítmicamente, `OrderedDict` puede manejar operaciones frecuentes de reordenamiento mejor que `dict`. Esto lo hace adecuado para rastrear accesos recientes (por ejemplo, en un [cache LRU](#)).
- La operación de igualdad para `OrderedDict` comprueba el orden coincidente.

- El método `popitem()` de `OrderedDict` tiene una firma diferente. Acepta un argumento opcional para especificar qué elemento es *popped*.
- `OrderedDict` tiene un método `move_to_end()` para reposiciones eficientemente un elemento a un punto final.
- Hasta Python 3.8, `dict` carecía de un método `__reversed__()`.

class `collections.OrderedDict` (`[items]`)

Retorna una instancia de una subclase `dict` que tiene métodos especializados para reorganizar el orden del diccionario.

Nuevo en la versión 3.1.

popitem (`last=True`)

El método `popitem()` para diccionarios ordenados retorna y elimina un par (llave, valor). Los pares se retornan en orden LIFO si el *último* es verdadero o en orden FIFO (first-in, first-out) si es falso.

move_to_end (`key, last=True`)

Mueva una *llave* existente a cualquier extremo de un diccionario ordenado. El elemento se mueve al final de la derecha si el *último* es verdadero (el valor predeterminado) o al principio si el *último* es falso. Lanza `KeyError` si la *llave* no existe:

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d.keys())
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d.keys())
'bacde'
```

Nuevo en la versión 3.2.

Además de los métodos de mapeo habituales, los diccionarios ordenados también admiten la iteración inversa usando `reversed()`.

Las pruebas de igualdad entre los objetos `OrderedDict` son sensibles al orden y se implementan como `list(od1.items())==list(od2.items())`. Las pruebas de igualdad entre los objetos `OrderedDict` y otros objetos `Mapping` son insensibles al orden como los diccionarios normales. Esto permite que los objetos `OrderedDict` sean sustituidos en cualquier lugar donde se utilice un diccionario normal.

Distinto en la versión 3.5: Los elementos, llaves y valores *vistas* de `OrderedDict` ahora admiten la iteración inversa usando `reversed()`.

Distinto en la versión 3.6: Con la aceptación de **PEP 468**, el orden se mantiene para los argumentos de palabras clave pasados al constructor `OrderedDict` y su método `update()`.

Ejemplos y Recetas OrderedDict

Es sencillo crear una variante de diccionario ordenado que recuerde el orden en que las llaves se insertaron por *última* vez. Si una nueva entrada sobrescribe una entrada existente, la posición de inserción original se cambia y se mueve al final:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

U `OrderedDict` también sería útil para implementar variantes de `functools.lru_cache()`:

```
class LRU(OrderedDict):
    'Limit size, evicting the least recently looked-up key when full'

    def __init__(self, maxsize=128, /, *args, **kwargs):
        self.maxsize = maxsize
        super().__init__(*args, **kwargs)

    def __getitem__(self, key):
        value = super().__getitem__(key)
        self.move_to_end(key)
        return value

    def __setitem__(self, key, value):
        if key in self:
            self.move_to_end(key)
        super().__setitem__(key, value)
        if len(self) > self.maxsize:
            oldest = next(iter(self))
            del self[oldest]
```

8.3.7 Objetos UserDict

La clase, *UserDict* actúa como un contenedor alrededor de los objetos del diccionario. La necesidad de esta clase ha sido parcialmente suplantada por la capacidad de crear subclases directamente desde *dict*; sin embargo, es más fácil trabajar con esta clase porque se puede acceder al diccionario subyacente como un atributo.

class `collections.UserDict` (`[initialdata]`)

Clase que simula un diccionario. El contenido de la instancia se guarda en un diccionario normal, al que se puede acceder mediante el atributo *data* de las instancias *UserDict*. Si se proporciona *initialdata*, *data* se inicializa con su contenido; tenga en cuenta que no se mantendrá una referencia a *initialdata*, lo que permite que se utilice para otros fines.

Además de admitir los métodos y operaciones de los mappings, las instancias *UserDict* proporcionan el siguiente atributo:

data

Un diccionario real utilizado para almacenar el contenido de la clase *UserDict*.

8.3.8 Objetos UserList

Esta clase actúa como un contenedor alrededor de los objetos de lista. Es una clase base útil para tus propias clases tipo lista que pueden heredar de ellas y anular métodos existentes o agregar nuevos. De esta forma, se pueden agregar nuevos comportamientos a las listas.

La necesidad de esta clase ha sido parcialmente suplantada por la capacidad de crear subclases directamente desde *list*; sin embargo, es más fácil trabajar con esta clase porque se puede acceder a la lista subyacente como atributo.

class `collections.UserList` (`[list]`)

Clase que simula una lista. El contenido de la instancia se mantiene en una lista normal, a la que se puede acceder mediante el atributo *data* de las instancias *UserList*. El contenido de la instancia se establece inicialmente en una copia de *list*, por defecto a la lista vacía `[]`. *list* puede ser cualquier iterable, por ejemplo, una lista de Python real o un objeto *UserList*.

Además de admitir los métodos y operaciones de secuencias mutables, las instancias *UserList* proporcionan el siguiente atributo:

data

Un objeto real *list* usado para almacenar el contenido de la clase *UserList*.

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.

Si una clase derivada no desea cumplir con este requisito, todos los métodos especiales admitidos por esta clase deberán de anularse; consulte las fuentes para obtener información sobre los métodos que deben proporcionarse en ese caso.

8.3.9 Objetos `UserString`

La clase, `UserString` actúa como un envoltorio alrededor de los objetos de cadena. La necesidad de esta clase ha sido parcialmente suplantada por la capacidad de crear subclases directamente de `str`; sin embargo, es más fácil trabajar con esta clase porque se puede acceder a la cadena subyacente como atributo.

class `collections.UserString` (*seq*)

Clase que simula un objeto de cadena. El contenido de la instancia se mantiene en un objeto de cadena normal, al que se puede acceder a través del atributo `data` de las instancias `UserString`. El contenido de la instancia se establece inicialmente en una copia de *seq*. El argumento *seq* puede ser cualquier objeto que se pueda convertir en una cadena usando la función incorporada `str()`.

Además de admitir los métodos y operaciones de cadenas, las instancias `UserString` proporcionan el siguiente atributo:

data

Un objeto real `str` usado para almacenar el contenido de la clase `UserString`.

Distinto en la versión 3.5: Nuevos métodos `__getnewargs__`, `__rmod__`, `casefold`, `format_map`, `isprintable`, y `maketrans`.

8.4 `collections.abc` — Clases Base Abstractas para Contenedores

Nuevo en la versión 3.3: Anteriormente, este módulo formaba parte del módulo `collections`.

Código fuente: [Lib/_collections_abc.py](#)

Este módulo proporciona *clases base abstractas* que pueden usarse para probar si una clase proporciona una interfaz específica; por ejemplo, si es hashable o si es un mapeo.

8.4.1 Colecciones Clases Base Abstractas

El módulo de colecciones ofrece lo siguiente *ABCs*:

ABC	Hereda de	Métodos Abstractos	Métodos Mixin
<i>Container</i>		<code>__contains__</code>	
<i>Hashable</i>		<code>__hash__</code>	
<i>Iterable</i>		<code>__iter__</code>	
<i>Iterator</i>	<i>Iterable</i>	<code>__next__</code>	<code>__iter__</code>
<i>Reversible</i>	<i>Iterable</i>	<code>__reversed__</code>	
<i>Generator</i>	<i>Iterator</i>	<code>send</code> , <code>throw</code>	<code>close</code> , <code>__iter__</code> , <code>__next__</code>
<i>Sized</i>		<code>__len__</code>	
<i>Callable</i>		<code>__call__</code>	
<i>Collection</i>	<i>Sized</i> , <i>Iterable</i> , <i>Container</i>	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code>	
<i>Sequence</i>	<i>Reversible</i> , <i>Collection</i>	<code>__getitem__</code> , <code>__len__</code>	<code>__contains__</code> , <code>__iter__</code> , <code>__reversed__</code> , <code>index</code> , and <code>count</code>
<i>MutableSequence</i>	<i>Sequence</i>	<code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__len__</code> , <code>insert</code>	Métodos heredados <i>Sequence</i> y <code>append</code> , <code>reverse</code> , <code>extend</code> , <code>pop</code> , <code>remove</code> , and <code>__iadd__</code>
<i>ByteString</i>	<i>Sequence</i>	<code>__getitem__</code> , <code>__len__</code>	Métodos heredados <i>Sequence</i>
<i>Set</i>	<i>Collection</i>	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code>	<code>__le__</code> , <code>__lt__</code> , <code>__eq__</code> , <code>__ne__</code> , <code>__gt__</code> , <code>__ge__</code> , <code>__and__</code> , <code>__or__</code> , <code>__sub__</code> , <code>__xor__</code> , and <code>isdisjoint</code>
<i>MutableSet</i>	<i>Set</i>	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code> , <code>add</code> , <code>discard</code>	Métodos heredados <i>Set</i> y <code>clear</code> , <code>pop</code> , <code>remove</code> , <code>__ior__</code> , <code>__iand__</code> , <code>__ixor__</code> , and <code>__isub__</code>
<i>Mapping</i>	<i>Collection</i>	<code>__getitem__</code> , <code>__iter__</code> , <code>__len__</code>	<code>__contains__</code> , <code>keys</code> , <code>items</code> , <code>values</code> , <code>get</code> , <code>__eq__</code> , and <code>__ne__</code>
<i>MutableMapping</i>	<i>Mapping</i>	<code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__iter__</code> , <code>__len__</code>	Métodos heredados <i>Mapping</i> y <code>pop</code> , <code>popitem</code> , <code>clear</code> , <code>update</code> , and <code>setdefault</code>
<i>MappingView</i>	<i>Sized</i>		<code>__len__</code>
<i>ItemsView</i>	<i>MappingView</i> , <i>Set</i>		<code>__contains__</code> , <code>__iter__</code>
<i>KeysView</i>	<i>MappingView</i> , <i>Set</i>		<code>__contains__</code> , <code>__iter__</code>
<i>ValuesView</i>	<i>MappingView</i> , <i>Collection</i>		<code>__contains__</code> , <code>__iter__</code>
<i>Awaitable</i>		<code>__await__</code>	
<i>Coroutine</i>	<i>Awaitable</i>	<code>send</code> , <code>throw</code>	<code>close</code>
<i>AsyncIterable</i>		<code>__aiter__</code>	
<i>AsyncIterator</i>	<i>AsyncIterable</i>	<code>__anext__</code>	<code>__aiter__</code>
<i>AsyncGenerator</i>	<i>AsyncIterator</i>	<code>asend</code> , <code>athrow</code>	<code>aclose</code> , <code>__aiter__</code> , <code>__anext__</code>

class `collections.abc.Container`
ABC for classes that provide the `__contains__()` method.

class `collections.abc.Hashable`
ABC for classes that provide the `__hash__()` method.

class `collections.abc.Sized`
ABC for classes that provide the `__len__()` method.

class `collections.abc.Callable`
ABC for classes that provide the `__call__()` method.

class collections.abc.Iterable

ABC para clases que proporcionan el método `__iter__()`.

Al marcar `isinstance(obj, Iterable)` se detectan las clases que están registradas como *Iterable* o que tienen un método `__iter__()`, pero no detecta clases que iteran con el método `__getitem__()`. La única forma confiable de determinar si un objeto es *iterable* es llamar a `iter(obj)`.

class collections.abc.Collection

ABC para clases de contenedor iterables de tamaño.

Nuevo en la versión 3.6.

class collections.abc.Iterator

ABC para clases que proporcionan el método `__iter__()` y `__next__()`. Ver también la definición de *iterator*.

class collections.abc.Reversible

ABC para clases iterables que también proporcionan `__reversed__()` method.

Nuevo en la versión 3.6.

class collections.abc.Generator

ABC para clases generadoras que implementan el protocolo definido en **PEP 342** que extiende los iteradores con los métodos `send()`, `throw()` and `close()`. Ver también la definición de *generator*.

Nuevo en la versión 3.5.

class collections.abc.Sequence**class collections.abc.MutableSequence****class collections.abc.ByteString**

ABC para solo lectura y mutable *secuencias*.

Nota de implementación: algunos de los métodos mixin, tales como `__iter__()`, `__reversed__()` and `index()`, hacen llamadas repetidas al subyacente `__getitem__()` method. En consecuencia, si `__getitem__()` se implementa con velocidad de acceso constante, los métodos mixin tendrán un rendimiento lineal; sin embargo, si el método subyacente es lineal (como lo sería con una lista vinculada), los mixins tendrán un rendimiento cuadrático y probablemente deberán ser anulados.

Distinto en la versión 3.5: El método `index()` agregó soporte para los argumentos *stop* y *start*.

class collections.abc.Set**class collections.abc.MutableSet**

ABC para conjuntos de solo lectura y mutables.

class collections.abc.Mapping**class collections.abc.MutableMapping**

ABC para solo lectura y mutable *mapeos*.

class collections.abc.MappingView**class collections.abc.ItemsView****class collections.abc.KeysView****class collections.abc.ValuesView**

ABC para mapeo, elementos, claves y valores *vistas*.

class collections.abc.Awaitable

ABC para objetos *awaitable*, que pueden ser usados en expresiones `await`. Las implementaciones personalizadas deben proporcionar el método `__await__()`.

Coroutine objects and instances of the *Coroutine* ABC are all instances of this ABC.

Nota: En CPython, las corrutinas basadas en generador (generadores decorados con `types.coroutine()` o `asyncio.coroutine()`) son *awaitables*, a pesar de que no tienen un método `__await__()`. El uso de `isinstance(gencoro, Awaitable)` para ellos retornará `False`. Use `inspect.isawaitable()` para detectarlos.

Nuevo en la versión 3.5.

class `collections.abc.Coroutine`

ABC para clases corrutinas compatibles. Estos implementan los siguientes métodos, definidos en `coroutine-objects`: `send()`, `throw()`, and `close()`. Las implementaciones personalizadas también deben implementar `__await__()`. Todas las instancias de `Coroutine` también son instancias de `Awaitable`. Ver también la definición de `coroutine`.

Nota: En CPython, las corrutinas basadas en generador (generadores decorados con `types.coroutine()` o `asyncio.coroutine()`) son *awaitables*, a pesar de que no tienen un método `__await__()`. El uso de `isinstance(gencoro, Coroutine)` para ellos retornará `False`. Use `inspect.isawaitable()` para detectarlos.

Nuevo en la versión 3.5.

class `collections.abc.AsyncIterable`

ABC para las clases que proporcionan el método `__aiter__`. Ver también la definición de *asynchronous iterable*.

Nuevo en la versión 3.5.

class `collections.abc.AsyncIterator`

ABC para clases que proveen métodos `__aiter__` and `__anext__`. Ver también la definición de *asynchronous iterator*.

Nuevo en la versión 3.5.

class `collections.abc.AsyncGenerator`

ABC para clases generadoras asincrónicas que implementan el protocolo definido en **PEP 525** y **PEP 492**.

Nuevo en la versión 3.6.

Estos ABC nos permiten preguntar a clases o instancias si proporcionan una funcionalidad particular, por ejemplo:

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

Varios ABCs también son útiles como mixins que facilitan el desarrollo de clases que admiten APIs de contenedor. Por ejemplo, para escribir una clase que admita toda la API `Set`, solo es necesario proporcionar los tres métodos abstractos subyacentes: `__contains__()`, `__iter__()`, y `__len__()`. El ABC proporciona los métodos restantes, como `__and__()` y `isdisjoint()`:

```
class ListBasedSet(collections.abc.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')
```

(continué en la próxima página)

(proviene de la página anterior)

```
s2 = ListBasedSet('defghi')
overlap = s1 & s2           # The __and__() method is supported automatically
```

Notas sobre el uso de *Set* y *MutableSet* como un mixin:

- (1) Dado que algunas operaciones de conjuntos crean nuevos conjuntos, los métodos mixin predeterminados necesitan una forma de crear nuevas instancias desde un iterable. Se supone que el constructor de la clase tiene una firma en el formato `ClassName (iterable)`. Esa suposición se tiene en cuenta para una clase interna llamada `_from_iterable()` que llama a `cls(iterable)` para producir un nuevo conjunto. Si el mixin *Set* se está utilizando en una clase con una firma de constructor diferente, deberá reemplazar `_from_iterable()` con un método de clase que pueda construir nuevas instancias a partir de un argumento iterable.
- (2) Para reemplazar las comparaciones (presumiblemente para la velocidad, ya que las semánticas son fijas), redefinir `__le__()` y `__ge__()`, luego las otras operaciones seguirán automáticamente su ejemplo.
- (3) El mixin *Set* proporciona un método `__hash__()` para calcular un valor hash para el conjunto; sin embargo, `__hash__()` no está definido porque no todos los conjuntos son encadenados o inmutables. Para agregar capacidad de encadenamiento en conjuntos que usan mixin, herede de ambos *Set()* y *Hashable()*, luego defina `__hash__ = Set.__hash__`.

Ver también:

- *OrderedSet* [receta](#) para un ejemplo basado en *MutableSet*.
- Para obtener más información sobre ABCs, ver el módulo *abc* y [PEP 3119](#).

8.5 heapq — Algoritmo de colas montículos (*heap*)

Código fuente: [Lib/heapq.py](#)

Este módulo proporciona una implementación del algoritmo de montículos, también conocido como algoritmo de cola con prioridad.

Los montículos son árboles binarios para los cuales cada nodo padre tiene un valor menor o igual que cualquiera de sus hijos. Esta implementación utiliza matrices para las cuales `heap[k] <= heap[2*k+1]` y `heap[k] <= heap[2*k+2]` para todo *k*, contando los elementos desde cero. Para poder comparar, los elementos inexistentes se consideran infinitos. La propiedad interesante de un montículo es que su elemento más pequeño es siempre la raíz, `heap[0]`.

El API que se presenta a continuación difiere de los algoritmos de los libros de texto en dos aspectos: (a) Utilizamos la indexación basada en cero. Esto hace que la relación entre el índice de un nodo y los índices de sus hijos sea un poco menos evidente, pero es más adecuado ya que Python utiliza la indexación basada en cero. (b) Nuestro método «pop» retorna el elemento más pequeño, no el más grande (llamado «min heap» o montículo por mínimo en los libros de texto; un «max heap» o montículo por máximos es más común en los textos debido a su idoneidad para la clasificación in situ).

Estos dos permiten ver el montículo como una lista Python normal sin sorpresas: `heap[0]` es el ítem más pequeño, y `heap.sort()` mantiene el montículo invariable!

Para crear un montículo, usa una lista inicializada como `[]`, o puedes transformar una lista poblada en un montículo a través de la función `heapify()`.

Las siguientes funciones están provistas:

`heapq.heappush(heap, item)`

Empujar el valor *item* en el *heap*, manteniendo el montículo invariable.

`heapq.heappop(heap)`

Desapila o *pop* y retorna el elemento más pequeño del *heap*, manteniendo el montículo invariable. Si el montículo está vacío, `IndexError` se lanza. Para acceder al elemento más pequeño sin necesidad de desapilar, usa `heap[0]`.

`heapq.heappushpop(heap, item)`

Apila el elemento o *item* en el montículo, y luego desapila y retorna el elemento más pequeño del montículo. La acción combinada se ejecuta más eficientemente que `heappush()` seguido de una llamada separada a `heappop()`.

`heapq.heapify(x)`

Transformar la lista *x* en un montículo, en el lugar, en tiempo lineal.

`heapq.heapreplace(heap, item)`

Desapila y retorna el elemento más pequeño del *heap*, y también apila el nuevo *item*. El tamaño del montículo no cambia. Si el montículo está vacío, `IndexError` se eleva.

Esta operación de un solo paso es más eficiente que un `heappop()` seguido por `heappush()` y puede ser más apropiada cuando se utiliza un montículo de tamaño fijo. La combinación pop/push siempre retorna un elemento del montículo y lo reemplaza con *item*.

El valor retornado puede ser mayor que el *item* añadido. Si no se desea eso, considere usar `heappushpop()` en su lugar. Su combinación push/pop retorna el menor de los dos valores, dejando el mayor valor en el montículo.

El módulo también ofrece tres funciones de propósito general basadas en los montículos.

`heapq.merge(*iterables, key=None, reverse=False)`

Fusionar varias entradas ordenadas en una sola salida ordenada (por ejemplo, fusionar entradas con marca de tiempo de varios archivos de registro). Retorna un *iterator* sobre los valores ordenados.

Similar a `sorted(itertools.chain(*iterables))` pero retorna un iterable, no hala los datos a la memoria de una sola vez, y asume que cada uno de los flujos de entrada ya están ordenado (de menor a mayor).

Tiene dos argumentos opcionales que deben ser especificados como argumentos de palabras clave.

key especifica una *key function* de un argumento que se utiliza para extraer una clave de comparación de cada elemento de entrada. El valor por defecto es `None` (compara los elementos directamente).

reverse es un valor booleano. Si se establece en `True`, entonces los elementos de entrada se fusionan como si cada comparación se invirtiera. Para lograr un comportamiento similar a `sorted(itertools.chain(*iterables), reverse=True)`, todos los iterables deben ser ordenados de mayor a menor.

Distinto en la versión 3.5: Añadió los parámetros opcionales de *key* y *reverse*.

`heapq.nlargest(n, iterable, key=None)`

Retorna una lista con los *n* elementos más grandes del conjunto de datos definidos por *iterable*. *key*, si se proporciona, especifica una función de un argumento que se utiliza para extraer una clave de comparación de cada elemento en *iterable* (por ejemplo, `key=str.lower`). Equivalente a: `sorted(iterable, key=clave, reverse=True)[:n]`.

`heapq.nsmallest(n, iterable, key=None)`

Retorna una lista con los *n* elementos más pequeños del conjunto de datos definidos por *iterable*. *key*, si se proporciona, especifica una función de un argumento que se utiliza para extraer una clave de comparación de cada elemento en *iterable* (por ejemplo, `key=str.lower`). Equivalente a: `sorted(iterable, key=clave)[:n]`.

Las dos últimas funciones funcionan mejor para valores más pequeños de *n*. Para valores más grandes, es más eficiente usar la función `sorted()`. Además, cuando `n==1`, es más eficiente usar las funciones incorporadas `min()` y `max()`. Si se requiere el uso repetido de estas funciones, considere convertir lo iterable en un verdadero montículo.

8.5.1 Ejemplos Básicos

Un `heapsort` puede ser implementado empujando todos los valores en un montículo y luego desapilando los valores más pequeños uno a la vez:

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Esto es similar a `sorted(iterable)`, pero a diferencia de `sorted()`, esta implementación no es estable.

Los elementos del montículo pueden ser tuplas. Esto es útil para asignar valores de comparación (como las prioridades de las tareas) junto con el registro principal que se está rastreando:

```
>>> 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')
```

8.5.2 Notas de Aplicación de la Cola de Prioridades

Una *cola de prioridad* es de uso común para un montículo, y presenta varios desafíos de implementación:

- Estabilidad de la clasificación: ¿cómo se consigue que dos tareas con iguales prioridades sean retornadas en el orden en que fueron añadidas originalmente?
- Interrupciones de comparación en tupla para pares (prioridad, tarea) si las prioridades son iguales y las tareas no tienen un orden de comparación por defecto.
- ¿Si la prioridad de una tarea cambia, cómo la mueves a una nueva posición en el montículo?
- ¿O si una tarea pendiente necesita ser borrada, cómo la encuentras y la eliminas de la cola?

Una solución a los dos primeros desafíos es almacenar las entradas como una lista de 3 elementos que incluya la prioridad, un recuento de entradas y la tarea. El recuento de entradas sirve como un desempate para que dos tareas con la misma prioridad sean retornadas en el orden en que fueron añadidas. Y como no hay dos recuentos de entradas iguales, la comparación tupla nunca intentará comparar directamente dos tareas.

Otra solución al problema de las tareas no comparables es crear una clase envolvente que ignore el elemento de la tarea y sólo compare el campo de prioridad:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any = field(compare=False)
```

Los desafíos restantes giran en torno a encontrar una tarea pendiente y hacer cambios en su prioridad o eliminarla por completo. Encontrar una tarea se puede hacer con un diccionario que apunta a una entrada en la cola.

Eliminar la entrada o cambiar su prioridad es más difícil porque rompería las invariantes de la estructura del montículo. Por lo tanto, una posible solución es marcar la entrada como eliminada y añadir una nueva entrada con la prioridad revisada:

```

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):
    'Mark an existing task as REMOVED. Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

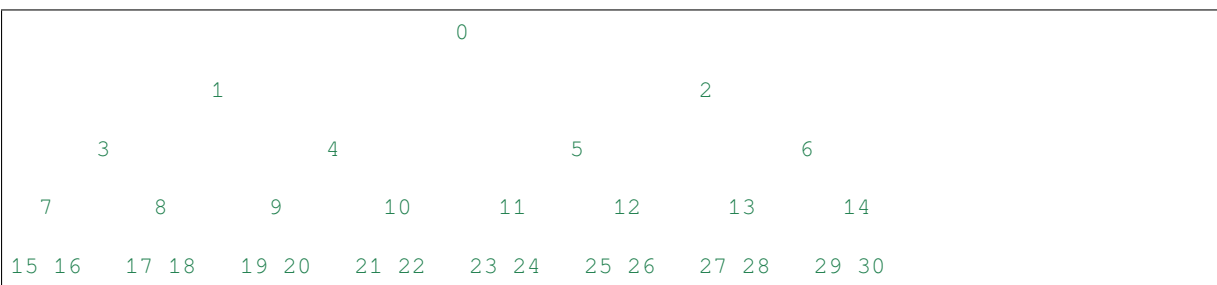
def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')

```

8.5.3 Teoría

Los montículos son conjuntos para los cuales $a[k] \leq a[2k+1]$ y $a[k] \leq a[2k+2]$ para todos los k , contando los elementos desde 0. Para comparar, los elementos no existentes se consideran infinitos. La interesante propiedad de un montículo es que $a[0]$ es siempre su elemento más pequeño.

La extraña invariante de arriba intenta ser una representación eficiente de la memoria para un torneo. Los números de abajo son k , no $a[k]$:



En el árbol de arriba, cada celda k está coronada por $2k+1$ y $2k+2$. En un torneo binario habitual que vemos en los deportes, cada celda es el ganador sobre las dos celdas que supera, y podemos rastrear al ganador hasta el árbol para ver todos los oponentes que tuvo. Sin embargo, en muchas aplicaciones informáticas de tales torneos, no necesitamos rastrear la historia de un ganador. Para ser más eficientes en la memoria, cuando un ganador es ascendido, tratamos de reemplazarlo por algo más en un nivel inferior, y la regla se convierte en que una celda y las dos celdas que supera contienen tres elementos diferentes, pero la celda superior «gana» sobre las dos celdas superiores.

Si esta invariante del montículo está protegida en todo momento, el índice 0 es claramente el ganador general. La forma algorítmica más simple de eliminarlo y encontrar el «próximo» ganador es mover algún perdedor (digamos la celda 30 en el diagrama de arriba) a la posición 0, y luego filtrar este nuevo 0 por el árbol, intercambiando valores, hasta que la invariante se reestablezca. Esto es claramente logarítmico en el número total de elementos del árbol. Al iterar sobre todos los elementos, se obtiene una clasificación $O(n \log n)$.

Una buena característica de este tipo es que puedes insertar nuevos elementos de manera eficiente mientras se realiza

la clasificación, siempre y cuando los elementos insertados no sean «mejores» que el último 0'th elemento que has extraído. Esto es especialmente útil en contextos de simulación, donde el árbol contiene todos los eventos entrantes, y la condición de «ganar» significa el menor tiempo programado. Cuando un evento programa otros eventos para su ejecución, se programan en el futuro, para que puedan ir fácilmente al montículo. Por lo tanto, un montículo es una buena estructura para implementar planificadores o *schedulers* (esto es lo que usé para mi secuenciador MIDI :-).

Se han estudiado extensamente varias estructuras para implementar los planificadores, y los montículos son buenos para ello, ya que son razonablemente rápidos, la velocidad es casi constante, y el peor de los casos no es muy diferente del caso promedio. Sin embargo, hay otras representaciones que son más eficientes en general, aunque los peores casos podrían ser terribles.

Los montículos también son muy útiles en las grandes ordenaciones de elementos en discos de memoria. Lo más probable es que todos sepan que un tipo grande implica la producción de «ejecuciones» (que son secuencias preclasificadas, cuyo tamaño suele estar relacionado con la cantidad de memoria de la CPU), seguidas de una fusión de pases para estas ejecuciones, cuya fusión suele estar muy inteligentemente organizada¹. Es muy importante que la clasificación inicial produzca las ejecuciones posibles más largas. Los torneos son una buena manera de lograrlo. Si, utilizando toda la memoria disponible para celebrar un torneo, sustituyes y filtras los elementos que encajan en la carrera actual, producirás carreras que tienen el doble del tamaño de la memoria para la entrada aleatoria, y mucho mejor para la entrada ordenada de forma difusa.

Además, si se da salida al 0'th item en el disco y se obtiene una entrada que no puede caber en el torneo actual (porque el valor «gana» sobre el último valor de salida), no puede caber en el montículo, por lo que el tamaño del montículo disminuye. La memoria liberada podría ser ingeniosamente reutilizada inmediatamente para construir progresivamente un segundo montículo, que crece exactamente al mismo ritmo que el primer montículo se está fundiendo. Cuando el primer montículo se desvanece completamente, se cambia de montículo y se inicia una nueva carrera. ¡Ingenioso y muy efectivo!

En una palabra, los montículos son estructuras de memoria útiles a conocer. Las uso en algunas aplicaciones, y creo que es bueno tener un módulo “heap” alrededor. :-)

Notas al pie de página

8.6 bisect — Algoritmo de bisección de arreglos

Código fuente: [Lib/bisect.py](#)

Este módulo brinda soporte para mantener una lista ordenada sin tener que reordenar la lista tras cada nueva inserción. Para listas largas de elementos que tienen operaciones de comparación costosas, será una mejora respecto a la estrategia más habitual. El módulo se llama *bisect* porque usa un algoritmo de bisección básico para lograr su objetivo. El código fuente puede ser útil como ejemplo del algoritmo en funcionamiento (¡las precondiciones ya están bien de antemano!).

Las siguientes funciones están disponibles:

`bisect.bisect_left(a, x, lo=0, hi=len(a))`

Ubicar el punto de inserción para *x* en *a* para mantener el ordenamiento. Los parámetros *lo* (inferior) y *hi* (superior) pueden utilizarse para especificar un subconjunto (*subset*) de la lista que debería considerarse. Por defecto, se utiliza la lista completa. Si *x* ya está presente en *a*, el punto de inserción será antes (a la izquierda de) cualquier elemento existente. El valor de retorno es adecuado para que se utilice como primer parámetro para `list.insert()`, suponiendo que *a* ya está ordenada.

El punto de inserción retornado *i* particiona al arreglo *a* en dos mitades, tal que `all(val < x for val in a[lo:i])` para el lado izquierdo y `all(val >= x for val in a[i:hi])` para el derecho.

¹ Los algoritmos de balanceo de discos que están vigentes hoy en día, son más molestos que inteligentes, y esto es una consecuencia de las capacidades de búsqueda de los discos. En los dispositivos que no pueden buscar, como las grandes unidades de cinta, la historia era muy diferente, y había que ser muy inteligente para asegurarse (con mucha antelación) de que cada movimiento de la cinta fuera el más efectivo (es decir, que participara mejor en el «progreso» de la fusión). Algunas cintas eran incluso capaces de leer al revés, y esto también se utilizó para evitar el tiempo rebobinado. Créanme, ¡la ordenación de elementos en cinta realmente buenos fueron espectaculares de ver! ¡Desde todos los tiempos, la ordenación de elementos siempre ha sido un Gran Arte! :-)


```
bisect.bisect_right(a, x, lo=0, hi=len(a))
```

```
bisect.bisect(a, x, lo=0, hi=len(a))
```

Similar a `bisect_left()`, pero retorna un punto de inserción que viene después (a la derecha de) cualquier entrada de `x` en `a`.

El punto de inserción retornado `i` particiona al arreglo `a` en dos mitades, tal que `all(val <= x for val in a[lo:i])` para el lado izquierdo y `all(val > x for val in a[i:hi])` para el derecho.

```
bisect.insort_left(a, x, lo=0, hi=len(a))
```

Inserta `x` en `a` de forma ordenada. Esto equivale a `a.insert(bisect.bisect_left(a, x, lo, hi), x)`, suponiendo que `a` ya está ordenada. Tenga presente que la búsqueda $O(\log n)$ está dominada por el paso de inserción $O(n)$ lento.

```
bisect.insort_right(a, x, lo=0, hi=len(a))
```

```
bisect.insort(a, x, lo=0, hi=len(a))
```

Similar a `insort_left()`, pero inserta `x` en `a` después de cualquier entrada `x` existente.

Ver también:

[Receta SortedCollection](#) que usa bisección para construir una «clase-colección» con todas las funcionalidades, con métodos de búsqueda directos y con soporte para una función-clave (*key-function*). Las claves son procesadas de antemano, para ahorrar llamadas innecesarias a la función clave durante las búsquedas.

8.6.1 Búsqueda en listas ordenadas

Las funciones anteriores `bisect()` son útiles para encontrar puntos de inserción, pero pueden resultar difíciles o engorrosas para tareas de búsqueda habituales. Las cinco funciones que siguen muestran cómo convertirlas en búsquedas estándar para listas ordenadas:

```
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]
```

(continué en la próxima página)

(proviene de la página anterior)

```

return a[i]
raise ValueError

```

8.6.2 Otros ejemplos

La función `bisect()` puede ser útil para búsquedas en tablas numéricas. Este ejemplo utiliza `bisect()` para buscar una calificación de un examen dada por una letra, basada en un conjunto de marcas numéricas ordenadas: 90 o más es una “A”, de 80 a 89 es una “B”, y así sucesivamente:

```

>>> 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']

```

A diferencia de la función `sorted()`, no tiene sentido para las funciones `bisect()` tener los argumentos `key` o `reversed`, porque conduciría a un diseño ineficiente (llamadas sucesivas a funciones de bisección no «recordarían» todas las búsquedas previas con clave).

En su lugar, es mejor buscar en una lista de claves procesadas de antemano para encontrar el índice del registro en cuestión:

```

>>> 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)

```

8.7 array — Arreglos eficientes de valores numéricos

Este módulo define un tipo de objeto que representa un arreglo de valores básicos: caracteres, números enteros y de punto flotante. Los arreglos son tipos de secuencias que se comportan de forma similar a las listas, a excepción que el tipo de objeto guardado es definido. El tipo es especificado al momento de crear el objeto mediante *type code*, que es un carácter simple. Se definen los siguientes tipos:

Código de tipo	Tipo C	Tipo Python	Tamaño mínimo en bytes	Notas
'b'	signed char	int	1	
'B'	unsigned char	int	1	
'u'	Py_UNICODE	Carácter Unicode	2	(1)
'h'	signed short	int	2	
'H'	unsigned short	int	2	
'i'	signed int	int	2	
'I'	unsigned int	int	2	
'l'	signed long	int	4	
'L'	unsigned long	int	4	
'q'	signed long long	int	8	
'Q'	unsigned long long	int	8	
'f'	float	float	4	
'd'	double	float	8	

Notas:

- (1) El código de tipo 'u' corresponde al carácter unicode obsoleto de Python (Py_UNICODE que es wchar_t). Dependiendo de la plataforma, puede ser de 16 bits o de 32 bits.

'u' será eliminada conjuntamente con el resto de Py_UNICODE API.

Deprecated since version 3.3, will be removed in version 4.0.

La representación real de los valores viene determinada por la arquitectura de la maquina (estrictamente hablando, por la implementación de C). El tamaño actual se puede obtener mediante el atributo `itemsize`.

El módulo define los siguientes tipos:

class `array.array` (*typecode* [, *initializer*])

Un nuevo arreglo cuyos elementos son restringidos por *typecode*, e inicializados con el valor opcional *initializer*, el cual debe ser una lista, un *bytes-like object*, o un iterable sobre los elementos del tipo apropiado.

Si dada una lista o un string, el inicializador es pasado a los nuevos métodos `fromlist()`, `frombytes()`, `fromunicode()` del arreglo (ver abajo) para añadir nuevos elementos al arreglo. De forma contraria, el iterable inicializador se pasa al método `extend()`.

Lanza un *evento de auditoría* `array.__new__` con argumentos `typecode`, `initializer`.

array.typecodes

Una cadena de caracteres con todos los códigos de tipos disponible.

Los objetos tipo arreglo soportan operaciones de secuencia ordinarias de indexación, segmentación, concatenación y multiplicación. Cuando se utiliza segmentación, el valor asignado debe ser un arreglo con el mismo código de tipo, en todos los otros casos se lanza `TypeError`. Los arreglos también implementan una interfaz de buffer, y puede ser utilizada en cualquier momento cuando los objetos *bytes-like objects* son soportados.

Los siguientes tipos de datos y métodos también son soportados:

array.typecode

El carácter typecode utilizado para crear el arreglo.

array.itemsize

La longitud en bytes de un elemento del arreglo en su representación interna.

array.append (*x*)

Añade un nuevo elemento con valor *x* al final del arreglo.

array.buffer_info ()

Retorna una tupla (*address*, *length*) con la dirección de memoria actual y la longitud de los elementos en el buffer utilizado para almacenar temporalmente los elementos del arreglo. El tamaño del buffer de memoria es calculado como `array.buffer_info() [1] * array.itemsize`. Ocasionalmente es

práctico cuando trabajamos en interfaces E/S de bajo nivel (de manera inherentemente insegura) que requieren direcciones de memoria, por ejemplo ciertas operaciones `ioctl()`. Los números retornados son válidos mientras el arreglo exista y no se cambie la longitud del mismo.

Nota: Cuando utilizamos objetos tipo arreglo escritos en C o C++ (la única manera de utilizar esta información de forma más efectiva), tiene más sentido utilizar interfaces `buffer` que soporten objetos del tipo arreglo. Este método es mantenido con retro compatibilidad y tiene que ser evitado en el nuevo código. Las interfaces de `buffer` son documentadas en `bufferobjects`.

`array.byteswap()`

«Byteswap» todos los elementos del arreglo. Solo es soportado para valores de tamaño 1,2,3,4 o 8 bytes; para otros valores se lanza `RuntimeError`. Es útil cuando leemos información de un fichero en una máquina con diferente orden de bytes.

`array.count(x)`

Retorna el número de ocurrencias de `x` en el arreglo.

`array.extend(iterable)`

Añade los elementos del `iterable` al final del arreglo. Si el `iterable` es de otro arreglo, este debe ser *exactamente* del mismo tipo; si no, se lanza `TypeError`. Si el `iterable` no es un arreglo, este debe de ser un iterable y sus elementos deben ser del tipo correcto para ser añadidos al arreglo.

`array.frombytes(s)`

Añade los elementos de la cadena de texto, interpretando la cadena de texto como un arreglo de valores máquina (como si se leyera de un fichero utilizando el método `fromfile()`).

Nuevo en la versión 3.2: `fromstring()` se renombra como `frombytes()` por claridad.

`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.

`array.fromlist(list)`

Añade los elementos de la lista. Es equivalente a `for x in list: a.append(x)` excepto que si hay un error de tipo, el arreglo no se modifica.

`array.fromstring()`

Alias obsoleto para `frombytes()`.

Deprecated since version 3.2, will be removed in version 3.9.

`array.fromunicode(s)`

Extiende este arreglo con datos de la cadena de texto unicode. El arreglo debe ser un arreglo tipo `'u'`; de forma contraria se lanza `ValueError`. Utiliza `array.frombytes(unicodestring.encode(enc))` para añadir datos Unicode a un arreglo de algún otro tipo.

`array.index(x)`

Retorna la `i` más pequeña de modo que `i` es el índice de la primera ocurrencia de `x` en el arreglo.

`array.insert(i, x)`

Inserta un nuevo elemento con valor `x` en el arreglo antes de la posición `i`. Si hay valores negativos son tratados como relativos a la posición final del arreglo.

`array.pop([i])`

Elimina el elemento con índice `i` del arreglo y lo retorna. El argumento opcional por defecto es `-1`, en caso de utilizar el argumento por defecto el ultimo elemento es eliminado y retornado.

`array.remove(x)`

Elimina la primera ocurrencia de `x` del arreglo.

`array.reverse()`

Invierte el orden de los elementos en el arreglo.

`array.tobytes()`

Convierte el arreglo en un arreglo de valores máquina y retorna una representación en formato de bytes (la misma secuencia de bytes que se deben escribir en un fichero por el método `tofile()`).

Nuevo en la versión 3.2: `tostring()` se renombra como `tobytes()` para claridad.

`array.tofile(f)`

Escribe todos los elementos (incluido elementos máquina) a el *file object* `f`.

`array.tolist()`

Convierte el arreglo a una lista ordinaria con los mismos elementos.

`array.tostring()`

Alias obsoleto para `tobytes()`.

Deprecated since version 3.2, will be removed in version 3.9.

`array.tounicode()`

Convierte el arreglo a una cadena de texto unicode. El arreglo debe ser un arreglo tipo `'u'`; en caso contrario se lanza `ValueError`. Utiliza `array.tobytes().decode(enc)` para obtener una cadena de texto unicode de un arreglo de algún otro tipo.

Cuando un objeto se imprime o se convierte a una cadena de texto, este se representa como `array(typecode, initializer)`. El *initializer* se omite cuando el arreglo está vacío, de forma contraria es una cadena de caracteres si su *typecode* es `'u'`, de lo contrario es una lista de números. La cadena de caracteres garantiza que es capaz de ser convertida de nuevo a un arreglo con el mismo tipo y valor utilizando `eval()`, hasta que la clase `array` ha sido importada utilizando `from array import array`. Ejemplos:

```
array('l')
array('u', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

Ver también:

Módulo `struct` Empaquetado y desempaquetado de datos binarios heterogéneos.

Módulo `xdrlib` Empaquetado y desempaquetado de datos de Representación de datos externos (XDR) como los utilizados en algunos sistemas de llamadas de procedimientos remotos.

La documentación numérica de Python La extensión numérica de Python (NumPy) define otro tipo de arreglos; consultar <http://www.numpy.org/> para información adicional sobre Python numérico.

8.8 weakref — Referencias débiles

Código Fuente: [Lib/weakref.py](#)

El módulo `weakref` le permite al programador de Python crear *referencias débiles* (*weak references*) a objetos.

A continuación, el término *referente* alude al objeto que es referenciado por una referencia débil.

Una referencia débil a un objeto no es suficiente para mantener al objeto con vida: cuando las únicas referencias que le queden a un referente son referencias débiles, la (*recolección de basura*) es libre de destruir al referente y reusar su memoria para algo más. Sin embargo, hasta que el objeto no sea realmente destruido, la referencia débil puede retornar el objeto incluso si no tiene referencias fuertes.

Un uso principal para las referencias débiles es para implementar caches o mapeados que mantienen objetos grandes, cuando no se desea que un objeto grande no sea mantenido con vida sólo porque aparece en un cache o mapeado.

Por ejemplo, si tienes un número de grandes objetos de imágenes binarias, puedes desear asociar un nombre con cada uno. Si usaras un diccionario de Python para mapear los nombres a imágenes, o imágenes a nombres, los objetos imagen quedarían con vida sólo porque aparecen como valores o llaves en los diccionarios. Las clases `WeakKeyDictionary` y `WeakValueDictionary` que se proporcionan por el módulo `weakref` son una

alternativa, usando referencias débiles para construir mapeados que no mantengan con vida el objeto sólo porque aparecen en el mapeado de objetos. Si, por ejemplo, un objeto imagen es un valor en un `WeakValueDictionary`, entonces cuando las últimas referencias que queden de ese objeto imagen sean las referencias débiles guardadas por mapeados débiles, la recolección de basura puede reclamar el objeto, y sus entradas correspondientes en mapeados débiles son simplemente eliminadas.

`WeakKeyDictionary` y `WeakValueDictionary` usan referencias débiles en sus implementaciones, estableciendo retrollamadas (*callback*) en las referencias débiles que notifiquen a los diccionarios débiles cuando una llave o valor ha sido reclamado por la recolección de basura. `WeakSet` implementa la interfaz `set`, pero mantiene referencias débiles de sus elementos, justo como lo hace `WeakKeyDictionary`.

`finalize` provee una forma directa de registrar una función de limpieza que se llame cuando un objeto es recogido por la recolección de basura. Esto es más simple que configurar una retrollamada en una referencia débil pura, ya que el módulo automáticamente se asegura que el finalizador se mantenga con vida hasta que el objeto sea recolectado.

La mayoría de programas deben descubrir que usar uno de estos tipos de contenedores débiles o la clase `finalize` es todo lo que necesitan – usualmente no es necesario crear tus propias referencias débiles directamente. La maquinaria de bajo nivel está expuesta por el módulo `weakref` para el beneficio de usuarios avanzados.

No todos los objetos pueden ser débilmente referenciados; esos objetos que pueden incluir instancias de clases, funciones escritas en Python (pero no en C), métodos de instancia, conjuntos, conjuntos congelados (*frozensets*), algunos *objetos de archivo*, *generadores*, objetos de tipos, sockets, arreglos, *deque*s, objetos de patrones de expresiones regulares, y objetos código.

Distinto en la versión 3.2: Se añadió el soporte para `thread.lock`, `threading.Lock`, y objetos código.

Varios tipos incorporados como `list` y `dict` no soportan directamente referencias débiles pero pueden añadir soporte al crear una subclase:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)  # this object is weak referenceable
```

CPython implementation detail: Otros tipos incorporados como `tuple` y `int` no soportan referencias débiles incluso cuando son usadas como clase base.

Los tipos de extensiones se pueden hacer para soportar referencias débiles; véase `weakref-support`.

class `weakref.ref(object[, callback])`

Retorna una referencia débil de *object*. El objeto original puede ser recuperado al llamar la referencia del objeto si el referente sigue con vida; si el referente ya no está con vida, llamar a la referencia del objeto causará que se retorne un `None`. Si se proporciona *callback* y no `None`, y el objeto *weakref* retornado aún sigue con vida, el *callback* será llamado cuando el objeto esté a punto de ser finalizado; el objeto de la referencia débil será pasado como el único parámetro a la retrollamada, el referente ya no estará disponible.

Se permite que muchas referencias débiles sean construidas por el mismo objeto. Las retrollamadas registradas por cada referencia débil serán llamados desde la retrollamada registrada más recientemente hasta la retrollamada registrada más antigua.

Las excepciones lanzadas por el *callback* serán anotadas en la salida de error estándar, pero no pueden ser propagadas; son manejadas igual que las excepciones lanzadas por el método `__del__()` de un objeto.

Las referencias débiles son *hashable* si el *objeto* es mapeable. Ellos mantendrán su valor del hash incluso cuando el *objeto* haya sido eliminado. Si `hash()` es llamado por primera vez sólo después de que *object* sea eliminado, la llamada lanzará un `TypeError`.

Las referencias débiles soportan pruebas para igualdad, pero no para ordenación. Si los referentes están todavía con vida, dos referencias tiene la misma relación de igualdad como sus referentes (sin importar el *callback*). Si un referente ha sido eliminado, las referencias son iguales sólo si el objetos de referencia son el mismo objeto.

Es un tipo del que se puede crear una subclase en vez de una función de fábrica.

__callback__

Este atributo de sólo lectura retorna la llamada que está asociada actualmente con el *weakref*. Si no hay retrollamadas o si el referente del *weakref* no está con vida entonces este atributo tendrá de valor `None`.

Distinto en la versión 3.4: Se añadió el atributo `__callback__`.

`weakref.proxy(object[, callback])`

Retorna un proxy a *object* que usa una referencia débil. Esto soporta el uso del proxy en la mayoría de los contextos en vez de requerir la dereferencia explícita usada con los objetos de referencia débil. El objeto retornado tendrá un tipo `ProxyType` o `CallableProxyType`, dependiendo si *object* es invocable. Objetos Proxy no son *hashable* independiente de la referencia; esto evita un número de problemas relacionados a su naturaleza mutable fundamental, y previene su uso como claves de diccionario. *callback* es el mismo como el parámetro del mismo nombre de la función `ref()`.

Distinto en la versión 3.8: Se extendió el soporte de operadores en objetos proxy para incluir los operadores de multiplicación de matrices `@` and `@=`.

`weakref.getweakrefcount(object)`

Retorna el número de referencias débiles y proxies que refieren a *object*.

`weakref.getweakrefs(object)`

Retorna una lista de todas las referencias débiles y objetos proxy que refieren a *object*.

class `weakref.WeakKeyDictionary([dict])`

Clase de mapeado que referencia llaves débilmente. Las entradas en el diccionario serán descartadas cuando no haya una referencia fuerte a la llave. Esto puede ser usado para asociar datos con un objeto apropiado por otras partes de una aplicación sin añadir atributos a esos objetos. Esto puede ser especialmente útil con objetos que sobre escriben atributos de acceso.

Los objetos *WeakKeyDictionary* tiene un método adicional que expone las referencias internas directamente. Las referencias no tienen garantía de estar con «vida» en el momento en que son usadas, por lo que el resultado de llamar las referencias necesita ser revisado antes de ser usado. Esto puede ser usado para evitar crear referencias que causen que recolector de basura mantenga las llaves en existencia más tiempo del que necesitan.

`WeakKeyDictionary.keyrefs()`

Retorna un iterable de las referencias débiles a las llaves.

class `weakref.WeakValueDictionary([dict])`

Clase de mapeado que referencia valores débilmente. Las entradas en el diccionario serán descartadas cuando ya no existan las referencias fuertes a los valores.

Los objetos *WeakValueDictionary* tienen un método adicional que tiene los mismos problemas que el método `keyrefs()` de los objetos *WeakKeyDictionary*.

`WeakValueDictionary.valuerefs()`

Retorna un iterable de las referencias débiles a los valores.

class `weakref.WeakSet([elements])`

Clase Conjunto que mantiene referencias débiles a sus elementos. Un elemento será descartado cuando ya no existan referencias fuertes.

class `weakref.WeakMethod(method)`

Una subclase *ref* personalizada que simula una referencia débil a un método vinculado (i.e., un método definido en una clase y visto en una instancia). Ya que un método atado es efímero, una referencia débil estándar no puede mantenerlo. El *WeakMethod* tiene un código especial para recrear el método atado hasta que o el objeto o la función original muera:

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r()()
```

(continué en la próxima página)

(proviene de la página anterior)

```
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

Nuevo en la versión 3.4.

class `weakref.finalize(obj, func, *args, **kwargs)`

Retorna un objeto finalizador invocable que será llamado cuando *obj* sea recolectado por el recolector de basura. A diferencia de referencias débiles ordinarias, un finalizador siempre sobrevivirá hasta que el objeto de referencia sea recolectado, simplificando enormemente la gestión del ciclo de vida.

Se considera a un finalizador como *vivo* hasta que sea llamado (o explícitamente o en la recolección de basura), y después que esté *muerto*. Llamar a un finalizador vivo retorna el resultado de evaluar `func(*arg, **kwargs)`, mientras que llamar a un finalizador muerto retorna *None*.

Las excepciones lanzadas por retrollamadas de finalizadores durante la recolección de basura serán mostradas en la salida de error estándar, pero no pueden ser propagadas. Son gestionados de la misma forma que las excepciones lanzadas del método `__del__()` de un objeto o la retrollamada de una referencia débil.

Cuando el programa sale, cada finalizador vivo que quede es llamado a menos que su atributo `atexit` sea falso. Son llamados en el orden reverso de creación.

Un finalizador nunca invocará su retrollamada durante la última parte del *interpreter shutdown* cuando los módulos globales están sujetos a ser reemplazados por *None*.

__call__()

Si *self* está vivo entonces lo marca como muerto y retorna el resultado de llamar a `func(*args, **kwargs)`. Si *self* está muerto entonces retorna *None*.

detach()

Si *self* está vivo entonces lo marca como muerto y retorna la tupla `(obj, func, args, kwargs)`. Si *self* está muerto entonces retorna *None*.

peek()

Si *self* está vivo entonces retorna la tupla `(obj, func, args, kwargs)`. Si *self* está muerto entonces retorna *None*.

alive

Propiedad que es verdadera si el finalizador está vivo, caso contrario, falso.

atexit

Una propiedad booleana con permisos de escritura que por defecto es verdadero. Cuando el programa sale, llama a todos los finalizadores vivos que queden para los cuales `atexit` es verdadero. Ellos son llamados en el orden reverso de creación.

Nota: Es importante asegurar que *func*, *args* y *kwargs* no sean dueños de ninguna referencia a *obj*, o directamente o indirectamente, ya que de otra manera *obj* nunca será recolectado por el recolector de basura. En particular, *func* no debe ser un método vinculado de *obj*.

Nuevo en la versión 3.4.

`weakref.ReferenceType`

El objeto de tipo para objetos de referencias débiles.

`weakref.ProxyType`

El objeto de tipo para proxies de objetos que no son invocables.

`weakref.CallableProxyType`

El objeto de tipo para proxies de objetos invocables.

weakref.ProxyTypes

Una secuencia que contiene todos los objetos de tipo para los proxies. Esto puede hacerlo más simple para pruebas si un objeto es un proxy sin ser dependiente en nombrar a ambos tipos proxy.

Ver también:

PEP 205 - Referencias Débiles La propuesta y lógica de esta característica, incluyendo los enlaces a implementaciones tempranas e información acerca de características similares en otros lenguajes.

8.8.1 Objetos de Referencias Débiles

Los objetos de referencias débiles no tiene métodos y atributos aparte de `ref.__callback__`. Un objeto de referencia débil permite que el referente sea obtenido, si todavía existe, al llamarlo:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

Si el referente no existe, llamar al objeto de referencia retorna *None*:

```
>>> del o, o2
>>> print(r())
None
```

Probar que un objeto de referencia débil está todavía con vida debe ser hecho usando la expresión `ref() is not None`. Normalmente, el código de aplicación que necesite usar un objeto de referencia debe seguir este patrón:

```
# 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()
```

Usar una prueba separada para «vividad» crea una condición de carrera en aplicaciones con hilos; otro hilo puede hacer que una referencia débil sea invalidada antes de que la referencia débil sea llamada; El modismo mostrado arriba es seguro en aplicaciones con hilos también como aplicaciones de un sólo hilo.

Versiones especializadas de objetos *ref* pueden ser creadas a través de creación por subclase. Esto es usado en la implementación de *WeakValueDictionary* para reducir la memoria elevada por cada entrada en el mapeado. Esto puede ser lo más útil para asociar información adicional con un referencia, pero también puede ser usado para insertar procesamiento adicional en llamadas para recuperar el referente.

Este ejemplo muestra como una subclase de *ref* puede ser usado para guardar información adicional sobre un objeto y afectar el valor que se retorna cuando el referente es accedido:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, /, **annotations):
        super().__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
```

(continué en la próxima página)

(proviene de la página anterior)

```

        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().__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob

```

8.8.2 Ejemplo

Este simple ejemplo muestra como una aplicación puede usar objetos ID para recuperar objetos que han sido visto antes. Los ID de los objetos pueden ser usados en otras estructuras de datos sin forzar que los objetos permanezcan con vida, pero los objetos pueden aún pueden ser recuperados por el ID si lo hacen.

```

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]

```

8.8.3 Objetos Finalizadores

El principal beneficio de usar *finalize* es que hace simple registrar una retrollamada sin necesitar preservar el objeto finalizador retornado. Por ejemplo

```

>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!

```

El finalizador puede ser llamado directamente también. Sin embargo, el finalizador invocará la retrollamada como máximo una vez.

```

>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK

```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> assert not f.alive
>>> f()                                # callback not called because finalizer dead
>>> del obj                            # callback not called because finalizer dead
```

Puedes de-registrar un finalizador usando su método `detach()`. Esto mata el finalizador y retorna los argumentos pasados al constructor cuando fue creado.

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<...Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

A menos que pongas el atributo `atexit` a `False`, un finalizador será llamado cuando el programa salga si todavía está con vida. Por ejemplo

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

8.8.4 Comparando finalizadores con los métodos `__del__()`

Suponga que queremos crear una clase cuyas instancias representan directorios temporales. Los directorios deben ser eliminados con sus contenidos cuando el primero de los siguiente eventos ocurre:

- el objeto es recolectado por el recolector de basura,
- el método `remove()` del objeto es llamado, o
- el programa sale.

Nosotros podemos intentar implementar la clase usando el método `__del__()` como sigue:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

Empezando con Python 3.4, Los métodos `__del__()` ya no previenen ciclos de referencia de ser recolectado como basura, y los módulos globales ya no fuerzan `None` durante *interpreter shutdown*. Por lo que este código debe trabajar sin ningún problema en CPython.

Sin embargo, la gestión de métodos `__del__()` es notoriamente específico por la implementación, ya que depende de detalles internos de la implementación del recolector de basura del intérprete.

Una alternativa más robusta puede ser para definir un finalizador que sólo hace referencia a funciones específicas y objetos que necesite, en vez de tener acceso al estado completo del objeto:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

    @property
    def removed(self):
        return not self._finalizer.alive
```

Definido así, nuestro finalizador sólo recibe una referencia a los detalles que necesita limpiar los directorios apropiadamente. Si el objeto nueva llega a ser recolectado como basura el finalizador aún será llamado al salir.

La otra ventaja de *weakref* basados en finalizadores es que ellos pueden ser usados para registrar finalizadores para clases donde la definición es controlado por terceros, como un código que corre cuando un módulo es *unloaded*:

```
import weakref, sys
def unloading_module():
    # implicit reference to the module globals from the function body
    weakref.finalize(sys.modules[__name__], unloading_module)
```

Nota: Si creas un objeto finalizador en un hilo *daemon* sólo como el programa sale entonces hay la posibilidad de que el finalizador no llegue a ser llamado. Sin embargo, en un hilo demoníaco `atexit.register()`, `try: ... finally: ...` y `with: ...` no garantizan que la limpieza ocurra tampoco.

8.9 types — Creación de tipos dinámicos y nombres para tipos integrados

Código fuente: [Lib/types.py](#)

Este módulo define funciones de utilidad para ayudar en la creación dinámica de tipos nuevos.

Este también define nombres para algunos tipos de objetos que son utilizados por el intérprete estándar de Python, pero no expuestos como integrados como lo son *int* o *str*.

Por último, este proporciona algunas clases de utilidad y funciones adicionales relacionadas con tipos que no son lo suficientemente fundamentales como para ser integradas.

8.9.1 Creación dinámica de tipos

`types.new_class` (*name*, *bases=()*, *kwds=None*, *exec_body=None*)

Crea un objeto de clase dinámicamente utilizando la metaclasses adecuada.

Los tres primeros argumentos son los componentes que componen un encabezado de definición de clase: el nombre de la clase, las clases base (en orden), los argumentos por palabra clave (tal como `metaclass`).

The *exec_body* argument is a callback that is used to populate the freshly created class namespace. It should accept the class namespace as its sole argument and update the namespace directly with the class contents. If no callback is provided, it has the same effect as passing in `lambda ns: None`.

Nuevo en la versión 3.3.

`types.prepare_class` (*name*, *bases*=(), *kwds*=None)

Calcula la metaclass adecuada y crea el espacio de nombre de clase.

Los argumentos son los componentes que constituyen un encabezado de definición de clase: el nombre de la clase, las clases base (en orden) y los argumentos de palabra clave (como `metaclass`).

El valor retornado es una tupla de 3: `metaclass`, `namespace`, `kwds`

metaclass es la metaclass adecuada, *namespace* es el espacio de nombre de clase preparado y *kwds* es una copia actualizada del pasado en el argumento *kwds* con cualquier entrada '`metaclass`' eliminada. Si no se pasa ningún argumento *kwds*, será un diccionario vacío.

Nuevo en la versión 3.3.

Distinto en la versión 3.6: El valor predeterminado para el elemento `namespace` de la tupla retornada ha cambiado. Ahora una asignación de inserción-orden-conservación es utilizada cuando la metaclass no tiene un método `__prepare__`.

Ver también:

metaclasses Detalles completos del proceso de creación de clases soportado por estas funciones

PEP 3115 - Metaclasses en Python 3000 Se presenta el *hook* de espacio de nombres `__prepare__`

`types.resolve_bases` (*bases*)

Resuelve las entradas MRO dinámicamente según lo especificado por **PEP 560**.

Esta función busca elementos en *bases* que no son instancias de *type* y retorna una tupla donde cada uno de estos objetos que tiene un método `__mro_entries__` se reemplaza con un resultado desempquetado de llamar a este método. Si un elemento *bases* es una instancia de *type* o no tiene un método `__mro_entries__`, se incluye en el retorno la tupla sin cambios.

Nuevo en la versión 3.7.

Ver también:

PEP 560 - Soporte principal para módulos de tipo y tipos genéricos

8.9.2 Tipos de Interpretes Estándar

Este módulo proporciona nombres para muchos de los tipos necesarios para implementar un intérprete de Python. Esto evita deliberadamente incluir algunos de los tipos que surgen sólo accidentalmente durante el procesamiento, tal como el tipo `listiterator`.

El uso típico de estos nombres es para verificar `isinstance()` o `issubclass()`.

Si se crea una instancia de cualquiera de estos tipos, tenga en cuenta que las firmas pueden variar entre las versiones de Python.

Los nombres estándar son definidos para los siguientes tipos:

`types.FunctionType`

`types.LambdaType`

El tipo de funciones definidas por el usuario y funciones creadas por expresiones `lambda`.

Raises an *auditing event* `function.__new__` with argument `code`.

The audit event only occurs for direct instantiation of function objects, and is not raised for normal compilation.

`types.GeneratorType`

El tipo de iterador *generator* de objetos, creados por funciones generadoras.

`types.CoroutineType`

El tipo de objetos *coroutine*, creados por funciones `async def`.

Nuevo en la versión 3.5.

types.AsyncGeneratorType

El tipo de iterador *asynchronous generator* de objetos, creados por funciones generadoras asíncronas.

Nuevo en la versión 3.6.

class types.CodeType (***kwargs*)

El tipo de objetos de código cómo los retornados por *compile()*.

Lanza un *evento auditor* *code.__new__* con los argumentos *code*, *filename*, *name*, *argcount*, *posonlyargcount*, *kwonlyargcount*, *nlocals*, *stacksize*, *flags*.

Note that the audited arguments may not match the names or positions required by the initializer. The audit event only occurs for direct instantiation of code objects, and is not raised for normal compilation.

replace (***kwargs*)

Retorna una copia del objeto de código con nuevos valores para los campos especificados.

Nuevo en la versión 3.8.

types.CellType

El tipo de objetos de celda: estos objetos se utilizan como contenedores para las variables libres de una función.

Nuevo en la versión 3.8.

types.MethodType

El tipo de métodos de instancias de clase definidas por el usuario.

types.BuiltinFunctionType**types.BuiltinMethodType**

El tipo de funciones integradas como *len()* o *sys.exit()* y métodos de clases integradas. (Aquí, el término «incorporado» significa «escrito en C».)

types WrapperDescriptorType

El tipo de métodos de algunos tipos de datos integrados y clases base como *object.__init__()* o *object.__lt__()*.

Nuevo en la versión 3.7.

types.MethodWrapperType

El tipo de métodos *bound* de algunos tipos de datos integrados y clases base. Por ejemplo, es el tipo de *object().__str__*.

Nuevo en la versión 3.7.

types.MethodDescriptorType

El tipo de métodos de algunos tipos de datos integrados como *str.join()*.

Nuevo en la versión 3.7.

types.ClassMethodDescriptorType

El tipo de métodos de clase *unbound* de algunos tipos de datos integrados como *dict.__dict__['fromkeys']*.

Nuevo en la versión 3.7.

class types.ModuleType (*name*, *doc=None*)

The type of *modules*. The constructor takes the name of the module to be created and optionally its *docstring*.

Nota: Utilice *importlib.util.module_from_spec()* para crear un nuevo módulo si desea establecer los diversos atributos controlados por importación.

__doc__

El *docstring* del módulo. El valor predeterminado es *None*.

__loader__

El *loader* que cargó el módulo. El valor predeterminado es *None*.

This attribute is to match `importlib.machinery.ModuleSpec.loader` as stored in the attr: `__spec__` object.

Nota: A future version of Python may stop setting this attribute by default. To guard against this potential change, preferably read from the `__spec__` attribute instead or use `getattr(module, "__loader__", None)` if you explicitly need to use this attribute.

Distinto en la versión 3.4: El valor predeterminado es `None`. Anteriormente el atributo era opcional.

`__name__`

The name of the module. Expected to match `importlib.machinery.ModuleSpec.name`.

`__package__`

A cuál *package* pertenece un módulo. Si el módulo es de nivel superior (es decir, no una parte de algún paquete específico), el atributo debe establecerse en `'`, de lo contrario debe establecerse en el nombre del paquete (el cual puede ser `__name__` si el módulo es un paquete). El valor predeterminado es `None`.

This attribute is to match `importlib.machinery.ModuleSpec.parent` as stored in the attr: `__spec__` object.

Nota: A future version of Python may stop setting this attribute by default. To guard against this potential change, preferably read from the `__spec__` attribute instead or use `getattr(module, "__package__", None)` if you explicitly need to use this attribute.

Distinto en la versión 3.4: El valor predeterminado es `None`. Anteriormente el atributo era opcional.

`__spec__`

A record of the the module's import-system-related state. Expected to be an instance of `importlib.machinery.ModuleSpec`.

Nuevo en la versión 3.4.

class `types.TracebackType` (*tb_next*, *tb_frame*, *tb_lasti*, *tb_lineno*)

El tipo de objetos *traceback* tal como los encontrados en `sys.exc_info()` [2].

Consulte la referencia de lenguaje para obtener detalles de los atributos y operaciones disponibles, y orientación sobre cómo crear *tracebacks* dinámicamente.

`types.FrameType`

El tipo de objetos de marco como se encuentra en `tb.tb_frame` si `tb` es un objeto *traceback*.

Consulte la referencia de lenguaje para obtener más información sobre los atributos y operaciones disponibles.

`types.GetSetDescriptorType`

El tipo de objetos definidos en módulos de extensión con `PyGetSetDef`, como `FrameType.f_locals` o `array.array.typecode`. Este tipo se utiliza como descriptor para los atributos de objeto; tiene el mismo propósito que el tipo *property*, pero para las clases definidas en los módulos de extensión.

`types.MemberDescriptorType`

El tipo de objetos definidos en módulos de extensión con `PyMemberDef`, como `datetime.timedelta`. Este tipo se utiliza como descriptor para miembros de datos C simples que utilizan funciones de conversión estándar; tiene el mismo propósito que el tipo *property*, pero para las clases definidas en los módulos de extensión.

CPython implementation detail: En otras implementaciones de Python, este tipo puede ser idéntico a `GetSetDescriptorType`.

class `types.MappingProxyType` (*mapping*)

Proxy de solo lectura de un mapeo. Proporciona una vista dinámica en las entradas de la asignación, lo que significa que cuando cambia la asignación, la vista refleja estos cambios.

Nuevo en la versión 3.3.

key in proxy

Retorna `True` si la asignación subyacente tiene una clave *key*, de lo contrario `False`.

proxy[key]

Retorna el elemento de la asignación subyacente con la clave *key*. Lanza un `KeyError` si *key* no está en la asignación subyacente.

iter(proxy)

Retorna un iterador sobre las claves de la asignación subyacente. Este es un método abreviado para `iter(proxy.keys())`.

len(proxy)

Retorna el número de elementos de la asignación subyacente.

copy()

Retorna una copia superficial de la asignación subyacente.

get(key[, default])

Retorna el valor de *key* si *key* está en la asignación subyacente, de lo contrario *default*. Si no se proporciona *default*, el valor predeterminado es `None`, por lo que este método nunca lanza un `KeyError`.

items()

Retorna una nueva vista de los elementos de la asignación subyacente (en pares (*key*, *value*)).

keys()

Retorna una nueva vista de las claves de la asignación subyacente.

values()

Retorna una nueva vista de los valores de la asignación subyacente.

8.9.3 Clases y funciones de utilidad adicionales

class types.SimpleNamespace

Una subclase simple *object* que proporciona acceso de atributo a su espacio de nombre, así como una representación significativa.

A diferencia de *object*, con `SimpleNamespace` puede agregar y eliminar atributos. Si un objeto `SimpleNamespace` se inicializa con argumentos de palabra clave, estos se agregan directamente al espacio de nombres subyacente.

El tipo es aproximadamente equivalente al código siguiente:

```
class SimpleNamespace:
    def __init__(self, /, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        keys = sorted(self.__dict__)
        items = ("{}={!r}".format(k, self.__dict__[k]) for k in keys)
        return "{}({})".format(type(self).__name__, ", ".join(items))

    def __eq__(self, other):
        if isinstance(self, SimpleNamespace) and isinstance(other,
↪SimpleNamespace):
            return self.__dict__ == other.__dict__
        return NotImplemented
```

`SimpleNamespace` puede ser útil como reemplazo para `class NS: pass`. Sin embargo, para un tipo de registro estructurado, utilice `namedtuple()` en su lugar.

Nuevo en la versión 3.3.

types.DynamicClassAttribute(fget=None, fset=None, fdel=None, doc=None)

Acceso de atributo de ruta en una clase para `__getattr__`.

Se trata de un descriptor, que se utiliza para definir atributos que actúan de forma diferente cuando se accede a través de una instancia y a través de una clase. El acceso a la instancia sigue siendo normal, pero el acceso a un atributo a través de una clase se enrutará al método `__getattr__` de la clase; esto se hace lanzando `AttributeError`.

Esto permite tener propiedades activas en una instancia y tener atributos virtuales en la clase con el mismo nombre (consulte `Enum` para obtener un ejemplo).

Nuevo en la versión 3.4.

8.9.4 Funciones de utilidad de corutina

`types.coroutine(gen_func)`

Esta función transforma una función *generador* en una función *coroutine* que retorna una corrutina basada en un generador. La corrutina basada en un generador sigue siendo un *generator iterator*, pero también se considera un objeto *coroutine* y es *awaitable*. Sin embargo, no puede necesariamente implementar el método `__await__()`.

Si *gen_func* es una función generadora, se modificará en el lugar.

Si *gen_func* no es una función generadora, se envolverá. Si retorna una instancia de `collections.abc.Generator`, la instancia se ajustará en un objeto proxy *awaitable*. Todos los demás tipos de objetos se retornarán tal cual.

Nuevo en la versión 3.5.

8.10 copy — Operaciones de copia superficial y profunda

Source code: [Lib/copy.py](#)

Las declaraciones de asignación en Python no copian objetos, crean enlaces entre un objetivo y un objeto. Para colecciones que son mutables o que contienen elementos mutables, a veces se necesita una copia para que uno pueda cambiar una copia sin cambiar la otra. Este módulo proporciona operaciones genéricas de copia superficial y profunda (explicadas a continuación).

Resumen de la interfaz:

`copy.copy(x)`

Retorna una copia superficial de *x*.

`copy.deepcopy(x[, memo])`

Retorna una copia profunda de *x*.

exception `copy.Error`

Levantado para errores específicos del módulo.

La diferencia entre copia superficial y profunda solo es relevante para objetos compuestos (objetos que contienen otros objetos, como listas o instancias de clase):

- Una copia superficial (*shallow copy*) construye un nuevo objeto compuesto y luego (en la medida de lo posible) inserta *references* en él a los objetos encontrados en el original.
- Una copia profunda (*deep copy*) construye un nuevo objeto compuesto y luego, recursivamente, inserta *copias* en él de los objetos encontrados en el original.

A menudo existen dos problemas con las operaciones de copia profunda que no existen con las operaciones de copia superficial:

- Los objetos recursivos (objetos compuestos que, directa o indirectamente, contienen una referencia a sí mismos) pueden causar un bucle recursivo.
- Debido a que la copia profunda copia todo, puede copiar demasiado, como los datos que deben compartirse entre copias.

La función `deepcopy()` evita estos problemas al:

- mantener un diccionario `memo` de objetos ya copiados durante la pasada de copia actual; y
- dejando que las clases definidas por el usuario anulen la operación de copia o el conjunto de componentes copiados.

Este módulo no copia tipos como módulo, método, seguimiento de pila, marco de pila, archivo, socket, ventana, matriz ni ningún tipo similar. «Copia» funciones y clases (superficiales y profundas), retornando el objeto original sin cambios; Esto es compatible con la forma en que son tratados por el módulo `pickle`.

Se pueden hacer copias superficiales de los diccionarios usando `dict.copy()`, y de las listas mediante la asignación de una porción de la lista completa, por ejemplo, `copied_list = original_list[:]`.

Las clases pueden usar las mismas interfaces para controlar la copia que usan para controlar el *pickling*. Consulte la descripción del módulo `pickle` para obtener información sobre estos métodos. De hecho, el módulo `copy` utiliza las funciones de `pickle` registradas del módulo `copyreg`.

Para que una clase defina su propia implementación de copia, puede definir métodos especiales `__copy__()` y `__deepcopy__()`. El primero se llama para implementar la operación de copia superficial; no se pasan argumentos adicionales. Este último está llamado a implementar la operación de copia profunda; se pasa un argumento, el diccionario `memo`. Si la implementación `__deepcopy__()` necesita hacer una copia profunda de un componente, debe llamar a la función `deepcopy()` con el componente como primer argumento y el diccionario `memo` como segundo argumento.

Ver también:

Módulo `pickle` Discusión de los métodos especiales utilizados para apoyar la recuperación y restauración del estado del objeto.

8.11 pprint — Impresión bonita de datos

Código fuente: `Lib/pprint.py`

El módulo `pprint` proporciona la capacidad de «imprimir de forma bonita» estructuras de datos arbitrarias de Python de manera que se puede utilizar como entrada para el intérprete. Si las estructuras formateadas incluyen objetos que no son tipos fundamentales de Python, es posible que la representación no sea válida como tal para el intérprete. Esto puede darse si se incluyen objetos como archivos, sockets o clases, así como muchos otros objetos que no se pueden representar como literales de Python.

La representación formateada mantiene los objetos en una sola línea siempre que sea posible y los divide en varias líneas si no encajan dentro del ancho permitido. Se deben crear objetos `PrettyPrinter` de forma explícita si se necesita ajustar la restricción de ancho.

Los diccionarios se ordenan por clave antes de que se calcule la representación en pantalla.

El módulo `pprint` define una sola clase:

```
class pprint.PrettyPrinter (indent=1, width=80, depth=None, stream=None, *, compact=False,
                             sort_dicts=True)
```

Construye una instancia de `PrettyPrinter`. Este constructor acepta varios argumentos por palabra clave. Se puede establecer un flujo de salida usando la palabra clave `stream`; el único método utilizado en el objeto `stream` es el método `write()` del protocolo de archivo. Si no se especifica, `PrettyPrinter` adopta `sys.stdout` por defecto. La cantidad de sangría agregada para cada nivel recursivo se especifica mediante `indent`; por defecto es uno. Otros valores pueden hacer que la salida se vea un poco extraña, pero pueden hacer más fácil la visualización de los anidamientos. El número de niveles que se pueden mostrar se controla mediante `depth`; si la estructura de datos que se muestra es demasiado profunda, el siguiente nivel contenido se reemplaza por `...`. De forma predeterminada, no hay restricciones en la profundidad de los objetos que se formatean. El ancho de salida deseado se restringe mediante el parámetro `width`; el valor predeterminado es 80 caracteres. Si no se puede formatear una estructura dentro del límite de ancho establecido, se ajustará lo mejor posible. Si `compact` es `False` (el valor por defecto), cada elemento de una secuencia larga se formateará en una línea

separada. Si *compact* es *True*, en cada línea de salida se formatearán todos los elementos que quepan dentro del ancho definido. Si *sort_dicts* es *True* (el valor por defecto), los diccionarios se formatearán con sus claves ordenadas; de lo contrario, se mostrarán según orden de inserción.

Distinto en la versión 3.4: Añadido el argumento *compact*.

Distinto en la versión 3.8: Añadido el argumento *sort_dicts*.

```
>>> 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']
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> 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', (...))))))
```

El módulo `pprint` también proporciona varias funciones de atajo:

`pprint.pformat(object, indent=1, width=80, depth=None, *, compact=False, sort_dicts=True)`

Retorna la representación formateada de *object* como una cadena de caracteres. *indent*, *width*, *depth*, *compact* y *sort_dicts* se pasarán al constructor de `PrettyPrinter` como parámetros de formato.

Distinto en la versión 3.4: Añadido el argumento *compact*.

Distinto en la versión 3.8: Añadido el argumento *sort_dicts*.

`pprint.pp(object, *args, sort_dicts=False, **kwargs)`

Imprime la representación formateada de *object* seguida de una nueva línea. Si *sort_dicts* es *False* (el valor por defecto), los diccionarios se mostrarán con sus claves según orden de inserción, de lo contrario, las claves del diccionario serán ordenadas. *args* y *kwargs* se pasarán a `pprint()` como parámetros de formato.

Nuevo en la versión 3.8.

`pprint.pprint(object, stream=None, indent=1, width=80, depth=None, *, compact=False, sort_dicts=True)`

Imprime la representación formateada de *object* en *stream*, seguida de una nueva línea. Si *stream* es *None*, se usa `sys.stdout`. Esta función se puede usar en el intérprete interactivo en lugar de la función `print()` para inspeccionar valores (incluso puedes reasignar `print = pprint.pprint` para su uso dentro del ámbito). *indent*, *width*, *depth*, *compact* y *sort_dicts* se pasarán al constructor de `PrettyPrinter` como parámetros de formato.

Distinto en la versión 3.4: Añadido el argumento *compact*.

Distinto en la versión 3.8: Añadido el argumento *sort_dicts*.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
```

(continué en la próxima página)

(proviene de la página anterior)

```
'spam',
'eggs',
'lumberjack',
'knights',
'ni']
```

`pprint.isreadable(object)`

Determina si la representación formateada de *object* es «legible» o si puede usarse para reconstruir el objeto usando `eval()`. Siempre retorna `False` para objetos recursivos.

```
>>> pprint.isreadable(stuff)
False
```

`pprint.isrecursive(object)`

Determina si *object* requiere una representación recursiva.

Una función extra de soporte es también definida:

`pprint.saferepr(object)`

Retorna una representación en forma de cadena de caracteres de *object*, que está protegida contra estructuras de datos recursivas. Si la representación de *object* presenta una entrada recursiva, dicha referencia recursiva se representará como `<Recursion on typename with id=number>`. Además, la representación no tendrá otro formato.

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni'
↪]"
```

8.11.1 Objetos *PrettyPrinter*

Las instancias de *PrettyPrinter* tienen los siguientes métodos:

`PrettyPrinter.pformat(object)`

Retorna la representación formateada de *object*. Tiene en cuenta las opciones pasadas al constructor de la clase *PrettyPrinter*.

`PrettyPrinter.pprint(object)`

Imprime la representación formateada de *object* en la secuencia configurada, seguida de una nueva línea.

Los siguientes métodos proporcionan las implementaciones para las funciones correspondientes con los mismos nombres. Usar estos métodos en una instancia es algo más eficiente, ya que no es necesario crear nuevos objetos *PrettyPrinter*.

`PrettyPrinter.isreadable(object)`

Determina si la representación formateada de *object* es «legible» o si se puede usar para reconstruir su valor usando `eval()`. Se debe tener en cuenta que se retorna `False` para objetos recursivos. Si el parámetro *depth* de *PrettyPrinter* es proporcionado y el objeto es más profundo de lo permitido, también se retorna `False`.

`PrettyPrinter.isrecursive(object)`

Determina si *object* requiere una representación recursiva.

Este método se proporciona como un punto de entrada o método de enlace automático (*hook* en inglés) para permitir que las subclases modifiquen la forma en que los objetos se convierten en cadenas de caracteres. La implementación por defecto utiliza la implementación interna de `saferepr()`.

`PrettyPrinter.format(object, context, maxlevels, level)`

Retorna tres valores: la versión formateada de *object* como una cadena de caracteres, una bandera que indica si el resultado es legible y una bandera que indica si se detectó recursividad. El primer argumento es el objeto a representar. El segundo es un diccionario que contiene la `id()` de los objetos que son parte del contexto de representación actual (contenedores directos e indirectos para *object* que están afectando a la representación),

como las claves; si es necesario representar un objeto que ya está representado en *context*, el tercer valor de retorno será True. Las llamadas recursivas al método `format()` deben agregar entradas adicionales a los contenedores de este diccionario. El tercer argumento, *maxlevels*, proporciona el límite máximo de recursividad; su valor por defecto es 0. Este argumento debe pasarse sin modificaciones a las llamadas recursivas. El cuarto argumento, *level*, da el nivel actual; las llamadas recursivas sucesivas deben pasar un valor menor que el de la llamada actual.

8.11.2 Ejemplo

Para demostrar varios usos de la función `pprint()` y sus parámetros, busquemos información sobre un proyecto en PyPI:

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('https://pypi.org/pypi/sampleproject/json') as resp:
...     project_info = json.load(resp)['info']
```

En su forma básica, la función `pprint()` muestra el objeto completo:

```
>>> pprint.pprint(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': ['Development Status :: 3 - Alpha',
                 'Intended Audience :: Developers',
                 'License :: OSI Approved :: MIT License',
                 'Programming Language :: Python :: 2',
                 'Programming Language :: Python :: 2.6',
                 'Programming Language :: Python :: 2.7',
                 'Programming Language :: Python :: 3',
                 'Programming Language :: Python :: 3.2',
                 'Programming Language :: Python :: 3.3',
                 'Programming Language :: Python :: 3.4',
                 'Topic :: Software Development :: Build Tools'],
 'description': 'A sample Python project\n'
               '=====\n'
               '\n'
               'This is the description file for the project.\n'
               '\n'
               'The file should use UTF-8 encoding and be written using '
               'ReStructured Text. It\n'
               'will be used to generate the project webpage on PyPI, and '
               'should be written for\n'
               'that purpose.\n'
               '\n'
               'Typical contents for this file would include an overview of '
               'the project, basic\n'
               'usage examples, etc. Generally, including the project '
               'changelog in here is not\n'
               'a good idea, although a simple "What\'s New" section for the '
               'most recent version\n'
               'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {'last_day': -1, 'last_month': -1, 'last_week': -1},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
```

(continué en la próxima página)

(proviene de la página anterior)

```
'maintainer_email': None,
'name': 'sampleproject',
'package_url': 'https://pypi.org/project/sampleproject/',
'platform': 'UNKNOWN',
'project_url': 'https://pypi.org/project/sampleproject/',
'project_urls': {'Download': 'UNKNOWN',
                  'Homepage': 'https://github.com/pypa/sampleproject'},
'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
'requires_dist': None,
'requires_python': None,
'summary': 'A sample Python project',
'version': '1.2.0'}
```

El resultado puede limitarse a una cierta profundidad asignando un valor al argumento *depth* (... se utiliza para contenidos más «profundos»):

```
>>> pprint.pprint(project_info, depth=1)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
               '=====\n'
               '\n'
               'This is the description file for the project.\n'
               '\n'
               'The file should use UTF-8 encoding and be written using '
               'ReStructured Text. It\n'
               'will be used to generate the project webpage on PyPI, and '
               'should be written for\n'
               'that purpose.\n'
               '\n'
               'Typical contents for this file would include an overview of '
               'the project, basic\n'
               'usage examples, etc. Generally, including the project '
               'changelog in here is not\n'
               'a good idea, although a simple "What\'s New" section for the '
               'most recent version\n'
               'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

Además, se puede establecer un valor máximo de caracteres por línea asignando un valor al parámetro *width*. Si un objeto largo no se puede dividir, el valor dado al ancho se anulará y será excedido:

```
>>> pprint.pprint(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
               '=====\n'
               '\n'
               'This is the description file for the '
               'project.\n'
               '\n'
               'The file should use UTF-8 encoding and be '
               'written using ReStructured Text. It\n'
               'will be used to generate the project '
               'webpage on PyPI, and should be written '
               'for\n'
               'that purpose.\n'
               '\n'
               'Typical contents for this file would '
               'include an overview of the project, '
               'basic\n'
               'usage examples, etc. Generally, including '
               'the project changelog in here is not\n'
               'a good idea, although a simple "What\'s '
               'New" section for the most recent version\n'
               'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

8.12 reprlib — Implementación repr() alternativa

Código fuente: [Lib/reprlib.py](#)

El módulo `reprlib` provee de los medios necesarios para producir representaciones de objetos con límites en el tamaño de las cadenas resultantes. Es usado en el depurador de Python y puede ser útil también en otros contextos.

Este módulo provee una clase, una instancia y una función:

class `reprlib.Repr`

Clase que provee de servicios de formateo útiles en la implementación de funciones similar a la integrada `repr()`; los límites de tamaño para diferentes tipos de objetos son añadidos para evitar la generación de representaciones que son excesivamente largas.

reprlib.aRepr

Esta es una instancia de `Repr` que es usada para proveer la función `repr()` descrita debajo. Cambiar los atributos de este objeto afectará los límites de tamaño usados por `repr()` y el depurador de Python.

reprlib.repr(obj)

Este es el método `repr()` de `aRepr`. Retorna una cadena similar a la retornada por la función integrada del mismo nombre, pero con límites en la mayoría de tamaños.

Además de las herramientas de limitación de tamaño, el módulo también provee un decorador para detectar invocaciones recursivas a `__repr__()` y sustituyendo por un marcador de posición de cadena en su lugar.

@reprlib.recursive_repr(fillvalue="...")

Decorador para métodos `__repr__()` que detecta invocaciones recursivas dentro del mismo hilo. Si se produce una invocación recursiva, el `fillvalue` es retornado, si no, se produce la invocación `__repr__()` habitual. Por ejemplo:

```
>>> from reprlib import recursive_repr
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

Nuevo en la versión 3.2.

8.12.1 Objetos Repr

Las instancias `Repr` proveen varios atributos que pueden ser usados para proporcionar límites de tamaño para las representaciones de diferentes tipos de objetos, y métodos que formatean tipos de objetos específicos.

Repr.maxlevel

Límite de profundidad en la creación de representaciones recursivas. El valor por defecto es 6.

Repr.maxdict**Repr.maxlist****Repr.maxtuple****Repr.maxset****Repr.maxfrozenset****Repr.maxdeque****Repr.maxarray**

Límites en el número de entradas representadas por el tipo de objeto nombrado. El valor por defecto es 4 para `maxdict`, 5 para `maxarray`, y 6 para los otros.

Repr.maxlong

Máximo número de caracteres en la representación para un entero. Los dígitos son eliminados desde el medio. El valor por defecto es 40.

Repr.maxstring

Límite en el número de caracteres en la representación de la cadena. Fíjese que la representación «normal» de la cadena es la usada como la fuente de caracteres: si se necesitan secuencias de escape en la representación, estas pueden ser desordenadas cuando la representación se ha acortado. El valor por defecto es 30.

Repr.maxother

Este límite es usado para controlar el tamaño de los tipos de objetos para los cuales no hay ningún método de formateo específico en el objeto `Repr`. Se aplica de una manera similar a `maxstring`. El valor por defecto es 20.

`Repr.repr(obj)`

El equivalente a la función integrada `repr()` que usa el formateo impuesto por la instancia.

`Repr.repr1(obj, level)`

Implementación recursiva usada por `repr()`. Este usa el tipo de `obj` para determinar qué método invocar, pasándole `obj` y `level`. Los métodos de tipo específico deben invocar `repr1()` para realizar formateo recursivo, con `level - 1` para el valor de `level` en la invocación recursiva.

`Repr.repr_TYPE(obj, level)`

Métodos de formateo para tipos específicos son implementados como métodos con un nombre basado en el nombre del tipo. En el nombre del método, **TYPE** es reemplazado por `'_'.join(type(obj).__name__.split())`. El envío a estos métodos es gestionado por `repr1()`. Los métodos de tipo específico que necesitan formatear recursivamente un valor deben invocar `self.repr1(subobj, level - 1)`.

8.12.2 Subclasificando Objetos Repr

El uso de envíos dinámicos por `Repr.repr1()` permite a las subclases de `Repr` añadir soporte para tipos adicionales de objetos integrados o modificar el manejo de tipos ya soportados. Este ejemplo muestra como el soporte especial para objetos de tipo archivo puede ser añadido.

```
import reprlib
import sys

class MyRepr(reprlib.Repr):

    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))           # prints '<stdin>'
```

8.13 enum — Soporte para enumeraciones

Nuevo en la versión 3.4.

Código fuente: [Lib/enum.py](#)

Una enumeración es un conjunto de nombres simbólicos (miembros) vinculados a valores únicos y constantes. Dentro de una enumeración, los miembros se pueden comparar por identidad, y la enumeración en sí se puede iterar.

Nota: Case of Enum Members

Because Enums are used to represent constants we recommend using UPPER_CASE names for enum members, and will be using that style in our examples.

8.13.1 Contenido del Módulo

Este módulo define cuatro clases de enumeración que se pueden usar para definir conjuntos únicos de nombres y valores: `Enum`, `IntEnum`, `Flag`, and `IntFlag`. También define un decorador, `unique()`, y un ayudante, `auto`.

class `enum.Enum`

Clase base para crear constantes enumeradas. Consulte la sección [API Funcional](#) para obtener una sintaxis de construcción alternativa.

class `enum.IntEnum`

Clase base para crear constantes enumeradas que también son sub clases de `int`.

class `enum.IntFlag`

Clase base para crear constantes enumeradas que se pueden combinar usando los operadores *bitwise* sin perder su membresía `IntFlag`. Los miembros de `IntFlag` también son subclases de `int`.

class `enum.Flag`

Clase base para crear constantes enumeradas que se pueden combinar utilizando las operaciones *bitwise* sin perder su membresía `Flag`.

`enum.unique()`

El decorador de clase Enum que garantiza que solo un nombre esté vinculado a cualquier valor.

class `enum.auto`

Las instancias se reemplazan con un valor apropiado para los miembros de Enum. El valor inicial comienza en 1.

Nuevo en la versión 3.6: `Flag`, `IntFlag`, `auto`

8.13.2 Creando un Enum

Las enumeraciones son creadas usando la sintaxis `class`, lo que las hace de fácil lectura y escritura. Un método de creación alternativo se describe en [API Funcional](#). Para definir una enumeración, hacer una subclase `Enum` de la siguiente manera:

```
>>> from enum import Enum
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
... 
```

Nota: Valores de miembros de Enum

Los valores de los miembros pueden ser cualquier cosa: `int`, `str`, etc.. Si el valor exacto no es importante, puede usar instancias `auto` y se elegirá un valor apropiado para usted. Se debe tener cuidado si se mezcla `auto` con otros valores.

Nota: Nomenclatura

- La clase `Color` es una *enumeración* (o *enum*)
- Los atributos `Color.RED`, `Color.GREEN`, etc., son *miembros de enumeración* (o *miembros de enum*) y son funcionalmente constantes.
- Los miembros de la enumeración tienen *nombres* y *valores* (el nombre de `Color.RED` es ROJO, el valor de `Color.BLUE` es 3, etc.)

Nota: Aunque usamos la sintaxis `class` para crear Enums, los Enums no son clases normales de Python. Consulte *¿En qué se diferencian las enumeraciones?* para obtener más detalles.

Los miembros de la enumeración tienen representaciones de cadenas legibles para humanos

```
>>> print (Color.RED)
Color.RED
```

...mientras que su `repr` tiene más información

```
>>> print (repr (Color.RED))
<Color.RED: 1>
```

El *tipo* de un miembro de enumeración es la enumeración a la que pertenece:

```
>>> type (Color.RED)
<enum 'Color'>
>>> isinstance (Color.GREEN, Color)
True
>>>
```

Los miembros de Enum también tienen una propiedad que contiene solo su nombre del elemento

```
>>> print (Color.RED.name)
RED
```

Las enumeraciones soportan iteración, en orden de definición:

```
>>> class Shake (Enum):
...     VANILLA = 7
...     CHOCOLATE = 4
...     COOKIES = 9
...     MINT = 3
...
>>> for shake in Shake:
...     print (shake)
...
Shake.VANILLA
Shake.CHOCOLATE
Shake.COOKIES
Shake.MINT
```

Los miembros de la enumeración son hashables, por lo que pueden usarse en diccionarios y conjuntos:

```
>>> apples = {}
>>> apples[Color.RED] = 'red delicious'
>>> apples[Color.GREEN] = 'granny smith'
>>> apples == {Color.RED: 'red delicious', Color.GREEN: 'granny smith'}
True
```

8.13.3 Acceso programático a los miembros de la enumeración y sus atributos

A veces es útil acceder a los miembros en enumeraciones mediante programación (es decir, situaciones en las que `Color.RED` no funcionará porque no se conoce el color exacto al momento de escribir el programa). `Enum` permite dicho acceso:

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

Si desea acceder a los miembros de enumeración por *nombre*, use el acceso a elementos:

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

Si tiene un miembro enum y necesita su `name` o `value`:

```
>>> member = Color.RED
>>> member.name
'RED'
>>> member.value
1
```

8.13.4 Duplicando miembros y valores enum

Tener dos miembros enum con el mismo nombre no es válido:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     SQUARE = 3
...
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'SQUARE'
```

Sin embargo, se permite que dos miembros enum tengan el mismo valor. Dado que dos miembros A y B tienen el mismo valor (y A se definió primero), B es un alias de A. La búsqueda por valor del valor de A y B retornará A. La búsqueda por nombre de B también retornará A:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

Nota: Intentar crear un miembro con el mismo nombre que un atributo ya definido (otro miembro, un método, etc.) o intentar crear un atributo con el mismo nombre que un miembro no está permitido.

8.13.5 Garantizando valores de enumeración únicos

Por defecto, las enumeraciones permiten múltiples nombres como alias para el mismo valor. Cuando no se desea este comportamiento, se puede usar el siguiente decorador para garantizar que cada valor se use solo una vez en la enumeración:

`@enum.unique`

Un decorador de `class` específicamente para enumeraciones. Busca una enumeración `__members__` reuniendo cualquier alias que encuentre; si no se encuentra alguno se genera un `ValueError` con los detalles:

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake': FOUR -> THREE
```

8.13.6 Usando valores automáticos

Si el valor exacto no es importante, puede usar `auto`:

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

Los valores se eligen por `_generate_next_value_()`, que se puede invalidar:

```
>>> class AutoName(Enum):
...     def _generate_next_value_(name, start, count, last_values):
...         return name
...
>>> class Ordinal(AutoName):
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...
>>> list(Ordinal)
[<Ordinal.NORTH: 'NORTH'>, <Ordinal.SOUTH: 'SOUTH'>, <Ordinal.EAST: 'EAST'>,
↪<Ordinal.WEST: 'WEST'>]
```

Nota: The goal of the default `_generate_next_value_()` method is to provide the next `int` in sequence with the last `int` provided, but the way it does this is an implementation detail and may change.

Nota: The `_generate_next_value_()` method must be defined before any members.

8.13.7 Iteración

Iterar sobre los miembros de una enumeración no proporciona los alias:

```
>>> list(Shape)
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
```

El atributo especial `__members__` es una asignación ordenada de solo lectura de nombres a miembros. Incluye todos los nombres definidos en la enumeración, incluidos los alias:

```
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

El atributo `__members__` se puede usar para el acceso programático detallado a los miembros de la enumeración. Por ejemplo, encontrar todos los alias:

```
>>> [name for name, member in Shape.__members__.items() if member.name != name]
['ALIAS_FOR_SQUARE']
```

8.13.8 Comparaciones

Los miembros de la enumeración se comparan por identidad:

```
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

Las comparaciones ordenadas entre valores de enumeración *no* son soportadas. Los miembros de Enum no son enteros (pero vea *IntEnum* a continuación):

```
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Color' and 'Color'
```

Aunque, las comparaciones de igualdad se definen:

```
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

Las comparaciones con valores de no enumeración siempre se compararán no iguales (de nuevo, *IntEnum* fue diseñado explícitamente para comportarse de manera diferente, ver más abajo):

```
>>> Color.BLUE == 2
False
```

8.13.9 Miembros permitidos y atributos de enumeraciones

Los ejemplos anteriores usan números enteros para los valores de enumeración. El uso de enteros es breve y útil (y lo proporciona de forma predeterminada la *API Funcional*), pero no se aplica estrictamente. En la gran mayoría de los casos de uso, a uno no le importa cuál es el valor real de una enumeración. Pero si el valor *es* importante, las enumeraciones pueden tener valores arbitrarios.

Las enumeraciones son clases de Python y pueden tener métodos y métodos especiales como de costumbre. Si tenemos esta enumeración

```
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
...
...     def describe(self):
...         # self is the member here
...         return self.name, self.value
...
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.value)
...
...     @classmethod
...     def favorite_mood(cls):
...         # cls here is the enumeration
...         return cls.HAPPY
...
... 
```

Después:

```
>>> Mood.favorite_mood()
<Mood.HAPPY: 3>
>>> Mood.HAPPY.describe()
('HAPPY', 3)
>>> str(Mood.FUNKY)
'my custom str! 1'
```

Las reglas para lo que está permitido son las siguientes: los nombres que comienzan y terminan con un solo guión bajo están reservados por enum y no se pueden usar; todos los demás atributos definidos dentro de una enumeración se convertirán en miembros de esta enumeración, con la excepción de métodos especiales (`__str__()`, `__add__()`, etc.), descriptores (los métodos también son descriptores) y nombres de variables listado en `_ignore_`.

Nota: si tu enumeración define `__new__()` y/o `__init__()`, los valores que se hayan dado al miembro enum se pasarán a esos métodos. Ver *Planeta* para un ejemplo.

8.13.10 Subclases restringidas de Enum

Una nueva clase *Enum* debe tener una clase base Enum, hasta un tipo de datos concreto, y tantas clases mixin basadas en *object* como sean necesarias. El orden de estas clases base es:

```
class EnumName([mix-in, ...,] [data-type,] base-enum):
    pass
```

Además, la subclasificación de una enumeración solo está permitida si la enumeración no define ningún miembro. Entonces esto está prohibido:

```
>>> class MoreColor(Color):
...     PINK = 17
...
Traceback (most recent call last):
...
TypeError: Cannot extend enumerations
```

Pero esto es permitido:

```
>>> class Foo(Enum):
...     def some_behavior(self):
...         pass
...
>>> class Bar(Foo):
...     HAPPY = 1
...     SAD = 2
... 
```

Permitir la subclasificación de enumeraciones que definen miembros conduciría a una violación de algunos invariantes importantes de tipos e instancias. Por otro lado, tiene sentido permitir compartir un comportamiento común entre un grupo de enumeraciones. (Ver *OrderedEnum* para un ejemplo.)

8.13.11 Serialización

Las enumeraciones se pueden serializar y desempaquetar:

```
>>> from test.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.TOMATO is loads(dumps(Fruit.TOMATO))
True
```

Se aplican las restricciones habituales para la serialización (*pickling*): las enum seleccionables se deben definir en el nivel superior de un módulo, ya que el desempaquetado requiere que sean importables desde ese módulo.

Nota: Con la versión 4 del protocolo de serialización (*pickle*), es posible seleccionar fácilmente las enumeraciones anidadas en otras clases.

Es posible modificar la forma en que los miembros de Enum se serializan/desempaquetan definiendo `__reduce_ex__()` en la clase de enumeración.

8.13.12 API Funcional

La clase *Enum* es invocable, proporcionando la siguiente API funcional:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG')
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
<Animal.ANT: 1>
>>> Animal.ANT.value
1
>>> list(Animal)
[<Animal.ANT: 1>, <Animal.BEE: 2>, <Animal.CAT: 3>, <Animal.DOG: 4>]
```

La semántica de esta API se parece a *namedtuple*. El primer argumento de la llamada a *Enum* es el nombre de la enumeración.

El segundo argumento es la *fente* de los nombres de los miembros de la enumeración. Puede ser una cadena de nombres separados por espacios en blanco, una secuencia de 2 tuplas con pares de clave/valor, o un mapeo de nombres y valores (ej. diccionario). Las últimas dos opciones permiten asignar valores arbitrarios a las enumeraciones; los otros asignan automáticamente enteros crecientes comenzando con 1 (use el parámetro `start` para especificar un valor de inicio diferente). Se regresa una nueva clase derivada de *Enum*. En otras palabras, la asignación de arriba `Animal` es equivalente a:

```
>>> class Animal(Enum):
...     ANT = 1
...     BEE = 2
...     CAT = 3
...     DOG = 4
... 
```

La razón por la que el valor predeterminado es 1 como número inicial y no 0 es que 0 es `False` en sentido booleano, pero todos los miembros `enum` evalúan como `True`.

Las enumeraciones serializadas creadas con la API funcional pueden ser complicadas ya que los detalles de implementación de la pila se usan para tratar de averiguar en qué módulo se está creando la enumeración (ej. fallará si usa una función de utilidad en un módulo separado, y también puede no funcionar en IronPython o Jython). La solución es especificar el nombre del módulo explícitamente de la siguiente manera:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', module=__name__)
```

Advertencia: Si no se suministra un `module`, y `Enum` no puede determinar que es, los miembros del nuevo `Enum` no se podrán desempaquetar; para mantener los errores más cerca de la fuente, la serialización se deshabilitará.

El nuevo protocolo 4 de serialización también, en ciertas circunstancias, se basa en `__qualname__` se establece en la ubicación donde la serialización podrá encontrar la clase. Por ejemplo, si la clase se hizo disponible en la clase `SomeData` en el campo global:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', qualname='SomeData.Animal')
```

La firma completa es:

```
Enum(value='NewEnumName', names=<...>, *, module='...', qualname='...', type=
↳<mixed-in class>, start=1)
```

valor Lo que la nueva clase `Enum` registrará como su nombre.

nombres Los miembros de `Enum`. Esto puede ser un espacio en blanco o una cadena separada por comas (los valores empezarán en 1 a menos que se especifique lo contrario):

```
'RED GREEN BLUE' | 'RED, GREEN, BLUE' | 'RED, GREEN, BLUE'
```

o un iterador de nombres:

```
['RED', 'GREEN', 'BLUE']
```

o un iterador de pares(nombre, valor):

```
[('CYAN', 4), ('MAGENTA', 5), ('YELLOW', 6)]
```

o un mapeo:

```
{'CHARTREUSE': 7, 'SEA_GREEN': 11, 'ROSEMARY': 42}
```

módulo nombre del módulo donde se puede encontrar la nueva clase `Enum`.

qualname donde en el módulo se puede encontrar la nueva clase `Enum`.

tipo escriba para mezclar en la nueva clase `Enum`.

inicio número para comenzar a contar si solo se pasan nombres.

Distinto en la versión 3.5: Se agregó el parámetro `start`.

8.13.13 Enumeraciones derivadas

IntEnum

La primera variación de *Enum* que se proporciona también es una subclase de *int*. Los miembros de *IntEnum* se pueden comparar con enteros; por extensión, las enumeraciones enteras de diferentes tipos también se pueden comparar entre sí:

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Request(IntEnum):
...     POST = 1
...     GET = 2
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

Sin embargo, todavía no se pueden comparar con las enumeraciones estándar *Enum*:

```
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

los valores *IntEnum* se comportan como enteros en otras maneras que esperarías:

```
>>> int(Shape.CIRCLE)
1
>>> ['a', 'b', 'c'][Shape.CIRCLE]
'b'
>>> [i for i in range(Shape.SQUARE)]
[0, 1]
```

IntFlag

La siguiente variación de *Enum* proporcionada, *IntFlag*, también se basa en *int*. La diferencia es que los miembros *IntFlag* se pueden combinar usando los operadores (&, |, ^, ~) y el resultado es un miembro *IntFlag*. Sin embargo, como su nombre lo indica, los miembros de *IntFlag* también son subclase *int* y pueden usarse siempre que *int* se use. Cualquier operación en un miembro *IntFlag* además de las operaciones de bit perderán la membresía *IntFlag*.

Nuevo en la versión 3.6.

Clase muestra *IntFlag*:

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     R = 4
...     W = 2
...     X = 1
...
>>> Perm.R | Perm.W
<Perm.R|W: 6>
>>> Perm.R + Perm.W
6
>>> RW = Perm.R | Perm.W
>>> Perm.R in RW
True
```

También es posible nombrar las combinaciones:

```
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...     RWX = 7
>>> Perm.RWX
<Perm.RWX: 7>
>>> ~Perm.RWX
<Perm.-8: -8>
```

Otra diferencia importante entre *IntFlag* y *Enum* es que si no hay banderas establecidas (el valor es 0), su evaluación booleana es *False*:

```
>>> Perm.R & Perm.X
<Perm.0: 0>
>>> bool(Perm.R & Perm.X)
False
```

Porque los miembros *IntFlag* también son subclases de *int* se pueden combinar con ellos:

```
>>> Perm.X | 8
<Perm.8|X: 9>
```

Bandera

La última variación es *Flag*. Al igual que *IntFlag*, *Flag* los miembros se pueden combinar usando los operadores (&, |, ^, ~). A diferencia de *IntFlag*, no pueden combinar ni comparar con ninguna otra enumeración *Flag*, ni con *int*. Es posible especificar los valores directamente, se recomienda usar *auto* como el valor y dejar que *Flag* seleccione el valor apropiado.

Nuevo en la versión 3.6.

Al igual que *IntFlag*, si una combinación de miembros *Flag* resultan en que no se establezcan banderas, la evaluación booleana es *False*:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.RED & Color.GREEN
<Color.0: 0>
>>> bool(Color.RED & Color.GREEN)
False
```

Las banderas individuales deben tener valores que sean potencias de dos (1, 2, 4, 8, ...), mientras que las combinaciones de banderas no:

```
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
...
>>> Color.WHITE
<Color.WHITE: 7>
```

Dar un nombre a la condición «sin banderas establecidas» no cambia su valor booleano:

```
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.BLACK
<Color.BLACK: 0>
>>> bool(Color.BLACK)
False
```

Nota: Para la mayoría del código nuevo, *Enum* y *Flag* son muy recomendables, ya que *IntEnum* y *IntFlag* rompen algunas promesas semánticas de una enumeración (al ser comparables con enteros, y por transitividad a otras enumeraciones no relacionadas). *IntEnum* y *IntFlag* deben usarse solo en casos donde *Enum* y *Flag* no son suficientes: por ejemplo, cuando las constantes enteras se reemplazan por enumeraciones, o por interoperabilidad con otros sistemas.

Otros

Mientras que *IntEnum* es parte del módulo *enum*, sería muy simple de implementar de forma independiente:

```
class IntEnum(int, Enum):
    pass
```

Esto demuestra que similares pueden ser las enumeraciones derivadas; por ejemplo una *StrEnum* que se mezcla en *str* en lugar de *int*.

Algunas reglas:

1. Al subclasificar *Enum*, los tipos mixtos deben aparecer antes *Enum* en la secuencia de bases, como en el ejemplo anterior *IntEnum*.
2. Mientras que *Enum* puede tener miembros de cualquier tipo, una vez que se mezcle tipos adicionales, todos los miembros deben de tener los valores de ese tipo, ej. *int* de arriba. Esta restricción no se aplica a las mezclas que solo agregan métodos y no especifican otro tipo de datos como *int* o *str*.
3. Cuando se mezcla otro tipo de datos, el atributo *value* *no es el mismo* que el mismo miembro enum, aunque es equivalente y se comparará igual.
4. Formato %-style: *%s* y *%r* llaman, respectivamente, a *__str__()* y *__repr__()* de la clase *Enum*; otros códigos (como *&i* o *%h* para *IntEnum*) tratan al miembro enum como su tipo mixto.
5. Formatted string literals, *str.format()*, and *format()* will use the mixed-in type's *__format__()* unless *__str__()* or *__format__()* is overridden in the subclass, in which case the overridden methods or *Enum* methods will be used. Use the *!s* and *!r* format codes to force usage of the *Enum* class's *__str__()* and *__repr__()* methods.

8.13.14 Cuándo usar `__new__()` contra `__init__()`

`__new__()` debe usarse siempre que desee personalizar el valor del miembro real `Enum`. Cualquier otra modificación puede ir en `__new__()` o `__init__()`, prefiriendo siempre `__init__()`.

Por ejemplo, si desea pasar varios elementos al constructor, pero solo desea que uno de ellos sea el valor:

```
>>> class Coordinate(bytes, Enum):
...     """
...     Coordinate with binary codes that can be indexed by the int code.
...     """
...     def __new__(cls, value, label, unit):
...         obj = bytes.__new__(cls, [value])
...         obj._value_ = value
...         obj.label = label
...         obj.unit = unit
...         return obj
...     PX = (0, 'P.X', 'km')
...     PY = (1, 'P.Y', 'km')
...     VX = (2, 'V.X', 'km/s')
...     VY = (3, 'V.Y', 'km/s')
...

>>> print(Coordinate['PY'])
Coordinate.PY

>>> print(Coordinate(3))
Coordinate.VY
```

8.13.15 Ejemplos interesantes

Si bien se espera que `Enum`, `IntEnum`, `IntFlag`, y `Flag` cubran la mayoría de los casos de uso, no pueden cubrirlos a todos. Aquí hay recetas para algunos tipos diferentes de enumeraciones que puede usarse directamente, o como ejemplos para crear los propios.

Omitir valores

En muchos casos de uso, a uno no le importa cuál es el valor real de una enumeración. Hay varias formas de definir este tipo de enumeración simple:

- use instancias de `auto` para el valor
- use instancias de `object` como el valor
- use a descriptive string as the value
- use una tupla como valor y un `__new__()` personalizado para reemplazar la tupla con un valor `int`

El uso de cualquiera de estos métodos significa para el usuario que estos valores no son importantes y también permite agregar, eliminar o reordenar miembros sin tener que volver a numerar los miembros restantes.

Cualquiera que sea el método que elijas, debe proporcionar un `repr()` que también oculte el valor (sin importancia):

```
>>> class NoValue(Enum):
...     def __repr__(self):
...         return '<%s.%s>' % (self.__class__.__name__, self.name)
... 
```

Usando `auto`

Usando `auto` se vería como:

```
>>> class Color(NoValue):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.GREEN
<Color.GREEN>
```

Usando `object`

Usando `object` se vería como:

```
>>> class Color(NoValue):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...
>>> Color.GREEN
<Color.GREEN>
```

Usando una cadena descriptiva

Usar una cadena como valor se vería así:

```
>>> class Color(NoValue):
...     RED = 'stop'
...     GREEN = 'go'
...     BLUE = 'too fast!'
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
'go'
```

Usando `__new__()` personalizados

Usando una numeración automática `__new__()` se vería como:

```
>>> class AutoNumber(NoValue):
...     def __new__(cls):
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
>>> class Color(AutoNumber):
...     RED = ()
...     GREEN = ()
...     BLUE = ()
...
>>> Color.GREEN
<Color.GREEN>
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> Color.GREEN.value
2
```

To make a more general purpose `AutoNumber`, add `*args` to the signature:

```
>>> class AutoNumber(NoValue):
...     def __new__(cls, *args):          # this is the only change from above
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
... 
```

Then when you inherit from `AutoNumber` you can write your own `__init__` to handle any extra arguments:

```
>>> class Swatch(AutoNumber):
...     def __init__(self, pantone='unknown'):
...         self.pantone = pantone
...     AUBURN = '3497'
...     SEA_GREEN = '1246'
...     BLEACHED_CORAL = () # New color, no Pantone code yet!
... 
```

```
>>> Swatch.SEA_GREEN
<Swatch.SEA_GREEN: 2>
>>> Swatch.SEA_GREEN.pantone
'1246'
>>> Swatch.BLEACHED_CORAL.pantone
'unknown'
```

Nota: El método `__new__()`, está definido, se usa durante la creación de los miembros `Enum`; se remplaza entonces por el `Enum __new__()` que se utiliza después de la creación de la clase para buscar miembros existentes.

OrderedEnum

Una enumeración ordenada que no se basa en `IntEnum` y, por lo tanto mantiene los invariantes normales de `Enum` (como no ser comparables con otras enumeraciones):

```
>>> class OrderedEnum(Enum):
...     def __ge__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value >= other.value
...         return NotImplemented
...     def __gt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value > other.value
...         return NotImplemented
...     def __le__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value <= other.value
...         return NotImplemented
...     def __lt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value < other.value
...         return NotImplemented
... 
```

```
>>> class Grade(OrderedEnum):
...     A = 5
```

(continué en la próxima página)

(proviene de la página anterior)

```

...     B = 4
...     C = 3
...     D = 2
...     F = 1
...
>>> Grade.C < Grade.A
True

```

DuplicateFreeEnum

Levanta un error si se encuentra un nombre de miembro duplicado en lugar de crear un alias:

```

>>> class DuplicateFreeEnum(Enum):
...     def __init__(self, *args):
...         cls = self.__class__
...         if any(self.value == e.value for e in cls):
...             a = self.name
...             e = cls(self.value).name
...             raise ValueError(
...                 "aliases not allowed in DuplicateFreeEnum: %r --> %r"
...                 % (a, e))
...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     GRENE = 2
...
Traceback (most recent call last):
...
ValueError: aliases not allowed in DuplicateFreeEnum: 'GRENE' --> 'GREEN'

```

Nota: Este es un ejemplo útil para subclasificar Enum para agregar o cambiar otros comportamientos, así como no permitir alias. Si el único cambio deseado es no permitir alias, el decorador `unique()` puede usarse en su lugar.

Planeta

Si `__new__()` o `__init__()` se definen el valor del miembro enum se pasará a estos métodos:

```

>>> class Planet(Enum):
...     MERCURY = (3.303e+23, 2.4397e6)
...     VENUS = (4.869e+24, 6.0518e6)
...     EARTH = (5.976e+24, 6.37814e6)
...     MARS = (6.421e+23, 3.3972e6)
...     JUPITER = (1.9e+27, 7.1492e7)
...     SATURN = (5.688e+26, 6.0268e7)
...     URANUS = (8.686e+25, 2.5559e7)
...     NEPTUNE = (1.024e+26, 2.4746e7)
...     def __init__(self, mass, radius):
...         self.mass = mass # in kilograms
...         self.radius = radius # in meters
...     @property
...     def surface_gravity(self):
...         # universal gravitational constant (m3 kg-1 s-2)
...         G = 6.67300E-11
...         return G * self.mass / (self.radius * self.radius)

```

(continué en la próxima página)

(proviene de la página anterior)

```
...
>>> Planet.EARTH.value
(5.976e+24, 6378140.0)
>>> Planet.EARTH.surface_gravity
9.802652743337129
```

Periodo de tiempo

Un ejemplo para mostrar el atributo `ignore` en uso:

```
>>> from datetime import timedelta
>>> class Period(timedelta, Enum):
...     "different lengths of time"
...     _ignore_ = 'Period i'
...     Period = vars()
...     for i in range(367):
...         Period['day_%d' % i] = i
...
>>> list(Period)[:2]
[<Period.day_0: datetime.timedelta(0)>, <Period.day_1: datetime.timedelta(days=1)>]
>>> list(Period)[-2:]
[<Period.day_365: datetime.timedelta(days=365)>, <Period.day_366: datetime.
↳timedelta(days=366)>]
```

8.13.16 ¿Cómo son diferentes las Enums?

Los Enums tienen una metaclasses personalizada que afecta muchos aspectos, tanto de las clases derivadas `Enum` como de sus instancias (miembros).

Clases Enum

La meta clase `EnumMeta` es responsable de proveer los métodos `__contains__()`, `__dir__()`, `__iter__()` y cualquier otro que permita hacer cosas con una clase `Enum` que falla en una clase típica, como `list(Color)` o `some_enum_var in Color`. `EnumMeta` es responsable de asegurar que los otros varios métodos en la clase final `Enum` sean correctos (como `__new__()`, `__getnewargs__()`, `__str__()` y `__repr__()`).

Miembros de Enum (también conocidos como instancias)

Lo más interesante de los miembros de `Enum` es que son únicos. `Enum` los crea todos mientras está creando la clase `Enum` misma, y después un `__new__()` personalizado para garantizar que nunca se creen instancias nuevas retornando solo las instancias de miembros existentes.

Puntos más finos

Nombres soportados `__dunder__`

`__members__` es una asignación ordenada de solo lectura de artículos `member_name:member`. Solo está disponible en la clase.

`__new__()`, si se especifica, debe crear y retornar los miembros de enumeración; también es una muy buena idea establecer el `_value_` del miembro apropiadamente. Una vez que se crean todos los miembros, ya no se usa.

Nombres `_sunder_` compatibles

- `_name_` — nombre del miembro
- `_value_` — valor del miembro; se puede definir / modificar en `__new__`
- `_missing_` — una función de búsqueda utilizada cuando no se encuentra un valor; puede ser anulado
- `_ignore_` — una lista de nombres, ya sea como una `list()` o una `str()` que no será transformada en miembros, y que se eliminará de la clase final
- `_order_` — usado en código Python 2/3 para asegurar que el orden de los miembros sea consistente (atributo de clase, eliminado durante la creación de la clase)
- `_generate_next_value_` — usado por la *Funcional API* y por *auto* para obtener un valor apropiado para un miembro enum; puede ser anulado

Nuevo en la versión 3.6: `_missing_`, `_order_`, `_generate_next_value_`

Nuevo en la versión 3.7: `_ignore_`

Para ayudar a mantener sincronizado el código Python 2 / Python 3 se puede proporcionar un atributo `_order_`. Se verificará con el orden real de la enumeración y generará un error si los dos no coinciden:

```
>>> class Color(Enum):
...     _order_ = 'RED GREEN BLUE'
...     RED = 1
...     BLUE = 3
...     GREEN = 2
...
Traceback (most recent call last):
...
TypeError: member order does not match _order_
```

Nota: En código Python 2 el atributo `_order_` es necesario ya que el orden de definición se pierde antes de que se pueda registrar.

Tipo de miembro Enum

Los miembros *Enum* son instancias de su clase *Enum*, y normalmente se accede a ellos como `EnumClass.member`. Bajo ciertas circunstancias también se puede acceder como `EnumClass.member.member`, pero nunca se debe hacer esto ya que esa búsqueda puede fallar, o peor aún, retornar algo además del miembro *Enum* que está buscando (esta es otra buena razón para usar solo mayúsculas en los nombres para los miembros):

```
>>> class FieldTypes(Enum):
...     name = 0
...     value = 1
...     size = 2
...
>>> FieldTypes.value.size
<FieldTypes.size: 2>
>>> FieldTypes.size.value
2
```

Distinto en la versión 3.5.

Valor booleano de las clases y miembros `Enum`

Los miembros `Enum` que están mezclados con tipos sin-`Enum` (como `int`, `str`, etc.) se evalúan de acuerdo con las reglas de tipo mixto; de lo contrario, todos los miembros evalúan como `True`. Para hacer que tu propia evaluación booleana de `Enum` dependa del valor del miembro, agregue lo siguiente a su clase:

```
def __bool__(self):
    return bool(self.value)
```

las clases `Enum` siempre evalúan como `True`.

`Enum` clases con métodos

Si le da a su subclase `Enum` métodos adicionales, como la clase `Planet` anterior, esos métodos aparecerán en una `dir()` del miembro, pero no de la clase

```
>>> dir(Planet)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN', 'URANUS', 'VENUS', '__
↳class__', '__doc__', '__members__', '__module__']
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'name', 'surface_gravity', 'value']
```

Combinando miembros de “Flag”

Si no se nombra una combinación de miembros de `Flag`, el `repr()` incluirá todos los flags con nombre y todas las combinaciones de flags con nombre que estén en el valor

```
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     MAGENTA = RED | BLUE
...     YELLOW = RED | GREEN
...     CYAN = GREEN | BLUE
...
>>> Color(3) # named combination
<Color.YELLOW: 3>
>>> Color(7) # not named combination
<Color.CYAN|MAGENTA|BLUE|YELLOW|GREEN|RED: 7>
```

Módulos numéricos y matemáticos

Los módulos descritos en este capítulo proporcionan funciones y tipos de datos numéricos y relacionados con las matemáticas. El módulo `numbers` define una jerarquía abstracta de tipos numéricos. Los módulos `math` y `cmath` contienen varias funciones matemáticas para números complejos y de punto flotante. El módulo `decimal` admite representaciones exactas de números decimales, utilizando aritmética de precisión arbitraria.

En este capítulo se documentan los siguientes módulos:

9.1 `numbers` — Clase base abstracta numérica

Código fuente: [Lib/numbers.py](#)

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`

La raíz de la jerarquía numérica. Si desea validar si un argumento `x` es un número, sin importar su tipo, use `isinstance(x, Number)`.

9.1.1 The numeric tower

class `numbers.Complex`

Las subclases de este tipo describen números complejos e incluyen las operaciones integradas del tipo `complex`. Estas son: conversiones a `complex` y `bool`, `real`, `imag`, `+`, `-`, `*`, `/`, `abs()`, `conjugate()`, `==`, y `!=`. Todos excepto `-` y `!=` estos son abstractos.

real

Abstracto. Recupera el componente real de este número.

imag

Abstracto. Recupera el componente imaginario de este número.

abstractmethod `conjugate()`

Abstracto. Retorna el complejo conjugado. Por ejemplo, `(1+3j).conjugate() == (1-3j)`.

class numbers.Real

Para *Complex*, *Real* agrega las operaciones que trabajan con números reales.

En resumen, estos son: conversiones a *float*, *math.trunc()*, *round()*, *math.floor()*, *math.ceil()*, *divmod()*, *//*, *%*, *<*, *<=*, *>*, *y >=*.

Real también proporciona valores predeterminados para *complex()*, *real*, *imag*, y *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

Abstracto.

denominator

Abstracto.

class numbers.Integral

Hereda *Rational* y agrega conversión a *int*. Proporciona valores predeterminados para *float()*, *numerator*, y *denominator*. Agrega métodos abstractos para operaciones **** y operaciones de cadena de bits: *<<*, *>>*, *&*, *^*, *|*, *~*.

9.1.2 Notas para implementadores de tipos

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))
```

Agregar más *ABCs* numéricos

Por supuesto, hay más *ABCs* posibles para los números, y esto sería una jerarquía deficiente si se excluye la posibilidad de añadirlos. Puede usar `MyFoo` entre `Complex` y `Real` así:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

Implementar operaciones aritméticas

Queremos implementar las operaciones aritméticas tal que las operaciones de modo mixto llamen a una implementación cuyo autor conocía los tipos de ambos argumentos, o convertir ambos argumentos al tipo incorporado más cercano antes de hacer la operación. Para subtipos de `Integral`, esto significa que `__add__()` y `__radd__()` tienen que ser definidos como:

```
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)
        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
```

Hay 5 casos diferentes para una operación de tipo mixto en subclases de `Complex`. Se explicará todo el código anterior que no se refiere a `MyIntegral` y `OtherTypeIKnowAbout` como "repetitivo". ``a` será una instancia de A, que es un subtipo de `Complex` (a: A <: Complex), y ``b` : B <: Complex. Consideraré a + b:

1. Si A define un `__add__()` que acepta b, todo está bien.
2. Si A recurre al código repetitivo y retorna un valor de `__add__()`, perderíamos la posibilidad de que B defina un `__radd__()` más inteligente, por lo que el código repetitivo debería retornar `NotImplemented` de `__add__()`. (O A no puede implementar `__add__()` en absoluto.)
3. Entonces B's `__radd__()` tiene una oportunidad. Si acepta a, todo está bien.
4. Si se vuelve a caer en el código repetitivo, no hay más posibles métodos para probar, por lo que acá debería vivir la implementación predeterminada.
5. Si B <: A, Python probará B.`__radd__` antes que A.`__add__`. Esto está bien, porque se implementó con conocimiento de A, por lo que puede manejar instancias antes de delegar un `Complex`.

Si A <: `Complex` y B <: `Real` sin compartir ningún otro conocimiento, la operación compartida apropiada es la que involucra la clase `complex` incorporada, y ambos `__radd__()` desencadenan allí, entonces `a+b == b+a`.

Dado que la mayoría de las operaciones en un tipo determinado serán muy similares, puede ser útil definir una función auxiliar que genere las instancias *forward* y *reverse* de cualquier operador dado. Por ejemplo, *fractions.Fraction* así:

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, 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.
            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 — Funciones matemáticas

Este módulo proporciona acceso a las funciones matemáticas definidas en el estándar de C.

Estas funciones no pueden ser usadas con números complejos; usa las funciones con el mismo nombre del módulo *cmath* si requieres soporte para números complejos. La distinción entre las funciones que admiten números complejos y las que no se hace debido a que la mayoría de los usuarios no quieren aprender tantas matemáticas como se requiere para comprender los números complejos. Recibir una excepción en lugar de un resultado complejo permite la detección temprana del número complejo inesperado utilizado como parámetro, de modo que el programador pueda determinar cómo y por qué se generó en primer lugar.

Este módulo proporciona las funciones descritas a continuación. Excepto cuando se indique lo contrario explícitamente, todos los valores retornados son flotantes.

9.2.1 Teoría de números y funciones de representación

`math.ceil(x)`

Retorna el «techo» de x , el número entero más pequeño que es mayor o igual que x . Si x no es un flotante, delega en `x.__ceil__()`, que debería retornar un valor *Integral*.

`math.comb(n, k)`

Retorna el número de formas posibles de elegir k elementos de n , de forma ordenada y sin repetición.

Se evalúa como $n! / (k! * (n - k)!)$ cuando $k \leq n$ y como cero cuando $k > n$.

También se llama coeficiente binomial porque es equivalente al coeficiente del k -ésimo término en el desarrollo polinomial de la expresión $(1 + x)^n$.

Lanza una excepción *TypeError* si alguno de los argumentos no es un entero. Lanza una excepción *ValueError* si alguno de los argumentos es negativo.

Nuevo en la versión 3.8.

`math.copysign(x, y)`

Retorna un flotante con la magnitud (valor absoluto) de x pero el signo de y . En plataformas que admiten ceros con signo, `copysign(1.0, -0.0)` retorna `-1.0`.

`math.fabs(x)`

Retorna el valor absoluto de x .

`math.factorial(x)`

Retorna el factorial de x como un número entero. Lanza una excepción *ValueError* si x no es un entero o es negativo.

`math.floor(x)`

Retorna el «suelo» de x , el primer número entero mayor o igual que x . Si x no es un flotante, delega en `x.__floor__()`, que debería retornar un valor *Integral*.

`math.fmod(x, y)`

Retorna `fmod(x, y)`, tal como se define en la biblioteca de C de la plataforma. Ten en cuenta que la expresión `x % y` de Python puede no retornar el mismo resultado. La intención del estándar de C es que `fmod(x, y)` sea exactamente (matemáticamente; con precisión infinita) igual a $x - n*y$ para algún número entero n tal que el resultado tenga el mismo signo que x y magnitud menor que `abs(y)`. La expresión `x % y` de Python retorna un resultado con el signo de y en su lugar, y es posible que no pueda calcularse con exactitud para argumentos flotantes. Por ejemplo, `fmod(-1e-100, 1e100)` es `-1e-100`, pero el resultado de `-1e-100 % 1e100` en Python es `1e100-1e-100`, que no se puede representar exactamente como un flotante, y se redondea sorprendentemente a `1e100`. Por esta razón, generalmente se prefiere la función `fmod()` cuando se trabaja con flotantes, mientras que se prefiere el uso de `x % y` de Python cuando se trabaja con enteros.

`math.frexp(x)`

Retorna la mantisa y el exponente de x como el par (m, e) . m es un flotante y e es un número entero tal que $x == m * 2**e$ exactamente. Si x es cero, retorna $(0.0, 0)$, y retorna $0.5 \leq \text{abs}(m) < 1$ en caso contrario. Se utiliza como una forma portable de «extraer» la representación interna de un flotante.

`math.fsum(iterable)`

Retorna una suma precisa en coma flotante de los valores de un iterable. Evita la pérdida de precisión mediante el seguimiento de múltiples sumas parciales intermedias:

```
>>> 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
```

La precisión del algoritmo depende de las garantías aritméticas de IEEE-754 y del caso típico en el que se usa el «medio redondo a par» (half-even) como método de redondeo. En algunas compilaciones que no son de Windows, la biblioteca de C subyacente utiliza la adición de precisión extendida y, ocasionalmente, puede realizar un doble redondeo en una suma intermedia, haciendo que el bit menos significativo tome el valor incorrecto.

Para una discusión más amplia y dos enfoques alternativos, consultar [ASPN cookbook recipes for accurate floating point summation](#).

`math.gcd(a, b)`

Retorna el máximo común divisor de los números enteros a y b . Si a o b son distintos de cero, el valor de `gcd(a, b)` es el mayor entero positivo que divide a ambos, a y b . `gcd(0, 0)` retorna 0.

Nuevo en la versión 3.5.

`math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)`

Retorna `True` si los valores a y b están cerca el uno del otro y `False` en caso contrario.

Que dos valores se consideren cercanos o no, se determina de acuerdo con las tolerancias absolutas y relativas dadas.

rel_tol es la tolerancia relativa: esta es la diferencia máxima permitida entre a y b , en relación con el valor absoluto mayor de a o b . Por ejemplo, para establecer una tolerancia del 5%, pasa `rel_tol=0.05`. La tolerancia predeterminada es $1e-09$, lo que garantiza que los dos valores sean iguales considerando 9 dígitos decimales aproximadamente. *rel_tol* debe ser mayor que cero.

abs_tol es la tolerancia absoluta mínima, útil para las comparaciones cercanas a cero. *abs_tol* debe valer al menos cero.

Si no se encuentran errores, el resultado será: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

Los valores especiales de IEEE 754 NaN, `inf` e `-inf` se manejarán de acuerdo con las reglas del IEEE. Concretamente, NaN no se considera cercano a ningún otro valor, incluido NaN. Por su parte, `inf` e `-inf` solo se consideran cercanos a sí mismos.

Nuevo en la versión 3.5.

Ver también:

PEP 485 – Una función para comprobar la igualdad aproximada

`math.isfinite(x)`

Retorna `True` si x no es infinito ni NaN, o `False` en caso contrario. (Ten en cuenta que `0.0` es considerado finito.)

Nuevo en la versión 3.2.

`math.isinf(x)`

Retorna `True` si x es infinito positivo o negativo, o `False` en caso contrario.

`math.isnan(x)`

Retorna `True` si x es NaN (not a number, en español: no es un número), o `False` en caso contrario.

`math.isqrt(n)`

Retorna la raíz cuadrada del número entero no negativo n . Es el resultado de aplicar la función suelo al valor exacto de la raíz cuadrada de n , o de forma equivalente, el mayor entero a tal que $a^2 \leq n$.

Para algunas aplicaciones, puede ser más conveniente tener el menor número entero a tal que $n \leq a^2$, en otras palabras, el resultado de aplicar la función techo a la raíz cuadrada exacta de n . Para n positivo, esto se puede calcular usando `a = 1 + isqrt(n - 1)`.

Nuevo en la versión 3.8.

`math.ldexp(x, i)`

Retorna `x * (2**i)`. Esta es esencialmente la función inversa de `frexp()`.

`math.modf(x)`

Retorna la parte fraccionaria y entera de x . Ambos resultados son flotantes y tienen el mismo signo que x .

`math.perm(n, k=None)`

Retorna el número de formas posibles de elegir k elementos de n elementos, sin repetición y en orden.

Se evalúa como $n! / (n - k)!$ cuando $k \leq n$ y como cero cuando $k > n$.

Si k no se especifica o es `None`, k será igual a n por defecto y la función retornará $n!$.

Lanza una excepción `TypeError` si alguno de los argumentos no es un entero. Lanza una excepción `ValueError` si alguno de los argumentos es negativo.

Nuevo en la versión 3.8.

`math.prod(iterable, *, start=1)`

Calcula el producto de todos los elementos en la entrada *iterable*. El valor *start* predeterminado para el producto es 1.

Cuando el iterable está vacío, retorna el valor inicial. Esta función está diseñada específicamente para su uso con valores numéricos y puede rechazar tipos no numéricos.

Nuevo en la versión 3.8.

`math.remainder(x, y)`

Retorna el resto o residuo según la norma IEEE 754 de x con respecto a y . Para un valor x finito y un valor y finito distinto de cero, es la diferencia $x - n * y$, donde n es el número entero más cercano al valor exacto del cociente x / y . Si x / y está exactamente en mitad de dos enteros consecutivos, el entero *par* más cercano se utiliza para n . Por lo tanto, el residuo $r = \text{remainder}(x, y)$ siempre satisface $\text{abs}(r) \leq 0.5 * \text{abs}(y)$.

Los casos especiales siguen el estándar IEEE 754: en particular, `remainder(x, math.inf)` es x para todo x finito, y `remainder(x, 0)` junto a `remainder(math.inf, x)` lanzan una excepción `ValueError` para todo x que no sea NaN. Si el resultado de la operación residuo es cero, este cero tendrá el mismo signo que x .

En plataformas que utilizan la norma IEEE 754 para números en coma flotante binarios, el resultado de esta operación siempre es exactamente representable: no se introduce ningún error de redondeo.

Nuevo en la versión 3.7.

`math.trunc(x)`

Retorna el valor *Real* x truncado a un *Integral* (generalmente un entero). Delega en `x.__trunc__()`.

Ten en cuenta que `frexp()` y `modf()` tienen un patrón de llamada/retorno diferente al de sus equivalentes en C: toman un solo argumento y retornan un par de valores, en lugar de retornar su segundo valor de retorno a través de un *parámetro de salida* (no existe tal cosa en Python).

Para las funciones `ceil()`, `floor()` y `modf()`, ten en cuenta que *todos* los números de coma flotante de magnitud suficientemente grande son enteros exactos. Los flotantes de Python normalmente no tienen más de 53 bits de precisión (lo mismo que el tipo `double` de C en la plataforma), en cuyo caso cualquier flotante x con $\text{abs}(x) \geq 2^{52}$ no necesariamente tiene bits fraccionarios.

9.2.2 Funciones logarítmicas y exponenciales

`math.exp(x)`

Retorna e elevado a la x potencia, dónde $e = 2.718281\dots$ es la base de los logaritmos naturales. Esto generalmente es más preciso que `math.e ** x` o `pow(math.e, x)`.

`math.expm1(x)`

Retorna e elevado a la x potencia, menos 1. Aquí e es la base de los logaritmos naturales. Para flotantes x pequeños, la resta en `exp(x) - 1` puede resultar en una *pérdida significativa de precisión*; la función `expm1()` proporciona una forma de calcular este valor con una precisión total:

```
>>> 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
```

Nuevo en la versión 3.2.

`math.log(x[, base])`

Con un argumento, retorna el logaritmo natural de x (en base e).

Con dos argumentos, retorna el logaritmo de x en la *base* dada, calculado como $\log(x) / \log(\text{base})$.

`math.log1p(x)`

Retorna el logaritmo natural de $1+x$ (base e). El resultado se calcula de forma precisa para x cercano a cero.

`math.log2(x)`

Retorna el logaritmo en base 2 de x . Esto suele ser más preciso que `log(x, 2)`.

Nuevo en la versión 3.3.

Ver también:

`int.bit_length()` retorna el número de bits necesarios para representar un entero en binario, excluyendo el signo y los ceros iniciales.

`math.log10(x)`

Retorna el logaritmo en base 10 de x . Esto suele ser más preciso que `log(x, 10)`.

`math.pow(x, y)`

Retorna x elevado a la potencia y . Los casos excepcionales siguen el Anexo “F” del estándar C99 en la medida de lo posible. En particular, `pow(1.0, x)` y `pow(x, 0.0)` siempre retornan `1.0`, incluso cuando x es cero o NaN. Si tanto x como y son finitos, x es negativo e y no es un número entero, entonces `pow(x, y)` no está definido y se lanza una excepción `ValueError`.

A diferencia del operador incorporado `**`, `math.pow()` convierte ambos argumentos al tipo `float`. Utiliza `**` o la función incorporada `pow()` para calcular potencias enteras exactas.

`math.sqrt(x)`

Retorna la raíz cuadrada de x .

9.2.3 Funciones trigonométricas

`math.acos(x)`

Retorna el arcocoseno de x , en radianes.

`math.asin(x)`

Retorna el arcoseno de x , en radianes.

`math.atan(x)`

Retorna la arcotangente de x , en radianes.

`math.atan2(y, x)`

Retorna $\text{atan}(y / x)$, en radianes. El resultado está entre $-\pi$ y π . El vector del plano que va del origen al punto (x, y) , forma este ángulo con el eje X positivo. La ventaja de `atan2()` es que el signo de ambas entradas es conocido, por lo que se puede calcular el cuadrante correcto para el ángulo. Por ejemplo, `atan(1)` y `atan2(1, 1)` son ambas $\pi/4$, pero `atan2(-1, -1)` es $-3\pi/4$.

`math.cos(x)`

Retorna el coseno de x radianes.

`math.dist(p, q)`

Retorna la distancia euclidiana entre dos puntos p y q , cada uno de ellos dado como una secuencia (o iterable) de coordenadas. Los dos puntos deben tener la misma dimensión.

Aproximadamente equivalente a:

```
sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))
```

Nuevo en la versión 3.8.

`math.hypot(*coordinates)`

Retorna la norma euclidiana, `sqrt(sum(x**2 for x in coordinates))`. Esta es la longitud del vector que va desde el origen hasta el punto dado por las coordenadas.

Para un punto bidimensional (x, y) , esto equivale a calcular la hipotenusa de un triángulo rectángulo usando el teorema de Pitágoras, `sqrt(x*x + y*y)`.

Distinto en la versión 3.8: Agregado soporte para puntos n-dimensionales. Anteriormente, solo se admitía el caso bidimensional.

`math.sin(x)`
Retorna el seno de x radianes.

`math.tan(x)`
Retorna la tangente de x radianes.

9.2.4 Conversión angular

`math.degrees(x)`
Convierte el ángulo x de radianes a grados.

`math.radians(x)`
Convierte el ángulo x de grados a radianes.

9.2.5 Funciones hiperbólicas

Las funciones hiperbólicas son análogas a las funciones trigonométricas pero basadas en hipérbolas en lugar de en círculos.

`math.acosh(x)`
Retorna el coseno hiperbólico inverso de x .

`math.asinh(x)`
Retorna el seno hiperbólico inverso de x .

`math.atanh(x)`
Retorna la tangente hiperbólica inversa de x .

`math.cosh(x)`
Retorna el coseno hiperbólico de x .

`math.sinh(x)`
Retorna el seno hiperbólico de x .

`math.tanh(x)`
Retorna la tangente hiperbólica de x .

9.2.6 Funciones especiales

`math.erf(x)`
Retorna la función error en x .

La función `erf()` se puede utilizar para calcular funciones estadísticas tradicionales como la distribución normal estándar acumulativa:

```
def phi(x):
    'Cumulative distribution function for the standard normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

Nuevo en la versión 3.2.

`math.erfc(x)`
Retorna la función error complementaria en x . La función error complementaria se define como $1.0 - \text{erf}(x)$. Se usa para valores grandes de x donde una resta de 1 causaría una pérdida de precisión.

Nuevo en la versión 3.2.

`math.gamma(x)`

Retorna la función `gamma` en x .

Nuevo en la versión 3.2.

`math.lgamma(x)`

Retorna el logaritmo natural del valor absoluto de la función `gamma` en x .

Nuevo en la versión 3.2.

9.2.7 Constantes

`math.pi`

La constante matemática $\pi = 3.141592\dots$, hasta la precisión disponible.

`math.e`

La constante matemática $e = 2.718281\dots$, hasta la precisión disponible.

`math.tau`

La constante matemática $\tau = 6.283185\dots$, hasta la precisión disponible. Tau es una constante del círculo igual a 2π , la razón entre la circunferencia de un círculo y su radio. Para obtener más información sobre Tau, consulta el video de Vi Hart, [Pi is \(still\) Wrong](#), y comienza a celebrar el [el día de Tau](#) ¡comiendo el doble de tarta!

Nuevo en la versión 3.6.

`math.inf`

Un valor infinito positivo en punto flotante. (Para un valor infinito negativo, usa `-math.inf`.) Equivalente a la salida de `float('inf')`.

Nuevo en la versión 3.5.

`math.nan`

Un valor de punto flotante que «no es un número» (NaN). Equivalente a la salida de `float('nan')`.

Nuevo en la versión 3.5.

CPython implementation detail: El módulo `math` consiste principalmente en delgados envoltorios alrededor de las funciones matemáticas de la biblioteca de C de la plataforma. El comportamiento en casos excepcionales sigue el Anexo F del estándar C99 cuando corresponda. La implementación actual lanzará un `ValueError` para operaciones no válidas como `sqrt(-1.0)` o `log(0.0)` (donde el estándar C99 recomienda señalar que la operación no es válida o que hay división entre cero), y un `OverflowError` para aquellos resultados de desbordamiento (por ejemplo, `exp(1000.0)`). No se retornará NaN para ninguna de las funciones anteriores, a no ser que al menos uno de los argumentos de la función sea NaN. En este caso, la mayoría de las funciones retornan NaN, pero de nuevo (de acuerdo con el apéndice F del estándar C99) hay algunas excepciones a esta regla, por ejemplo `pow(float('nan'), 0.0)` o `hypot(float('nan'), float('inf'))`.

Ten en cuenta que Python no hace ningún esfuerzo por distinguir los NaN de señalización de los NaN silenciosos, y el comportamiento de señalización de los NaN permanece sin especificar. El comportamiento estándar es tratar a todos los NaN como silenciosos.

Ver también:

Módulo `cmath` Versiones de muchas de estas funciones para números complejos.

9.3 cmath – Función matemática para números complejos

Este módulo proporciona acceso a funciones matemáticas para números complejos. Las funciones de este módulo aceptan números enteros, números de coma flotante o números complejos como argumentos. Aceptarán además cualquier objeto de Python que tenga tanto un método `__complex__()` o un método `__float__()`: estos métodos son usados para convertir el objeto a un número complejo o de coma flotante, respectivamente, y la función es entonces aplicada al resultado de la conversión.

Nota: En sistemas con hardware y soporte del sistema para ceros con signo, las funciones que involucran tramos son continuas en *ambos* lados del tramo: el signo de cero distingue un lado del tramo del otro. En plataformas que no soportan el cero con signo la continuidad es especificada abajo.

9.3.1 Conversión a y desde coordenadas polares

Un número complejo de Python `z` se almacena internamente usando coordenadas *rectangulares* o *cartesianas*. Esta determinado completamente por su *parte real* `z.real` y su *parte imaginaria* `z.imag`. Dicho de otra forma:

```
z == z.real + z.imag*1j
```

Las *coordenadas polares* dan una alternativa a la representación de números complejos. En las coordenadas polares, un número complejo `z` se define por los módulos `r` y el ángulo de fase `phi`. El módulo `r` es la distancia desde `z` hasta el origen, mientras que la fase `phi` es el ángulo que va en contra de las agujas del reloj, medido en radianes, desde el eje positivo de las X hasta el segmento de línea que une el origen con `z`.

Las siguientes funciones pueden ser usadas para convertir desde coordenadas rectangulares nativas hasta coordenadas polares y viceversa.

`cmath.phase(x)`

Retorna la fase de `x` (también conocido como el *argumento* de `x`), como un número de coma flotante. `phase(x)` es equivalente a `math.atan2(x.imag, x.real)`. El resultado se encuentra en el rango $[-\pi, \pi]$, y el tramo para esta operación se sostiene a lo largo del eje de abscisas negativo, continuo desde abajo. En sistemas con soporte para el número 0 con signo (que son la mayoría de ellos en uso vigente), esto significa que el signo del resultado es el mismo como el signo de `x.imag`, incluso donde `x.imag` es cero:

```
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```

Nota: El módulo (valor absoluto) de un número complejo `x` puede ser calculado usando la función predeterminada `abs()`. No existe otra función aparte del módulo `cmath` para esta operación.

`cmath.polar(x)`

Retorna la representación de `x` en coordenadas polares. Retorna un par `(r, phi)` donde `r` es el módulo de `x` y `phi` es la fase de `x`. `polar(x)` es equivalente a `(abs(x), phase(x))`.

`cmath.rect(r, phi)`

Retorna el número complejo `x` con coordenadas polares `r` y `phi`. Esto es equivalente a `r*(math.cos(phi) + math.sin(phi)*1j)`.

9.3.2 Funciones logarítmicas y de potencias

`cmath.exp(x)`

Retorna e elevado a la potencia de x , donde e es la base de los logaritmos naturales.

`cmath.log(x[, base])`

Retorna el logaritmo de x dada una *base*. Si la base no se especifica, retorna el logaritmo natural de x . No hay tramo, desde 0 en el eje negativo real hasta $-\infty$, continuo desde arriba.

`cmath.log10(x)`

Retorna el logaritmo en base de 10 de x . Tiene el mismo tramo que `log()`.

`cmath.sqrt(x)`

Retorna la raíz cuadrada de x . Tiene el mismo tramo que `log()`.

9.3.3 Funciones trigonométricas

`cmath.acos(x)`

Retorna el arcocoseno de x . Existen dos tramos: Uno que se extiende desde 1 sobre todo el eje de abscisas hasta ∞ , continuo desde abajo. El otro se extiende desde -1 a lo largo del eje de abscisas hasta $-\infty$, continuo por arriba.

`cmath.asin(x)`

Retorna el arcoseno de x . Este tiene los mismos tramos que `acos()`.

`cmath.atan(x)`

Retorna el arcotangente de x . Tiene dos tramos. Uno se extiende desde $1j$ a lo largo de el eje de abscisas imaginario ∞j , continuo desde la derecha. El otro extiende desde $-1j$ a lo largo de el eje de abscisas hasta $-\infty j$, continuo desde la izquierda.

`cmath.cos(x)`

Retorna el coseno de x .

`cmath.sin(x)`

Retorna el seno de x .

`cmath.tan(x)`

Retorna la tangente de x .

9.3.4 Funciones hiperbólicas

`cmath.acosh(x)`

Retorna el inverso del coseno hiperbólico de x . Tiene un tramo, que se extiende desde la izquierda desde 1 a lo largo del eje de abscisas hasta $-\infty$, continuo desde abajo.

`cmath.asinh(x)`

Retorna el inverso del seno hiperbólico de x . Tiene dos tramos. Uno se extiende desde $1j$ a lo largo de el eje de abscisas imaginario ∞j , continuo desde la derecha. El otro se extiende desde $-1j$ a lo largo de el eje de abscisas hasta $-\infty j$, continuo desde la izquierda.

`cmath.atanh(x)`

Retorna el inverso de la tangente hiperbólica de x . Tiene dos tramos: Uno se extiende desde 1 a lo largo de las abscisas reales hasta ∞ , continuo desde abajo. El otro se extiende desde -1 a lo largo de las abscisas reales hasta $-\infty$, continuo desde arriba.

`cmath.cosh(x)`

Retorna el coseno hiperbólico de x .

`cmath.sinh(x)`

Retorna el seno hiperbólico de x .

`cmath.tanh(x)`

Retorna la tangente hiperbólica de x .

9.3.5 Funciones de clasificación

`cmath.isfinite(x)`

Retorna `True` si tanto la parte imaginaria como real de x son finitas, y `False` en cualquier otro caso.

Nuevo en la versión 3.2.

`cmath.isinf(x)`

Retorna `True` si la parte real o la imaginaria de x es un infinito, y `False` en el caso contrario.

`cmath.isnan(x)`

Retorna `True` tanto si la parte real o imaginaria de x es `NaN`, y `False` en cualquier otro caso.

`cmath.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)`

Retorna `True` si los valores a y b son cercanos el uno al otro y `False` de otro modo.

Que dos valores sean o no considerados como cercanos es determinado de acuerdo al valor absoluto y las tolerancias relativas.

rel_tol es la tolerancia relativa – es el máximo valor permitido de la resta entre a y b , relativo al valor absoluto más grande de a o b . Por ejemplo, para fijar una tolerancia del 5%, usar `rel_tol=0.05`. El valor de tolerancia por defecto es `1e-09`, lo que asegura que los dos valores son los mismos en aproximadamente 9 dígitos decimales. *rel_tol* debe ser mayor que cero.

abs_tol es la tolerancia mínima absoluta – útil a la hora de hacer comparaciones cercanas al cero. *abs_tol* debe ser al menos cero.

Si no ocurren errores, el resultado será: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

Los valores especiales IEEE 754 de `NaN`, `inf` y `-inf` serán manejados de acuerdo al estándar de IEEE. Especialmente, `NaN` no se considera cercano a ningún otro valor, incluido `NaN`. `inf` y `-inf` solo son considerados cercanos a sí mismos.

Nuevo en la versión 3.5.

Ver también:

PEP 485 – Una función para probar igualdad aproximada.

9.3.6 Constantes

`cmath.pi`

La constante matemática π , como número de coma flotante.

`cmath.e`

La constante matemática e , como número de coma flotante.

`cmath.tau`

La constante matemática τ , como número de coma flotante.

Nuevo en la versión 3.6.

`cmath.inf`

Números de coma flotante de +infinito. Equivalente a `float('inf')`.

Nuevo en la versión 3.6.

`cmath.infj`

Números complejos con la parte real cero y números positivos infinitos como la parte imaginaria. Equivalente a `complex(0.0, float('inf'))`.

Nuevo en la versión 3.6.

`cmath.nan`

El valor del número de coma flotante «not a number» (`NaN`). Equivalente a `float('nan')`.

Nuevo en la versión 3.6.

`cmath.nanj`

Números complejos con parte real cero y como parte imaginaria NaN. Equivalente a `complex(0.0, float('nan'))`.

Nuevo en la versión 3.6.

Nótese que la selección de funciones es similar, pero no idéntica, a la del módulo `math`. El motivo de tener dos módulos se halla en que algunos usuarios no se encuentran interesados en números complejos, y quizás ni siquiera sepan que son. Preferirían que `math.sqrt(-1)` lance una excepción a que retorne un número complejo. Además fíjese que las funciones definidas en `cmath` siempre retornan un número complejo, incluso si la respuesta puede ser expresada como un número real (en cuyo caso el número complejo tiene una parte imaginaria de cero).

Un apunte en los tramos: Se tratan de curvas en las cuales las funciones fallan a ser continua. Son un complemento necesario de muchas funciones complejas. Se asume que si se necesitan cálculos con funciones complejas, usted entenderá sobre tramos. Consulte casi cualquier(no muy elemental) libro sobre variables complejas para saber más. Para más información en la correcta elección de los tramos para propósitos numéricos, se recomienda la siguiente bibliografía:

Ver también:

Kahan, W: Branch cuts for complex elementary functions; o, Much ado about nothing's sign bit. En 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

Source code: [Lib/decimal.py](#)

The `decimal` module provides support for fast correctly-rounded 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('0.1428571428571428571428571428571429')
```

- 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`, `Underflow` and `FloatOperation`.

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.

Ver también:

- 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=-999999, Emax=999999,
        capitals=1, clamp=0, 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('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.414213562373095048801688724')
>>> Decimal('NaN')
Decimal('NaN')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

If the `FloatOperation` signal is trapped, accidental mixing of decimals and floats in constructors or ordering comparisons raises an exception:

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [
>>> Decimal('3.5') == 3.5
True
```

Nuevo en la versión 3.3.

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')
```

If the internal limits of the C version are exceeded, constructing a decimal raises *InvalidOperation*:

```
>>> Decimal("1e999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [
```

Distinto en la versión 3.3.

Decimals interact well with much of the rest of Python. Here is a small decimal floating point flying circus:

```
>>> data = list(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)
[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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
Decimal('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('0.142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
       capitals=1, clamp=0, 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=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])
```

The *flags* entry shows that the rational approximation to π 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*="0", *context*=None)

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, as well as underscores throughout, 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
```

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 `'\uff10'` through `'\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.

Distinto en la versión 3.2: The argument to the constructor is now permitted to be a *float* instance.

Distinto en la versión 3.3: *float* arguments raise an exception if the *FloatOperation* trap is set. By default the trap is off.

Distinto en la versión 3.6: Underscores are allowed for grouping, as with integral and floating-point literals in code.

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 *int*).

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 or instances of *fractions.Fraction* in arithmetic operations: an attempt to add a *Decimal* to a *float*, for example, will raise a *TypeError*. However, it is possible to use Python's comparison operators to compare a *Decimal* instance *x* with another number *y*. This avoids confusing results when doing equality comparisons between numbers of different types.

Distinto en la versión 3.2: Mixed-type comparisons between *Decimal* instances and other numeric types are now fully supported.

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_integer_ratio()

Return a pair (n, d) of integers that represent the given *Decimal* instance as a fraction, in lowest terms and with a positive denominator:

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

The conversion is exact. Raise `OverflowError` on infinities and `ValueError` on NaNs.

Nuevo en la versión 3.6.

as_tuple()

Return a *named tuple* representation of the number: `DecimalTuple(sign, digits, exponent)`.

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.

compare(other, context=None)

Compare the values of two *Decimal* instances. *compare()* returns a *Decimal* instance, 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=None)

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.

compare_total(other, context=None)

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.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise `InvalidOperation` if the second operand cannot be converted exactly.

compare_total_mag(other, context=None)

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())`.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise `InvalidOperation` if the second operand cannot be converted exactly.

conjugate()

Just returns self, this method is only to comply with the Decimal Specification.

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.

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.

copy_sign(other, context=None)

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 context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise `InvalidOperation` if the second operand cannot be converted exactly.

exp (*context=None*)

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')
```

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`.

Nota: From Python 3.2 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')
```

Nuevo en la versión 3.1.

fma (*other, third, context=None*)

Fused multiply-add. Return `self*other+third` with no rounding of the intermediate product `self*other`.

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

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`.

is_finite ()

Return `True` if the argument is a finite number, and `False` if the argument is an infinity or a NaN.

is_infinite ()

Return `True` if the argument is either positive or negative infinity and `False` otherwise.

is_nan ()

Return `True` if the argument is a (quiet or signaling) NaN and `False` otherwise.

is_normal (*context=None*)

Return `True` if the argument is a *normal* finite number. Return `False` if the argument is zero, subnormal, infinite or a NaN.

is_qnan ()

Return `True` if the argument is a quiet NaN, and `False` otherwise.

is_signed()

Return *True* if the argument has a negative sign and *False* otherwise. Note that zeros and NaNs can both carry signs.

is_snan()

Return *True* if the argument is a signaling NaN and *False* otherwise.

is_subnormal(context=None)

Return *True* if the argument is subnormal, and *False* otherwise.

is_zero()

Return *True* if the argument is a (positive or negative) zero and *False* otherwise.

ln(context=None)

Return the natural (base e) logarithm of the operand. The result is correctly rounded using the *ROUND_HALF_EVEN* rounding mode.

log10(context=None)

Return the base ten logarithm of the operand. The result is correctly rounded using the *ROUND_HALF_EVEN* rounding mode.

logb(context=None)

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.

logical_and(other, context=None)

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.

logical_invert(context=None)

logical_invert() is a logical operation. The result is the digit-wise inversion of the operand.

logical_or(other, context=None)

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.

logical_xor(other, context=None)

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.

max(other, context=None)

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=None)

Similar to the *max()* method, but the comparison is done using the absolute values of the operands.

min(other, context=None)

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=None)

Similar to the *min()* method, but the comparison is done using the absolute values of the operands.

next_minus(context=None)

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.

next_plus(context=None)

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.

next_toward(other, context=None)

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.

normalize (*context=None*)

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=None*)

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.

quantize (*exp, rounding=None, context=None*)

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.

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.

remainder_near (*other, context=None*)

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')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate (*other*, *context=None*)

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.

same_quantum (*other*, *context=None*)

Test whether self and other have the same exponent or whether both are NaN.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise `InvalidOperation` if the second operand cannot be converted exactly.

scaleb (*other*, *context=None*)

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.

shift (*other*, *context=None*)

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.

sqrt (*context=None*)

Return the square root of the argument to full precision.

to_eng_string (*context=None*)

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=None*, *context=None*)

Identical to the `to_integral_value()` method. The `to_integral` name has been kept for compatibility with older versions.

to_integral_exact (*rounding=None*, *context=None*)

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.

to_integral_value (*rounding=None*, *context=None*)

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.

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`.

You can also use the `with` statement and the `localcontext()` function to temporarily change the active context.

`decimal.localcontext(ctx=None)`

Return a context manager that will set the current context for the active thread to a copy of `ctx` 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.

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
```

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 `prec=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, Emin=None, Emax=None, capitals=None, clamp=None, flags=None, traps=None*)

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.

prec is an integer in the range [1, *MAX_PREC*] that sets the precision for arithmetic operations in the context.

The *rounding* option is one of the constants listed in the section *Rounding Modes*.

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. *Emin* must be in the range [*MIN_EMIN*, 0], *Emax* in the range [0, *MAX_EMAX*].

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')`.

The *clamp* field is either 0 (the default) or 1. If set to 1, the exponent *e* of a *Decimal* instance representable in this context is strictly limited to the range $E_{min} - prec + 1 \leq e \leq E_{max} - prec + 1$. If *clamp* is 0 then a weaker condition holds: the adjusted exponent of the *Decimal* instance is at most *Emax*. When *clamp* is 1, a large normal number will, where possible, have its exponent reduced and a corresponding number of zeros added to its coefficient, in order to fit the exponent constraints; this preserves the value of the number but loses information about significant trailing zeros. For example:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

A *clamp* value of 1 allows compatibility with the fixed-width decimal interchange formats specified in IEEE 754.

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*) anywhere that a *Decimal* instance is accepted.

clear_flags()

Resets all of the flags to 0.

clear_traps()

Resets all of the traps to 0.

Nuevo en la versión 3.3.

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 or underscores are 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):
...
decimal.Inexact: None
```

Nuevo en la versión 3.1.

Etiny()

Returns a value equal to $E_{\min} - \text{prec} + 1$ which is the minimum exponent value for subnormal results. When underflow occurs, the exponent is set to *Etiny*.

Etop()

Returns a value equal to $E_{\max} - \text{prec} + 1$.

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=None*)
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 rounding mode of the context is used. Results are always correctly-rounded in the Python version.
- `Decimal(0) ** Decimal(0)` results in `InvalidOperation`, and if `InvalidOperation` is not trapped, then results in `Decimal('NaN')`.
- Distinto en la versión 3.3: The C module computes `power()` in terms of the correctly-rounded `exp()` and `ln()` functions. The result is well-defined but only «almost always correctly-rounded».
- 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.
- 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 Constants

The constants in this section are only relevant for the C module. They are also included in the pure Python version for compatibility.

	32-bit	64-bit
<code>decimal.MAX_PREC</code>	425000000	999999999999999999
<code>decimal.MAX_EMAX</code>	425000000	999999999999999999
<code>decimal.MIN_EMIN</code>	-425000000	-999999999999999999
<code>decimal.MIN_ETINY</code>	-849999999	-1999999999999999997

`decimal.HAVE_THREADS`The value is `True`. Deprecated, because Python now always has threads.

Obsoleto desde la versión 3.9.

decimal.HAVE_CONTEXTVAR

The default value is `True`. If Python is compiled `--without-decimal-contextvar`, the C version uses a thread-local rather than a coroutine-local context and the value is `False`. This is slightly faster in some nested context scenarios.

Nuevo en la versión 3.9: backported to 3.7 and 3.8

9.4.5 Rounding modes

decimal.ROUND_CEILING

Round towards Infinity.

decimal.ROUND_DOWN

Round towards zero.

decimal.ROUND_FLOOR

Round towards -Infinity.

decimal.ROUND_HALF_DOWN

Round to nearest with ties going towards zero.

decimal.ROUND_HALF_EVEN

Round to nearest with ties going to nearest even integer.

decimal.ROUND_HALF_UP

Round to nearest with ties going away from zero.

decimal.ROUND_UP

Round away from zero.

decimal.ROUND_05UP

Round away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise round towards zero.

9.4.6 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
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.

class decimal.FloatOperation

Enable stricter semantics for mixing floats and Decimals.

If the signal is not trapped (default), mixing floats and Decimals is permitted in the *Decimal* constructor, *create_decimal()* and all comparison operators. Both conversion and comparisons are exact. Any occurrence of a mixed operation is silently recorded by setting *FloatOperation* in the context flags. Explicit conversions with *from_float()* or *create_decimal_from_float()* do not set the flag.

Otherwise (the signal is trapped), only equality comparisons and explicit conversions are silent. All other mixed operations raise *FloatOperation*.

The following table summarizes the hierarchy of signals:

```
exceptions.ArithmeticError(exceptions.Exception)
  DecimalException
    Clamped
    DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
    Inexact
      Overflow(Inexact, Rounded)
      Underflow(Inexact, Rounded, Subnormal)
```

(continué en la próxima página)

(proviene de la página anterior)

```

InvalidOperation
Rounded
Subnormal
FloatOperation(DecimalException, exceptions.TypeError)

```

9.4.7 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('9.5111111')
>>> u + (v + w)
Decimal('9.5111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> 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-1000026')
```

9.4.8 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
```

(continué en la próxima página)

(proviene de la página anterior)

```

setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
. . .

```

9.4.9 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 = list(map(str, digits))
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    if places:
        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)

```

(continué en la próxima página)

(proviene de la página anterior)

```

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
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2

```

(continué en la próxima página)

(proviene de la página anterior)

```

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.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(sin(Decimal('0.5')))
    0.4794255386042030002732879352
    >>> 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

```

9.4.10 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 zeroes, losing significance, but keeping the value unchanged:

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```



```
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

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')
```

Q. Is the CPython implementation fast for large numbers?

A. Yes. In the CPython and PyPy3 implementations, the C/CFFI versions of the decimal module integrate the high speed *libmpdec* library for arbitrary precision correctly-rounded decimal floating point arithmetic. *libmpdec* uses *Karatsuba multiplication* for medium-sized numbers and the *Number Theoretic Transform* for very large numbers. However, to realize this performance gain, the context needs to be set for unrounded calculations.

```
>>> c = getcontext()
>>> c.prec = MAX_PREC
>>> c.Emax = MAX_EMAX
>>> c.Emin = MIN_EMIN
```

Nuevo en la versión 3.3.

9.5 fractions — Números racionales

Source code: [Lib/fractions.py](#)

El módulo *fractions* provee soporte para aritmética de números racionales.

Una instancia de *Fraction* puede construirse desde dos enteros, desde otro número racional, o desde una cadena de caracteres.

```
class fractions.Fraction (numerator=0, denominator=1)
class fractions.Fraction (other_fraction)
class fractions.Fraction (float)
class fractions.Fraction (decimal)
class fractions.Fraction (string)
```

La primera versión necesita que *numerator* y *denominator* sean instancias de *numbers.Rational* y retorna una nueva instancia de *Fraction* con valor *numerator/denominator*. Si *denominator* es 0, esto arrojará un error *ZeroDivisionError*. La segunda versión necesita que *other_fraction* sea una instancia de *numbers.Rational* y retorna una instancia *Fraction* con el mismo valor. Las restantes dos versiones aceptan igualmente instancias *float* o *decimal.Decimal* y retornan una instancia *Fraction* con exactamente el mismo valor. Nota que debido a los problemas usuales con la representación binaria en punto flotante (ver [tut-fp-issues](#)), el argumento de *Fraction(1.1)* no es exactamente igual a $11/10$, por lo que *Fraction(1.1)* no retorna *Fraction(11, 10)* como uno esperaría. (Mira la documentación para el método *limit_denominator()* abajo.) La última versión del constructor espera una cadena de caracteres o una instancia Unicode. La forma usual para esta instancia es:

```
[sign] numerator ['/' denominator]
```

donde el *sign* opcional puede ser “+” o “-” y *numerator* y *denominator* (si están presentes) son cadenas de caracteres de dígitos decimales. Además, cualquier cadena de caracteres que represente un valor finito y sea aceptado por el constructor de *float* también es aceptado por el constructor de *Fraction*. En cualquier caso, la cadena de caracteres de entrada también puede tener espacios en blanco iniciales y / o finales. Aquí hay unos ejemplos:

```
>>> 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)
>>> 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)
```

La clase *Fraction* hereda de la clase base abstracta *numbers.Rational*, e implementa todos los métodos y operaciones de esa clase. Las instancias *Fraction* son *hashable*, y deben ser tratadas como inmutables.

Adicionalmente *Fraction* tiene los siguientes métodos y propiedades:

Distinto en la versión 3.2: El constructor de *Fraction* ahora acepta instancias de *float* y *decimal*.

numerator

Numerador de la fracción irreducible.

denominator

Denominador de la fracción irreducible.

as_integer_ratio()

Retorna una tupla de dos enteros, cuyo ratio es igual a la fracción y con un denominador positivo.

Nuevo en la versión 3.8.

from_float(*flt*)

Este método de clase construye una instancia de *Fraction* representando el valor exacto de *flt* que debe ser un *float*. Ten cuidado, observa que `Fraction.from_float(0.3)` no es el mismo valor que `Fraction(3, 10)`.

Nota: Desde Python 3.2 en adelante, puedes construir una instancia *Fraction* directamente desde *float*.

from_decimal(*dec*)

Este método de clase construye una instancia de *Fraction* representando el valor exacto de *dec*, que debe ser una instancia de *decimal.Decimal*.

Nota: Desde Python 3.2 en adelante, puedes construir una instancia *Fraction* directamente desde una instancia *decimal.Decimal*.

limit_denominator(*max_denominator=1000000*)

Busca y retorna la instancia de *Fraction* mas cercana a *self* que tenga como denominador *max_denominator*. Este método es útil para encontrar aproximaciones racionales a un número en punto flotante determinado:

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

o para recuperar un numero racional que esta representado como flotante:

```
>>> 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)
```

__floor__()

Retorna el máximo *int* \leq *self*. Este método puede accederse también a través de la función *math.floor()*:

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()

Retorna el mínimo *int* \geq *self*. Este método puede accederse también a través de la función *math.ceil()*.

`__round__()`
`__round__(ndigits)`

La primera versión retorna el valor `int` mas cercano a `self` redondeando mitades al valor par. La segunda versión redondea `self` al múltiplo mas cercano de `Fraction(1, 10**ndigits)` (lógicamente, si `ndigits` es negativo), nuevamente redondeando mitades al valor par. Este método también puede accederse a través de la función `round()`.

`fractions.gcd(a, b)`

Retorna el máximo común divisor de los enteros `a` y `b`. Si `a` o `b` no son cero, entonces el valor absoluto de `gcd(a, b)` es el entero mas grande que divide ambos, `a` y `b`. `gcd(a, b)` tiene el mismo signo que `b` si `b` no es cero; sino toma el signo de `a`. `gcd(0, 0)` retorna 0.

Obsoleto desde la versión 3.5: Usa en cambio la función `math.gcd()`.

Ver también:

Módulo `numbers` Las clases base abstractas que representan la jerarquía de números.

9.6 random —Generar números pseudoaleatorios

Código fuente: [Lib/random.py](#)

Este módulo implementa generadores de números pseudoaleatorios para varias distribuciones.

Para los enteros, existe una selección uniforme dentro de un rango. Para las secuencias, existe una selección uniforme de un elemento aleatorio, una función para generar una permutación aleatoria de una lista *in-situ* y una función para el muestreo aleatorio sin reemplazo.

Para números reales, existen funciones para calcular distribuciones uniformes, normales (Gaussianas), log-normales, exponenciales negativas, gamma y beta. Para generar distribuciones angulares está disponible la distribución de von Mises.

Casi todas las funciones del módulo dependen de la función básica `random()`, la cuál genera uniformemente un número flotante aleatorio en el rango semiabierto [0.0, 1.0). Python utiliza Mersenne Twister como generador principal. Éste produce números de coma flotante de 53 bits de precisión y tiene un periodo de $2^{19937}-1$. La implementación subyacente en C es rápida y segura para subprocesos. Mersenne Twister es uno de los generadores de números aleatorios más testados que existen. Sin embargo, al ser completamente determinístico, no es adecuado para todos los propósitos y es completamente inadecuado para fines criptográficos.

Las funciones proporcionadas por este módulo en realidad son métodos enlazados a instancias ocultas a la clase `random.Random`. Puedes instanciar tus propias instancias de `Random` para obtener generadores que no compartan estado.

La clase `Random` puede ser también subclaseada si quieres usar un generador básico diferente para tu propio diseño: en este caso, invalida los métodos `random()`, `seed()`, `getstate()` y `setstate()`. Opcionalmente, se puede substituir un método `getrandbits()` por un nuevo generador — esto permite a `randrange()` producir selecciones sobre un rango arbitrariamente amplio.

El módulo `random` también proporciona la clase `SystemRandom`, la cuál usa la función del sistema `os.urandom()` para generar números aleatorios a partir de fuentes proporcionadas por el sistema operativo.

Advertencia: Los generadores pseudoaleatorios de este módulo no deben ser utilizados con fines de seguridad. Para usos de seguridad o criptográficos, consulta el módulo `secrets`.

Ver también:

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.

Complementary-Multiply-with-Carry recipe for a compatible alternative random number generator with a long period and comparatively simple update operations.

9.6.1 Bookkeeping functions

`random.seed(a=None, version=2)`

Inicializa el generador de números aleatorios.

Si *a* es omitida o `None`, se utilizará la hora actual del sistema. Si las fuentes de aleatoriedad vienen del sistema operativo, éstas se usarán en vez de la hora del sistema (ver la función `os.urandom()` para detalles sobre su disponibilidad).

Si *a* es un entero, se usará directamente.

Con la versión 2 (la versión por defecto), un objeto `str`, `bytes`, o `bytearray` se convierte en `int` y se usarán todos sus bits.

Con la versión 1 (proporcionada para reproducir secuencias aleatorias desde versiones anteriores de Python), el algoritmo para `str` y `bytes` genera un rango de semillas más estrecho.

Distinto en la versión 3.2: El esquema que usa todos los bits en una semilla de tipo cadena, se ha movido a la versión 2.

`random.getstate()`

Retorna un objeto capturando el estado interno del generador. Este objeto puede pasarse a `setstate()` para restaurar su estado.

`random.setstate(state)`

El *state* debería haberse obtenido de una llamada previa a `getstate()`, y `setstate()` reestablece el estado interno del generador al que tenía cuando se llamó a la función `getstate()`.

`random.getrandbits(k)`

Returns a Python integer with *k* random bits. This method is supplied with the Mersenne Twister 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.

9.6.2 Funciones para enteros

`random.randrange(stop)`

`random.randrange(start, stop[, step])`

Retorna un elemento de `range(start, stop, step)` seleccionado aleatoriamente. Esto es equivalente a `choice(range(start, stop, step))`, pero en realidad no crea un objeto rango.

El patrón de argumento posicional coincide con el de `range()`. Los argumentos no deben usarse porque la función puede usarlos de forma inesperada.

Distinto en la versión 3.2: `randrange()` es más sofisticado produciendo valores igualmente distribuidos. Anteriormente utilizaba un estilo como `int(random()*n)` el cual puede producir distribuciones ligeramente desiguales.

`random.randint(a, b)`

Retorna un entero aleatorio *N* tal que $a \leq N \leq b$. Alias de `randrange(a, b+1)`.

9.6.3 Funciones para secuencias

`random.choice(seq)`

Retorna un elemento aleatorio de una secuencia *seq* no vacía. Si *seq* está vacía, lanza `IndexError`.

`random.choices(population, weights=None, *, cum_weights=None, k=1)`

Retorna una lista de elementos de tamaño *k* elegidos de la *population* con reemplazo. Si la *population* está vacía, lanza `IndexError`.

Si se especifica una secuencia *weights*, las selecciones se realizan de acuerdo con las ponderaciones relativas. Alternativamente, si se da una secuencia *cum_weights*, las selecciones se harán según los pesos cumulativos (posiblemente se calculen usando `itertools.accumulate()`). Por ejemplo, los pesos relativos `[10, 5, 30, 5]` son equivalentes a los pesos cumulativos `[10, 15, 45, 50]`. Internamente, los pesos relativos se convierten en pesos cumulativos antes de hacer selecciones, por lo cual suplir los pesos cumulativos ahorra trabajo.

Si ni *weights* ni *cum_weights* están especificadas, las selecciones se realizan con la misma probabilidad. Si se proporciona una secuencia de ponderaciones, debe tener la misma longitud que la secuencia *population*. Es un `TypeError` especificar ambas *weights* y *cum_weights*.

The *weights* or *cum_weights* can use any numeric type that interoperates with the `float` values returned by `random()` (that includes integers, floats, and fractions but excludes decimals). Weights are assumed to be non-negative.

Dada una semilla, la función `choices()` normalmente produce una secuencia diferente a las llamadas repetidas a `choice()` con la misma ponderación. El algoritmo usado por `choices()` emplea aritmética de coma flotante para la consistencia interna y velocidad. El algoritmo usado por `choice()` emplea por defecto aritmética de enteros con selecciones repetidas para evitar pequeños sesgos de errores de redondeo.

Nuevo en la versión 3.6.

`random.shuffle(x[, random])`

Shuffle the sequence *x* in place.

El argumento opcional *random* es una función de 0 argumentos que retorna un flotante random en `[0.0, 1.0)`; por defecto esta es la función `random()`.

Para bajar una secuencia inmutable y retornar una nueva lista barajada, utilice `sample(x, k=len(x))` en su lugar.

Tenga en cuenta que incluso para pequeños `len(x)`, el número total de permutaciones de *x* puede crecer rápidamente más que el periodo de muchos generadores de números aleatorios. Esto implica que la mayoría de las permutaciones de una secuencia larga nunca se pueden generar. Por ejemplo, una secuencia de longitud 2080 es la más grande que cabe dentro del período del generador de números aleatorios de Mersenne Twister.

`random.sample(population, k)`

Retorna una lista de longitud *k* de elementos únicos elegidos de la secuencia de población o conjunto. Se utiliza para el muestreo aleatorio sin reemplazo.

Retorna una nueva lista que contiene elementos de la población sin modificar la población original. La lista resultante está en orden de selección de forma que todos los subsectores también son muestras aleatorias válidas. Esto permite que los ganadores de la rifa (la muestra) se dividan en primer premio y ganadores del segundo lugar (los subsectores).

Los miembros de la población no tienen porqué ser *hashable* o únicos. Si la población incluye repeticiones, entonces cada ocurrencia es una posible selección en la muestra.

Para escoger una muestra de un rango de enteros, use un objeto `range()` como argumento. Esto es especialmente rápido y eficiente en espacio para el muestreo de poblaciones grandes: `sample(range(10000000), k=60)`.

Si el tamaño de la muestra es mayor que el tamaño de la población, se lanzará un `ValueError`.

9.6.4 Distribuciones para los nombres reales

Las siguientes funciones generan distribuciones específicas para números reales. Los parámetros de la función reciben el nombre de las variables correspondientes en la ecuación de distribución, tal y como se utilizan en la práctica matemática común.; la mayoría de estas ecuaciones se pueden encontrar en cualquier texto estadístico.

`random.random()`

Retorna el siguiente número en coma flotante aleatorio dentro del rango [0.0, 1.0).

`random.uniform(a, b)`

Retorna un número en coma flotante aleatorio N tal que $a \leq N \leq b$ para $a \leq b$ y $b \leq N \leq a$ para $b < a$.

El valor final b puede o no estar incluido en el rango, dependiendo del redondeo de coma flotante en la ecuación $a + (b-a) * \text{random}()$.

`random.triangular(low, high, mode)`

Retorna un número de coma flotante N tal que $\text{low} \leq N \leq \text{high}$ y con el *mode* especificado entre esos límites. Los límites *low* (inferior) y *high* (superior) son por defecto cero y uno. El argumento *mode* tiene como valor por defecto el punto medio entre los límites, dando lugar a una distribución simétrica.

`random.betavariate(alpha, beta)`

Distribución beta. Las condiciones de los parámetros son $\alpha > 0$ y $\beta > 0$. Retorna valores dentro del rango entre 0 y 1.

`random.expovariate(lambd)`

Distribución exponencial. *lambd* es 1.0 dividido entre la media deseada. Debe ser distinto a cero (El parámetro debería llamarse *lambda* pero esa es una palabra reservada en Python). Retorna valores dentro del rango de 0 a infinito positivo si *lambd* es positivo, y de infinito negativo a 0 si *lambd* es negativo.

`random.gammavariate(alpha, beta)`

Distribución gamma. (¡No la función gamma!) Las condiciones en los parámetros son $\alpha > 0$ y $\beta > 0$.

La función de distribución de la probabilidad es:

$$\text{pdf}(x) = \frac{x^{(\alpha - 1)} * \text{math.exp}(-x / \beta)}{\text{math.gamma}(\alpha) * \beta^{\alpha}}$$

`random.gauss(mu, sigma)`

Distribución gaussiana. *mu* es la media y *sigma* es la desviación estándar. Es un poco más rápida que la función *normalvariate()* definida debajo.

`random.lognormvariate(mu, sigma)`

Logaritmo de la distribución normal. Si se usa un logaritmo natural de esta distribución, se obtendrá una distribución normal con media *mu* y desviación estándar *sigma*. *mu* puede tener cualquier valor, y *sigma* debe ser mayor que cero.

`random.normalvariate(mu, sigma)`

Distribución normal. *mu* es la media y *sigma* es la desviación estándar.

`random.vonmisesvariate(mu, kappa)`

mu es el ángulo medio, expresado en radianes entre 0 y 2π , y *kappa* es el parámetro de concentración, que debe ser mayor o igual a cero. Si *kappa* es igual a cero, esta distribución se reduce a un ángulo aleatorio uniforme sobre el rango de 0 a 2π .

`random.paretovariate(alpha)`

Distribución de Pareto. *alpha* es el parámetro de forma.

`random.weibullvariate(alpha, beta)`

Distribución de Weibull. *alpha* es el parámetro de escala y *beta* es el parámetro de forma.

9.6.5 Generador alternativo

class `random.Random([seed])`

Esta clase implementa el generador de números pseudoaleatorios predeterminado que usa el módulo `random`.

class `random.SystemRandom([seed])`

Clase que utiliza la función `os.urandom()` para generar números aleatorios a partir de fuentes proporcionadas por el sistema operativo. No está disponible en todos los sistemas. No se basa en el estado del software y las secuencias no son reproducibles. En consecuencia, el método `seed()` no tiene efecto y es ignorado. Los métodos `getstate()` y `setstate()` lanzan `NotImplementedError` si se les llama.

9.6.6 Notas sobre la Reproducibilidad

A veces es necesario poder reproducir las secuencias dadas por un generador de números pseudoaleatorios. Al volver a usar el valor de una semilla, la misma secuencia debería ser reproducible en cada ejecución siempre que no se ejecuten múltiples subprocesos.

Muchos de los algoritmos y de las funciones de generación de semillas del módulo aleatorio pueden cambiar entre versiones de Python, pero se garantiza que dos aspectos no cambien:

- Si se añade un nuevo método de generación de semilla, se ofrecerá un generador de semilla retrocompatible.
- El método generador `random()` continuará produciendo la misma secuencia cuando se le da la misma semilla al generador de semilla compatible.

9.6.7 Ejemplos y Recetas

Ejemplos básicos:

```
>>> random()                                # Random float:  0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                      # Random float:  2.5 <= x < 10.0
3.1800146073117523

>>> expovariate(1 / 5)                      # Interval between arrivals averaging 5
↪seconds
5.148957571865031

>>> randrange(10)                           # Integer from 0 to 9 inclusive
7

>>> randrange(0, 101, 2)                    # Even integer from 0 to 100 inclusive
26

>>> choice(['win', 'lose', 'draw'])          # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                           # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4)        # Four samples without replacement
[40, 10, 50, 30]
```

Simulaciones:


```

>>> # Six roulette wheel spins (weighted sampling with replacement)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # Deal 20 cards without replacement from a deck of 52 playing cards
>>> # and determine the proportion of cards with a ten-value
>>> # (a ten, jack, queen, or king).
>>> deck = collections.Counter(tens=16, low_cards=36)
>>> seen = sample(list(deck.elements()), k=20)
>>> seen.count('tens') / 20
0.15

>>> # Estimate the probability of getting 5 or more heads from 7 spins
>>> # of a biased coin that settles on heads 60% of the time.
>>> def trial():
...     return choices('HT', cum_weights=(0.60, 1.00), k=7).count('H') >= 5
...
>>> sum(trial() for i in range(10_000)) / 10_000
0.4169

>>> # Probability of the median of 5 samples being in middle two quartiles
>>> def trial():
...     return 2_500 <= sorted(choices(range(10_000), k=5))[2] < 7_500
...
>>> sum(trial() for i in range(10_000)) / 10_000
0.7958

```

Example of statistical bootstrapping using resampling with replacement to estimate a confidence interval for the mean of a sample:

```

# http://statistics.about.com/od/Applications/a/Example-Of-Bootstrapping.htm
from statistics import fmean as mean
from random import choices

data = [41, 50, 29, 37, 81, 30, 73, 63, 20, 35, 68, 22, 60, 31, 95]
means = sorted(mean(choices(data, k=len(data))) for i in range(100))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[5]:.1f} to {means[94]:.1f}')

```

Ejemplo de un test de permutación en remuestreo (en) para determinar la significación estadística o p-valor de una diferencia observada entre los efectos de un fármaco y un placebo:

```

# Example from "Statistics is Easy" by Dennis Shasha and Manda Wilson
from statistics import fmean as mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10_000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances with a difference')
print(f'at least as extreme as the observed difference of {observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to reject the null')
print(f'hypothesis that there is no difference between the drug and the placebo.')

```

Simulation of arrival times and service deliveries for a multiserver queue:

```
from heapq import heappush, heappop
from random import expovariate, gauss
from statistics import mean, median, stdev

average_arrival_interval = 5.6
average_service_time = 15.0
stdev_service_time = 3.5
num_servers = 3

waits = []
arrival_time = 0.0
servers = [0.0] * num_servers # time when each server becomes available
for i in range(100_000):
    arrival_time += expovariate(1.0 / average_arrival_interval)
    next_server_available = heappop(servers)
    wait = max(0.0, next_server_available - arrival_time)
    waits.append(wait)
    service_duration = gauss(average_service_time, stdev_service_time)
    service_completed = arrival_time + wait + service_duration
    heappush(servers, service_completed)

print(f'Mean wait: {mean(waits):.1f}. Stdev wait: {stdev(waits):.1f}.')
print(f'Median wait: {median(waits):.1f}. Max wait: {max(waits):.1f}.')
```

Ver también:

Statistics for Hackers un video tutorial de Jake Vanderplas sobre análisis estadístico usando sólo algunos conceptos fundamentales incluyendo simulación, muestreo, baraja y validación cruzada.

Economics Simulation <<http://nbviewer.jupyter.org/url/norvig.com/ipython/Economics.ipynb>> ‘una simulación de un mercado por Peter Norvig que muestra un uso efectivo de las herramientas y distribuciones proporcionadas por este modulo (*gauss*, *uniform*, *sample*, *betavariate*, *choice*, *triangular*, y *randrange*).

A Concrete Introduction to Probability (using Python) <<http://nbviewer.jupyter.org/url/norvig.com/ipython/Probability.ipynb>> ‘un tutorial de Peter Norvig cubriendo teoría básica de probabilidad, cómo escribir simulaciones y cómo realizar un análisis de datos usando Python.

9.7 statistics — Funciones de estadística matemática

Nuevo en la versión 3.4.

Código fuente: [Lib/statistics.py](#)

Este módulo proporciona funciones para calcular estadísticas matemáticas de datos numéricos (de tipo *Real*).

Este módulo no pretende ser competidor o sustituto de bibliotecas de terceros como [NumPy](#) o [SciPy](#), ni de paquetes completos de software propietario para profesionales como Minitab, SAS o Matlab. Este módulo se ubica a nivel de calculadoras científicas gráficas.

A menos que se indique explícitamente lo contrario, las funciones de este módulo manejan objetos *int*, *float*, *Decimal* y *Fraction*. No se garantiza un correcto funcionamiento con otros tipos (numéricos o no). El comportamiento de estas funciones con colecciones mixtas que contengan objetos de diferente tipo no está definido y depende de la implementación. Si tus datos de entrada consisten en una mezcla de varios tipos, puedes usar *map()* para asegurarte de que el resultado sea consistente, por ejemplo: *map(float, input_data)*.

9.7.1 Promedios y medidas de tendencia central

Estas funciones calculan el promedio o el valor típico de una población o muestra.

<code>mean()</code>	Media aritmética («promedio») de los datos.
<code>fmean()</code>	Media aritmética usando coma flotante, más rápida.
<code>geometric_mean()</code>	Media geométrica de los datos.
<code>harmonic_mean()</code>	Media armónica de los datos.
<code>median()</code>	Mediana (valor central) de los datos.
<code>median_low()</code>	Mediana baja de los datos.
<code>median_high()</code>	Mediana alta de los datos.
<code>median_grouped()</code>	Mediana, o percentil 50, de los datos agrupados.
<code>mode()</code>	Moda única (valor más común) de datos discretos o nominales.
<code>multimode()</code>	Lista de modas (valores más comunes) de datos discretos o nominales.
<code>quantiles()</code>	Divide los datos en intervalos equiprobables.

9.7.2 Medidas de dispersión

Estas funciones calculan una medida de cuánto tiende a desviarse la población o muestra de los valores típicos o promedios.

<code>pstdev()</code>	Desviación típica poblacional de los datos.
<code>pvariance()</code>	Varianza poblacional de los datos.
<code>stdev()</code>	Desviación típica muestral de los datos.
<code>variance()</code>	Varianza muestral de los datos.

9.7.3 Detalles de las funciones

Nota: Las funciones no requieren que se ordenen los datos que se les proporcionan. Sin embargo, para facilitar la lectura, la mayoría de los ejemplos muestran secuencias ordenadas.

`statistics.mean(data)`

Retorna la media aritmética muestral de *data*, que puede ser una secuencia o un iterable.

La media aritmética es la suma de los valores dividida entre el número de observaciones. Es comúnmente denominada «promedio», aunque hay muchas formas de definir el promedio matemáticamente. Es una medida de tendencia central de los datos.

Se lanza una excepción `StatisticsError` si *data* está vacío.

Algunos ejemplos de uso:

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

Nota: La media aritmética se ve fuertemente afectada por la presencia de valores atípicos en la muestra y no es un estimador robusto de tendencia central: la media no es necesariamente un ejemplo representativo de la

muestra. Consulta `median()` y `mode()` para obtener medidas más robustas de tendencia central.

La media muestral proporciona una estimación no sesgada de la media real de la población. Por lo tanto, al calcular el promedio de todas las muestras posibles, `mean(sample)` converge con el promedio real de toda la población. Si `data` representa a una población completa, en lugar de a una muestra, entonces `mean(data)` equivale a calcular la media poblacional verdadera μ .

`statistics.fmean(data)`

Convierte los valores de `data` a flotantes y calcula la media aritmética.

Esta función se ejecuta más rápido que `mean()` y siempre retorna un `float`. `data` puede ser una secuencia o un iterable. Si el conjunto de datos de entrada está vacío, se lanza una excepción `StatisticsError`.

```
>>> fmean([3.5, 4.0, 5.25])
4.25
```

Nuevo en la versión 3.8.

`statistics.geometric_mean(data)`

Convierte los valores de `data` a flotantes y calcula la media geométrica.

La media geométrica indica la tendencia central o valor típico de `data` utilizando el producto de los valores (en oposición a la media aritmética, que utiliza su suma).

Lanza una excepción `StatisticsError` si el conjunto de datos de entrada está vacío, o si contiene un cero o un valor negativo. `data` puede ser una secuencia o un iterable.

No se toman medidas especiales para garantizar que el resultado sea completamente preciso. (Sin embargo, esto puede cambiar en una versión futura.)

```
>>> round(geometric_mean([54, 24, 36]), 1)
36.0
```

Nuevo en la versión 3.8.

`statistics.harmonic_mean(data)`

Retorna la media armónica de `data`, que debe ser una secuencia o un iterable de números que pertenezcan al conjunto de los números reales.

La media armónica es el inverso o recíproco de la media aritmética (`mean()`) de los inversos multiplicativos de los datos. Por ejemplo, la media armónica de tres valores a , b y c es $3 / (1/a + 1/b + 1/c)$. El resultado es cero si uno de los valores es cero.

La media armónica es un tipo de promedio, una medida de la tendencia central de los datos. Generalmente es adecuada para calcular promedios de tasas o fracciones, por ejemplo, velocidades.

Supongamos que un automóvil viaja 10 km a 40 km/h, luego otros 10 km a 60 km/h. ¿Cuál es su velocidad media?

```
>>> harmonic_mean([40, 60])
48.0
```

Supongamos que un inversor compra la misma cantidad de acciones de tres empresas diferentes, con unos PER (ratio precio-beneficio) de 2.5, 3 y 10. ¿Cuál es el PER promedio para la cartera del inversor?

```
>>> harmonic_mean([2.5, 3, 10]) # For an equal investment portfolio.
3.6
```

Una excepción `StatisticsError` es lanzada si `data` está vacío o algún elemento es menor que cero.

El algoritmo actual tiene una salida anticipada cuando encuentra un cero en la entrada. Esto significa que no se comprueba la validez de las entradas posteriores al cero. (Este comportamiento puede cambiar en el futuro.)

Nuevo en la versión 3.6.

`statistics.median(data)`

Retorna la mediana (valor central) de los datos numéricos, utilizando el método clásico de «media de los dos del medio». Si *data* está vacío, se lanza una excepción *StatisticsError*. *data* puede ser una secuencia o un iterable.

La mediana es una medida de tendencia central robusta y es menos sensible a la presencia de valores atípicos que la media. Cuando el número de casos es impar, se retorna el valor central:

```
>>> median([1, 3, 5])
3
```

Cuando el número de observaciones es par, la mediana se interpola calculando el promedio de los dos valores centrales:

```
>>> median([1, 3, 5, 7])
4.0
```

Este enfoque es adecuado para datos discretos, siempre que se acepte que la mediana no es necesariamente parte de las observaciones.

Si los datos son ordinales (se pueden ordenar) pero no numéricos (no se pueden sumar), considera usar *median_low()* o *median_high()* en su lugar.

`statistics.median_low(data)`

Retorna la mediana baja de los datos numéricos. Se lanza una excepción *StatisticsError* si *data* está vacío. *data* puede ser una secuencia o un iterable.

La mediana baja es siempre un valor presente en el conjunto de datos. Cuando el número de casos es impar, se retorna el valor central. Cuando el número de casos es par, se retorna el menor de los dos valores centrales.

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

Utiliza la mediana baja cuando tus datos sean discretos y prefieras que la mediana sea un valor representado en tus observaciones, en lugar de ser el resultado de una interpolación.

`statistics.median_high(data)`

Retorna la mediana alta de los datos. Lanza una excepción *StatisticsError* si *data* está vacío. *data* puede ser una secuencia o un iterable.

La mediana alta es siempre un valor presente en el conjunto de datos. Cuando el número de casos es impar, se retorna el valor central. Cuando el número de casos es par, se retorna el mayor de los dos valores centrales.

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

Utiliza la mediana alta cuando tus datos sean discretos y prefieras que la mediana sea un valor representado en tus observaciones, en lugar de ser el resultado de una interpolación.

`statistics.median_grouped(data, interval=1)`

Retorna la mediana de los datos continuos agrupados, calculada como el percentil 50, usando interpolación. Se lanza una excepción *StatisticsError* si *data* está vacío. *data* puede ser una secuencia o un iterable.

```
>>> median_grouped([52, 52, 53, 54])
52.5
```

En el siguiente ejemplo, los valores se redondean para que cada valor represente la mitad de un grupo. Por ejemplo, 1 es la mitad del grupo 0.5–1.5, 2 es la mitad del grupo 1.5–2.5, 3 es la mitad de 2.5–3.5, etc. En los datos proporcionados a continuación, el valor medio está en algún lugar del grupo que va de 3,5 a 4,5 y se estima mediante interpolación:

```
>>> median_grouped([1, 2, 2, 3, 4, 4, 4, 4, 4, 5])
3.7
```

El argumento opcional *interval* representa el intervalo de clase y el valor predeterminado es 1. Cambiar el intervalo de clase cambiará la interpolación, como es natural:

```
>>> median_grouped([1, 3, 3, 5, 7], interval=1)
3.25
>>> median_grouped([1, 3, 3, 5, 7], interval=2)
3.5
```

Esta función no comprueba si los valores están separados por al menos un *interval*.

CPython implementation detail: Bajo algunas circunstancias, `median_grouped()` puede convertir algunos de los valores proporcionados en flotantes. Es probable que este comportamiento cambie en el futuro.

Ver también:

- «Statistics for the Behavioral Sciences», Frederick J Gravetter y Larry B Wallnau (8ª edición).
- La función `SSMEDIAN` del programa de hojas de cálculo Gnumeric de Gnome, incluyendo [esta discusión](#).

`statistics.mode(data)`

Retorna el valor más común del conjunto de datos discretos o nominales *data*. La moda (cuando existe) es el valor más representativo y sirve como medida de tendencia central.

Si hay varias modas con la misma frecuencia, retorna la primera encontrada en *data*. Si deseas la menor o la mayor de ellas, usa `min(multimode(data))` o `max(multimode(data))`. Se lanza una excepción `StatisticsError` si la entrada *data* está vacía.

`mode` asume que los datos de entrada son discretos y retorna un solo valor. Esta es la definición habitual de la moda que se enseña en las escuelas:

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

La moda tiene la particularidad de ser la única estadística de este módulo que se puede calcular sobre datos nominales (no numéricos):

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

Distinto en la versión 3.8: Ahora maneja conjuntos de datos multimodales, retornando la primera moda encontrada. Anteriormente, se lanzaba una excepción `StatisticsError` cuando se daba esta situación.

`statistics.multimode(data)`

Retorna una lista de los valores más frecuentes en el orden en que aparecen en *data*. Retornará varios resultados en el caso de que existan varias modas, o una lista vacía si *data* está vacío:

```
>>> multimode('aabbbbccdddeeffffgg')
['b', 'd', 'f']
>>> multimode('')
[]
```

Nuevo en la versión 3.8.

`statistics.pstdev(data, mu=None)`

Retorna la desviación típica poblacional (la raíz cuadrada de la varianza poblacional). Consultar `pvariance()` para los argumentos y otros detalles.

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

`statistics.pvariance` (*data*, *mu=None*)

Retorna la varianza poblacional de *data*, que debe ser una secuencia no vacía o un iterable de números reales. La varianza, o momento de segundo orden respecto a la media, es una medida de la variabilidad (o dispersión) de los datos. Una alta varianza indica una amplia dispersión de valores; una varianza baja indica que los valores están agrupados alrededor de la media.

El segundo argumento opcional *mu*, que normalmente será la media de *data*, también se puede utilizar para calcular el momento de segundo orden alrededor de un punto que no es la media. Si no se proporciona o es `None` (el valor predeterminado), la media aritmética se calcula automáticamente.

Utiliza esta función para calcular la varianza de toda la población. Para estimar la varianza de una muestra, la función `variance()` suele ser una opción mejor.

Lanza una excepción `StatisticsError` si *data* está vacío.

Ejemplos:

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

Si ya has calculado la media de tus datos, puedes pasarla como segundo argumento opcional *mu* para evitar que se tenga que volver a calcular:

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

Se admiten decimales (`Decimal`) y fracciones (`Fraction`):

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

Nota: Esta función retorna la varianza poblacional σ^2 cuando se aplica a toda la población. Si se aplica solo a una muestra, el resultado es la varianza muestral s^2 , conocida también como varianza con *N* grados de libertad.

Si se conoce de antemano la verdadera media poblacional μ , se puede usar esta función para calcular la varianza muestral, pasando la media poblacional conocida como segundo argumento. Suponiendo que las observaciones provienen de una selección aleatoria uniforme de la población, el resultado será una estimación no sesgada de la varianza poblacional.

`statistics.stdev` (*data*, *xbar=None*)

Retorna la desviación típica muestral (la raíz cuadrada de la varianza muestral). Consultar `variance()` para los argumentos y otros detalles.

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

`statistics.variance` (*data*, *xbar=None*)

Retorna la varianza muestral de *data*, que debe ser un iterable de al menos dos números reales. La varianza, o momento de segundo orden respecto a la media, es una medida de la variabilidad (difusión o dispersión) de los datos. Una alta varianza indica que los datos están dispersos; una baja varianza indica que los datos están agrupados estrechamente alrededor de la media.

Si se proporciona el segundo argumento opcional *xbar*, este debe ser la media aritmética de *data*. Si no se proporciona o es `None` (el valor predeterminado), la media aritmética se calcula automáticamente.

Utiliza esta función cuando tus datos sean una muestra de una población. Para calcular la varianza de toda la población, consulta `pvariance()`.

Lanza una excepción `StatisticsError` si `data` tiene menos de dos valores.

Ejemplos:

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

Si previamente se ha calculado la media de los datos, puede pasarse como segundo argumento opcional `xbar` para evitar que se tenga que volver a calcular:

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

Esta función no comprueba si el valor pasado al argumento `xbar` corresponde al promedio. El uso de valores arbitrarios para `xbar` produce resultados imposibles o incorrectos.

La función maneja decimales (`Decimal`) y fracciones (`Fraction`):

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

Nota: Esta es la varianza muestral s^2 con la corrección de Bessel, también conocida como varianza con $N-1$ grados de libertad. Suponiendo que las observaciones son representativas de la población (es decir, independientes y distribuidas de forma idéntica), el resultado es una estimación no sesgada de la varianza.

Si conoces de antemano la verdadera media poblacional μ , debes pasarla a `pvariance()` mediante el parámetro `mu` para obtener la varianza muestral.

`statistics.quantiles` (`data`, `*`, `n=4`, `method='exclusive'`)

Divide `data` en `n` intervalos continuos equiprobables. Retorna una lista de `n - 1` límites que delimitan los intervalos (cuantiles).

Establece `n` en 4 para obtener los cuartiles (el valor predeterminado), en 10 para obtener los deciles y en 100 para obtener los percentiles (lo que produce 99 valores que separan `data` en 100 grupos del mismo tamaño). Si `n` es menor que 1, se lanza una excepción `StatisticsError`.

`data` puede ser cualquier iterable que contenga los valores de la muestra. Para que los resultados sean significativos, el número de observaciones en la muestra `data` debe ser mayor que `n`. Si no hay al menos dos observaciones se lanza una excepción `StatisticsError`.

Los límites de los intervalos se interpolan linealmente a partir de los dos valores más cercanos de la muestra. Por ejemplo, si un límite es un tercio de la distancia entre los valores 100 y 112 de la muestra, el límite será 104.

El argumento `method` indica el método que se utilizará para calcular los cuantiles y se puede modificar para especificar si se deben incluir o excluir valores de `data` extremos, altos y bajos, de la población.

El valor predeterminado para `method` es «exclusive» y es aplicable a los datos muestreados de una población que puede tener valores más extremos que los encontrados en las muestras. La proporción de la población que se encuentra por debajo del *i*-ésimo valor de *m* valores ordenados se calcula mediante la fórmula $i / (m + 1)$. Por ejemplo, asumiendo que hay 9 valores en la muestra, este método los ordena y los asocia con los siguientes percentiles: 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%.

Si se usa «inclusive» como valor para el parámetro *method*, se asume que los datos corresponden a una población completa o que los valores extremos de la población están representados en la muestra. El valor mínimo de *data* se considera entonces como percentil 0 y el máximo como percentil 100. La proporción de la población que se encuentra por debajo del *i*-ésimo valor de *m* valores ordenados se calcula mediante la fórmula $(i - 1) / (m - 1)$. Suponiendo que tenemos 11 valores en la muestra, este método los ordena y los asocia con los siguientes percentiles: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80 %, 90%, 100%.

```
# Decile cut points for empirically sampled data
>>> data = [105, 129, 87, 86, 111, 111, 89, 81, 108, 92, 110,
...         100, 75, 105, 103, 109, 76, 119, 99, 91, 103, 129,
...         106, 101, 84, 111, 74, 87, 86, 103, 103, 106, 86,
...         111, 75, 87, 102, 121, 111, 88, 89, 101, 106, 95,
...         103, 107, 101, 81, 109, 104]
>>> [round(q, 1) for q in quantiles(data, n=10)]
[81.0, 86.2, 89.0, 99.4, 102.5, 103.6, 106.0, 109.8, 111.0]
```

Nuevo en la versión 3.8.

9.7.4 Excepciones

Se define una sola excepción:

exception `statistics.StatisticsError`

Subclase de `ValueError` para excepciones relacionadas con la estadística.

9.7.5 Objetos `NormalDist`

`NormalDist` es una herramienta para crear y manipular distribuciones normales de una *variable aleatoria*. Esta clase gestiona la desviación típica y la media de un conjunto de observaciones como una sola entidad.

Las distribuciones normales surgen del *Teorema del límite central* y tienen una amplia gama de aplicaciones en estadística.

class `statistics.NormalDist (mu=0.0, sigma=1.0)`

Retorna un nuevo objeto `NormalDist` donde *mu* representa la *media aritmética* y *sigma* representa la *desviación típica*.

Se lanza una excepción `StatisticsError` si *sigma* es negativo.

mean

Una propiedad de solo lectura para la *media aritmética* de una distribución normal.

median

Una propiedad de solo lectura para la *mediana* de una distribución normal.

mode

Una propiedad de solo lectura para la *moda* de una distribución normal.

stdev

Una propiedad de solo lectura para la *desviación típica* de una distribución normal.

variance

Una propiedad de solo lectura para la *varianza* de una distribución normal. Es igual al cuadrado de la desviación típica.

classmethod `from_samples (data)`

Crea una instancia de distribución normal con los parámetros *mu* y *sigma* estimados a partir de *data* usando `fmean()` y `stdev()`.

data puede ser cualquier *iterable* de valores que se puedan convertir al tipo `float`. Se lanza una excepción `StatisticsError` si *data* no contiene al menos dos elementos, esto se debe a que se necesita al menos un punto para estimar un valor central y al menos dos puntos para estimar la dispersión.

samples (*n*, *, *seed*=None)

Genera *n* muestras aleatorias para una media y una desviación típica proporcionadas. Retorna un objeto *list* de valores *float*.

Si se proporciona *seed*, su valor se usa para inicializar una nueva instancia del generador de números aleatorios subyacente. Esto permite producir resultados reproducibles incluso en un contexto de paralelismo con múltiples hilos.

pdf (*x*)

Haciendo uso de una [función de densidad de probabilidad \(FPD o PDF en inglés\)](#), calcula la verosimilitud relativa de que una variable aleatoria *X* caiga en una región cercana al valor *x* proporcionado. Matemáticamente, esto corresponde al límite de la razón $P(x \leq X < x+dx) / dx$ cuando *dx* tiende a cero.

La verosimilitud relativa se calcula como la probabilidad de que una observación pertenezca a un intervalo estrecho dividida entre el ancho del intervalo (de ahí el término «densidad»). Como la verosimilitud es relativa a los otros puntos, su valor puede ser mayor que 1.0.

cdf (*x*)

Usando una [función de distribución acumulada \(FDA, CDF en inglés\)](#), calcula la probabilidad de que una variable aleatoria *X* sea menor o igual que *x*. Matemáticamente, se escribe $P(X \leq x)$.

inv_cdf (*p*)

Calcula la función de distribución acumulada inversa, también conocida como [función cuantil](#) o [función punto porcentual](#). Matemáticamente, se escribe $x : P(X \leq x) = p$.

Calcula el valor *x* de la variable aleatoria *X* tal que la probabilidad de que la variable sea menor o igual a este valor es igual a la probabilidad *p* dada.

overlap (*other*)

Mide la concordancia entre dos distribuciones de probabilidad normales. Retorna un valor entre 0.0 y 1.0 que indica [el área de superposición de dos funciones de densidad de probabilidad](#).

quantiles (*n*=4)

Divide la distribución normal en *n* intervalos continuos equiprobables. Retorna una lista de (*n* - 1) cuantiles que separan los intervalos.

Establece *n* en 4 para obtener los cuartiles (el valor predeterminado), en 10 para obtener los deciles y en 100 para obtener los percentiles (lo que produce 99 límites que separan los datos en 100 grupos del mismo tamaño).

Las instancias de la clase *NormalDist* soportan la suma, resta, multiplicación y división por una constante. Estas operaciones se pueden utilizar para traducir o escalar, por ejemplo:

```
>>> temperature_february = NormalDist(5, 2.5)           # Celsius
>>> temperature_february * (9/5) + 32                  # Fahrenheit
NormalDist(mu=41.0, sigma=4.5)
```

No se admite la división de una constante entre una instancia de *NormalDist* debido a que el resultado no sería una distribución normal.

Dado que las distribuciones normales se derivan de las propiedades aditivas de variables independientes, es posible [sumar o restar dos variables independientes con distribución normal](#) representadas por instancias de *NormalDist*. Por ejemplo :

```
>>> birth_weights = NormalDist.from_samples([2.5, 3.1, 2.1, 2.4, 2.7, 3.5])
>>> drug_effects = NormalDist(0.4, 0.15)
>>> combined = birth_weights + drug_effects
>>> round(combined.mean, 1)
3.1
>>> round(combined.stdev, 1)
0.5
```

Nuevo en la versión 3.8.

Ejemplos de uso de `NormalDist`

`NormalDist` permite resolver fácilmente problemas probabilísticos clásicos.

Por ejemplo, sabiendo que los datos históricos de los exámenes SAT siguen una distribución normal con una media de 1060 y una desviación típica de 195, determinar el porcentaje de estudiantes con puntuaciones entre 1100 y 1200, redondeado al número entero más cercano:

```
>>> sat = NormalDist(1060, 195)
>>> fraction = sat.cdf(1200 + 0.5) - sat.cdf(1100 - 0.5)
>>> round(fraction * 100.0, 1)
18.4
```

Determinar los cuartiles y deciles de las puntuaciones del SAT:

```
>>> list(map(round, sat.quantiles()))
[928, 1060, 1192]
>>> list(map(round, sat.quantiles(n=10)))
[810, 896, 958, 1011, 1060, 1109, 1162, 1224, 1310]
```

Con la finalidad de estimar la distribución de un modelo que es difícil de resolver analíticamente, `NormalDist` puede generar muestras de entrada para una simulación utilizando el método Montecarlo:

```
>>> def model(x, y, z):
...     return (3*x + 7*x*y - 5*y) / (11 * z)
...
>>> n = 100_000
>>> X = NormalDist(10, 2.5).samples(n, seed=3652260728)
>>> Y = NormalDist(15, 1.75).samples(n, seed=4582495471)
>>> Z = NormalDist(50, 1.25).samples(n, seed=6582483453)
>>> quantiles(map(model, X, Y, Z))
[1.4591308524824727, 1.8035946855390597, 2.175091447274739]
```

Las distribuciones normales se pueden utilizar para aproximar distribuciones binomiales cuando el tamaño de la muestra es grande y la probabilidad de un ensayo exitoso es cercana al 50%.

Por ejemplo, 750 personas asisten a una conferencia sobre código abierto y se dispone de dos salas con capacidad para 500 personas cada una. En la primera sala hay una charla sobre Python, en la otra una sobre Ruby. En conferencias pasadas, el 65% de las personas prefirieron escuchar las charlas sobre Python. Suponiendo que las preferencias de la población no hayan cambiado, ¿cuál es la probabilidad de que la sala de Python permanezca por debajo de su capacidad máxima?

```
>>> n = 750                # Sample size
>>> p = 0.65               # Preference for Python
>>> q = 1.0 - p            # Preference for Ruby
>>> k = 500                # Room capacity

>>> # Approximation using the cumulative normal distribution
>>> from math import sqrt
>>> round(NormalDist(mu=n*p, sigma=sqrt(n*p*q)).cdf(k + 0.5), 4)
0.8402

>>> # Solution using the cumulative binomial distribution
>>> from math import comb, fsum
>>> round(fsum(comb(n, r) * p**r * q**(n-r) for r in range(k+1)), 4)
0.8402

>>> # Approximation using a simulation
>>> from random import seed, choices
>>> seed(8675309)
>>> def trial():
...     return choices(('Python', 'Ruby'), (p, q), k=n).count('Python')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> mean(trial() <= k for i in range(10_000))
0.8398
```

Las distribuciones normales a menudo están involucradas en el aprendizaje automático.

Wikipedia detalla un buen ejemplo de un [clasificador bayesiano ingenuo](#). El objetivo es predecir el género de una persona a partir de características físicas que siguen una distribución normal, como la altura, el peso y el tamaño del pie.

Disponemos de un conjunto de datos de entrenamiento que contiene las medidas de ocho personas. Se supone que estas medidas siguen una distribución normal, por lo que podemos sintetizar los datos usando *NormalDist*:

```
>>> height_male = NormalDist.from_samples([6, 5.92, 5.58, 5.92])
>>> height_female = NormalDist.from_samples([5, 5.5, 5.42, 5.75])
>>> weight_male = NormalDist.from_samples([180, 190, 170, 165])
>>> weight_female = NormalDist.from_samples([100, 150, 130, 150])
>>> foot_size_male = NormalDist.from_samples([12, 11, 12, 10])
>>> foot_size_female = NormalDist.from_samples([6, 8, 7, 9])
```

A continuación, nos encontramos con un nuevo individuo del que conocemos las medidas de sus características pero no su género:

```
>>> ht = 6.0          # height
>>> wt = 130          # weight
>>> fs = 8            # foot size
```

Partiendo de una [probabilidad a priori](#) del 50% de ser hombre o mujer, calculamos la probabilidad a posteriori como el producto de la probabilidad a priori y la verosimilitud de las diferentes medidas dado el género:

```
>>> prior_male = 0.5
>>> prior_female = 0.5
>>> posterior_male = (prior_male * height_male.pdf(ht) *
...                   weight_male.pdf(wt) * foot_size_male.pdf(fs))

>>> posterior_female = (prior_female * height_female.pdf(ht) *
...                     weight_female.pdf(wt) * foot_size_female.pdf(fs))
```

La predicción final es la que tiene mayor probabilidad a posteriori. Este enfoque se denomina [máximo a posteriori](#) o MAP:

```
>>> 'male' if posterior_male > posterior_female else 'female'
'female'
```

Módulos de programación funcional

Los módulos descritos en este capítulo proporcionan funciones y clases que admiten un estilo de programación funcional y operaciones generales invocables (*callable*s).

En este capítulo se documentan los siguientes módulos:

10.1 `itertools` — Funciones que crean iteradores para bucles eficientes

Este módulo implementa un número de piezas básicas *iterator* inspiradas en *constructs* de APL, Haskell y SML. Cada pieza ha sido reconvertida a una forma apropiada para Python.

El módulo estandariza un conjunto base de herramientas rápidas y eficientes en memoria, útiles por sí mismas o en combinación con otras. Juntas, forman un «álgebra de iteradores», haciendo posible la construcción de herramientas especializadas, sucintas y eficientes, en Python puro.

Por ejemplo, SML provee una herramienta de tabulación `tabulate(f)`, que produce una secuencia `f(0)`, `f(1)`, ... En Python, se puede lograr el mismo efecto al combinar `map()` y `count()` para formar `map(f, count())`.

Estas herramientas y sus contrapartes incorporadas también funcionan bien con funciones de alta velocidad del módulo *operator*. Por ejemplo, el operador de multiplicación se puede mapear a lo largo de dos vectores para formar un eficiente producto escalar: `sum(map(operator.mul, vector1, vector2))`.

Iteradores infinitos:

Iterador	Argumentos	Resultados	Ejemplo
<code>count()</code>	start, [step]	start, start+step, start+2*step, ...	<code>count(10)</code> --> 10 11 12 13 14 ...
<code>cycle()</code>	p	p0, p1, ... pfinal, p0, p1, ...	<code>cycle('ABCD')</code> --> A B C D A B C D ...
<code>repeat()</code>	elem [,n]	elem, elem, elem, ... indefinidamente o hasta n veces	<code>repeat(10, 3)</code> --> 10 10 10

Iteradores que terminan en la secuencia de entrada más corta:

Iterador	Argumentos	Resultados	Ejemplo
<code>accumulate()</code>	<code>p [,func]</code>	<code>p0, p0+p1, p0+p1+p2, ...</code>	<code>accumulate([1,2,3,4,5]) --> 1 3 6 10 15</code>
<code>chain()</code>	<code>p, q, ...</code>	<code>p0, p1, ... pfinal, q0, q1, ...</code>	<code>chain('ABC', 'DEF') --> A B C D E F</code>
<code>chain.from_iterable()</code>	iterable	<code>p0, p1, ... pfinal, q0, q1, ...</code>	<code>chain.from_iterable(['ABC', 'DEF']) --> A B C D E F</code>
<code>compress()</code>	<code>data, selectors</code>	<code>(d[0] if s[0]), (d[1] if s[1]), ...</code>	<code>compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F</code>
<code>dropwhile()</code>	<code>pred, seq</code>	<code>seq[n], seq[n+1], comenzando cuando pred falla</code>	<code>dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1</code>
<code>filterfalse()</code>	<code>pred, seq</code>	elementos de <code>seq</code> donde <code>pred(elem)</code> es falso	<code>filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8</code>
<code>groupby()</code>	iterable[, key]	sub-iteradores agrupados según el valor de <code>key(v)</code>	
<code>islice()</code>	<code>seq, [start,] stop [, step]</code>	elementos de <code>seq[start:stop:step]</code>	<code>islice('ABCDEFGH', 2, None) --> C D E F G</code>
<code>starmap()</code>	<code>func, seq</code>	<code>func(*seq[0]), func(*seq[1]), ...</code>	<code>starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000</code>
<code>takewhile()</code>	<code>pred, seq</code>	<code>seq[0], seq[1], hasta que pred falle</code>	<code>takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4</code>
<code>tee()</code>	<code>it, n</code>	<code>it1, it2, ... itn</code> divide un iterador en <code>n</code>	
<code>zip_longest()</code>	<code>p, q, ...</code>	<code>(p[0], q[0]), (p[1], q[1]), ...</code>	<code>zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-</code>

Iteradores combinatorios:

Iterador	Argumentos	Resultados
<code>product()</code>	<code>p, q, ... [repeat=1]</code>	producto cartesiano, equivalente a un bucle <i>for</i> anidado
<code>permutations()</code>	<code>p[, r]</code>	tuplas de longitud <code>r</code> , en todas los órdenes posibles, sin elementos repetidos
<code>combinations()</code>	<code>p, r</code>	tuplas de longitud <code>r</code> , ordenadas, sin elementos repetidos
<code>combinations_with_replacement()</code>	<code>p, r</code>	tuplas de longitud <code>r</code> , ordenadas, con elementos repetidos

Ejemplos	Resultados
<code>product('ABCD', repeat=2)</code>	AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
<code>permutations('ABCD', 2)</code>	AB AC AD BA BC BD CA CB CD DA DB DC
<code>combinations('ABCD', 2)</code>	AB AC AD BC BD CD
<code>combinations_with_replacement('ABCD', 2)</code>	AA AB AC AD BB BC BD CC CD DD

10.1.1 Funciones de itertools

Todas las funciones del siguiente módulo construyen y retornan iteradores. Algunas proveen flujos infinitos, por lo que deberían ser sólo manipuladas por funciones o bucles que cortan el flujo.

`itertools.accumulate(iterable[, func, *, initial=None])`

Crea un iterador que retorna sumas acumuladas o resultados acumulados de otra función binaria (especificada a través del argumento opcional *func*).

Si *func* es definido, debería ser una función de 2 argumentos. Los elementos de entrada de *iterable* pueden ser de cualquier tipo que puedan ser aceptados como argumentos de *func*. (Por ejemplo, con la operación por defecto –adición, los elementos pueden ser cualquier tipo que sea adicionable, incluyendo *Decimal* o *Fraction*.)

Usualmente el número de elementos de salida corresponde con el número de elementos del iterador de entrada. Sin embargo, si el argumento clave *initial* es suministrado, la acumulación empieza con *initial* como valor inicial y el resultado contiene un elemento más que el iterador de entrada.

Aproximadamente equivalente a:

```
def accumulate(iterable, func=operator.add, *, initial=None):
    'Return running totals'
    # accumulate([1,2,3,4,5]) --> 1 3 6 10 15
    # accumulate([1,2,3,4,5], initial=100) --> 100 101 103 106 110 115
    # accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120
    it = iter(iterable)
    total = initial
    if initial is None:
        try:
            total = next(it)
        except StopIteration:
            return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

Hay un número de usos para el argumento *func*. Se le puede asignar *min()* para calcular un mínimo acumulado, *max()* para un máximo acumulado, o *operator.mul()* para un producto acumulado. Se pueden crear tablas de amortización al acumular intereses y aplicando pagos. [Relaciones de recurrencias de primer orden](#) se puede modelar al proveer el valor inicial en el iterable y utilizando sólo el total acumulado en el argumento *func*:

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, operator.mul))          # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
>>> list(accumulate(data, max))                  # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]

# Amortize a 5% loan of 1000 with 4 annual payments of 90
>>> cashflows = [1000, -90, -90, -90, -90]
>>> list(accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt))
[1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]

# Chaotic recurrence relation https://en.wikipedia.org/wiki/Logistic_map
>>> logistic_map = lambda x, _: r * x * (1 - x)
>>> r = 3.8
>>> x0 = 0.4
>>> inputs = repeat(x0, 36)                      # only the initial value is used
>>> [format(x, '.2f') for x in accumulate(inputs, logistic_map)]
['0.40', '0.91', '0.30', '0.81', '0.60', '0.92', '0.29', '0.79', '0.63',
 '0.88', '0.39', '0.90', '0.33', '0.84', '0.52', '0.95', '0.18', '0.57',
```

(continué en la próxima página)

(proviene de la página anterior)

```
'0.93', '0.25', '0.71', '0.79', '0.63', '0.88', '0.39', '0.91', '0.32',  
'0.83', '0.54', '0.95', '0.20', '0.60', '0.91', '0.30', '0.80', '0.60']
```

Para una función similar que retorne únicamente el valor final acumulado, revisa `functools.reduce()`.

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Adicionó el argumento opcional *func*.

Distinto en la versión 3.8: Adicionó el argumento opcional *initial*.

`itertools.chain(*iterables)`

Crea un iterador que retorna elementos del primer iterable hasta que es consumido, para luego proceder con el siguiente iterable, hasta que todos los iterables son consumidos. Se utiliza para tratar secuencias consecutivas como una sola secuencia. Aproximadamente equivalente a:

```
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)`

Constructor alternativo para `chain()`. Obtiene entradas enlazadas de un mismo argumento que se evalúa perezosamente. Aproximadamente equivalente a:

```
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
```

`itertools.combinations(iterable, r)`

Retorna subsecuencias de longitud *r* con elementos del *iterable* de entrada.

The combination tuples are emitted in lexicographic ordering according to the order of the input *iterable*. So, if the input *iterable* is sorted, the combination tuples will be produced in sorted order.

Los elementos son tratados como únicos basados en su posición, no en su valor. De esta manera, si los elementos de entrada son únicos, no habrá valores repetidos en cada combinación.

Aproximadamente equivalente a:

```
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 = list(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)
```


El código para `combinations()` se puede expresar también como una subsecuencia de `permutations()`, luego de filtrar entradas donde los elementos no están ordenados (de acuerdo a su posición en el conjunto de entrada):

```
def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in permutations(range(n), r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

El número de elementos retornados es $n! / r! / (n-r)!$ cuando $0 \leq r \leq n$ o cero cuando $r > n$.

`itertools.combinations_with_replacement(iterable, r)`

Retorna subsecuencias, de longitud *r*, con elementos del *iterable* de entrada, permitiendo que haya elementos individuales repetidos más de una vez.

The combination tuples are emitted in lexicographic ordering according to the order of the input *iterable*. So, if the input *iterable* is sorted, the combination tuples will be produced in sorted order.

Los elementos son tratados como únicos basados en su posición, no en su valor. De esta manera, si los elementos de entrada son únicos, las combinaciones generadas también serán únicas.

Aproximadamente equivalente a:

```
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)
```

El código para `combinations_with_replacement()` se puede expresar también como una subsecuencia de `product()`, luego de filtrar entradas donde los elementos no están ordenados (de acuerdo a su posición en el conjunto de entrada):

```
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)
```

El número de elementos retornados es $(n+r-1)! / r! / (n-1)!$ cuando $n > 0$.

Nuevo en la versión 3.1.

`itertools.compress(data, selectors)`

Crea un iterador que filtra elementos de *data*, retornando sólo aquellos que tienen un elemento correspondiente en *selectors* que evalúa a True. El iterador se detiene cuando alguno de los iterables (*data* o *selectors*) ha sido consumido. Aproximadamente equivalente a:

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in zip(data, selectors) if s)
```

Nuevo en la versión 3.1.

`itertools.count` (*start=0, step=1*)

Crea un iterador que retorna valores espaciados uniformemente, comenzando con el número *start*. Usualmente se utiliza como argumento en `map()` para generar puntos de datos consecutivos. También se utiliza en `zip()` para agregar secuencias de números. Aproximadamente equivalente a:

```
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
```

Cuando se hace conteo con números de punto flotante, se puede lograr una mejor precisión al sustituir código multiplicativo como: `(start + step * i for i in count())`.

Distinto en la versión 3.1: Se adicionó el argumento *step* y se permitieron argumentos diferentes a enteros.

`itertools.cycle` (*iterable*)

Crea un iterador que retorna elementos del iterable y hace una copia de cada uno. Cuando el iterable es consumido, retornar los elementos de la copia almacenada. Se repite indefinidamente. Aproximadamente equivalente a:

```
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
```

Ten en cuenta, este miembro del conjunto de herramientas puede requerir almacenamiento auxiliar importante (dependiendo de la longitud del iterable).

`itertools.dropwhile` (*predicate, iterable*)

Crea un iterador que descarta elementos del iterable, siempre y cuando el predicado sea verdadero; después, retorna cada elemento. Ten en cuenta, el iterador no produce *ningún* resultado hasta que el predicado se hace falso, pudiendo incurrir en un tiempo de arranque extenso. Aproximadamente equivalente a:

```
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.filterfalse` (*predicate, iterable*)

Crea un iterador que filtra elementos de un iterable, retornando sólo aquellos para los cuales el predicado es False. Si *predicate* es None, retorna los elementos que son falsos. Aproximadamente equivalente a:

```
def filterfalse(predicate, iterable):
    # filterfalse(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.groupby(iterable, key=None)`

Crea un iterador que retorna claves consecutivas y grupos del *iterable*. *key* es una función que calcula un valor clave para cada elemento. Si no se especifica o es `None`, *key* es una función de identidad por defecto y retorna el elemento sin cambios. Generalmente, el iterable necesita estar ordenado con la misma función *key*.

El funcionamiento de `groupby()` es similar al del filtro `uniq` en Unix. Genera un salto o un nuevo grupo cada vez que el valor de la función clave cambia (por lo que usualmente es necesario ordenar los datos usando la misma función clave). Ese comportamiento difiere del de GROUP BY de SQL, el cual agrega elementos comunes sin importar el orden de entrada.

El grupo retornado es un iterador mismo que comparte el iterable subyacente con `groupby()`. Al compartir la fuente, cuando el objeto `groupby()` se avanza, el grupo previo deja de ser visible. En ese caso, si los datos se necesitan posteriormente, se deberían almacenar como lista:

```
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()` es aproximadamente equivalente a:

```
class groupby:
    # [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):
        self.id = object()
        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, self.id))
    def _grouper(self, tgtkey, id):
        while self.id is id and self.currkey == tgtkey:
            yield self.currvalue
            try:
                self.currvalue = next(self.it)
            except StopIteration:
                return
            self.currkey = self.keyfunc(self.currvalue)
```

`itertools.islice(iterable, stop)`

`itertools.islice(iterable, start, stop[, step])`

Crea un iterador que retorna los elementos seleccionados del iterable. Si *start* es diferente a cero, los elementos del iterable son ignorados hasta que se llegue a *start*. Después de eso, los elementos son retornados consecuti-

vamente a menos que *step* posea un valor tan alto que permita que algunos elementos sean ignorados. Si *stop* es *None*, la iteración continúa hasta que el iterador sea consumido (si es que llega a ocurrir); de lo contrario, se detiene en la posición especificada. A diferencia de la segmentación normal, *islice()* no soporta valores negativos para *start*, *stop*, o *step*. Puede usarse para extraer campos relacionados de estructuras de datos que internamente has sido simplificadas (por ejemplo, un reporte multi-línea puede contener un nombre de campo cada tres líneas). Aproximadamente equivalente a:

```
def islice(iterable, *args):
    # islice('ABCDEFGH', 2) --> A B
    # islice('ABCDEFGH', 2, 4) --> C D
    # islice('ABCDEFGH', 2, None) --> C D E F G
    # islice('ABCDEFGH', 0, None, 2) --> A C E G
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop or sys.maxsize, s.step or 1
    it = iter(range(start, stop, step))
    try:
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in zip(range(start), iterable):
            pass
        return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
                nexti = next(it)
    except StopIteration:
        # Consume to *stop*.
        for i, element in zip(range(i + 1, stop), iterable):
            pass
```

Si *start* es *None*, la iteración empieza en cero. Si *step* es *None*, *step* se establece en uno por defecto.

`itertools.permutations(iterable, r=None)`

Retorna permutaciones de elementos sucesivas de longitud *r* en el *iterable*.

Si *r* no es especificado o si es *None*, entonces por defecto *r* será igual a la longitud de *iterable* y todas las permutaciones de máxima longitud serán generadas.

The permutation tuples are emitted in lexicographic ordering according to the order of the input *iterable*. So, if the input *iterable* is sorted, the combination tuples will be produced in sorted order.

Los elementos son tratados como únicos según su posición, y no su valor. Por ende, no habrá elementos repetidos en cada permutación si los elementos de entrada son únicos.

Aproximadamente equivalente a:

```
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 = list(range(n))
    cycles = list(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]
```

(continué en la próxima página)

(proviene de la página anterior)

```

        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

```

El código para `permutations()` también se puede expresar como una subsecuencia de `product()`, filtrado para excluir registros con elementos repetidos (aquellos en la misma posición que en el conjunto de entrada):

```

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)

```

El número de elementos retornados es $n! / (n-r)!$ cuando $0 \leq r \leq n$ o cero cuando $r > n$.

`itertools.product(*iterables, repeat=1)`

Producto cartesiano de los iterables de entrada.

Aproximadamente equivalente a tener bucles *for* anidados en un generador. Por ejemplo, `product(A, B)` es equivalente a `((x,y) for x in A for y in B)`.

Los bucles anidados hacen ciclos como un cuentapaseos o taxímetro, con el elemento más hacia la derecha avanzando en cada iteración. Este patrón crea un orden lexicográfico en el que, si los iterables de entrada están ordenados, las tuplas producidas son emitidas de manera ordenada.

Para calcular el producto de un iterable consigo mismo, especifica el número de repeticiones con el argumento opcional `repeat`. Por ejemplo, `product(A, repeat=4)` es equivalente a `product(A, A, A, A)`.

Esta función es aproximadamente equivalente al código siguiente, exceptuando que la implementación real no acumula resultados intermedios en memoria:

```

def product(*args, repeat=1):
    # 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 = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)

```

`itertools.repeat(object[, times])`

Crea un iterador que retorna *object* una y otra vez. Se ejecuta indefinidamente a menos que se especifique el argumento *times*. Se utiliza como argumento de `map()` para argumentos invariantes de la función invocada. También se usa con `zip()` para crear una parte invariante de una tupla.

Aproximadamente equivalente a:

```

def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:

```

(continué en la próxima página)

(proviene de la página anterior)

```
for i in range(times):
    yield object
```

Un uso común de *repeat* es el de proporcionar un flujo de valores constantes a *map* o *zip*:

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`itertools.starmap` (*function, iterable*)

Crea un iterador que calcula la función utilizando argumentos obtenidos del iterable. Se usa en lugar de *map()* cuando los argumentos ya están agrupados en tuplas de un mismo iterable (los datos ya han sido «pre-comprimidos»). La diferencia entre *map()* y *starmap()* es similar a la distinción entre *function(a, b)* y *function(*c)*. Aproximadamente equivalente a:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

`itertools.takewhile` (*predicate, iterable*)

Crea un iterador que retorna elementos del iterador siempre y cuando el predicado sea cierto. Aproximadamente equivalente a:

```
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*)

Retorna *n* iteradores independientes de un mismo iterador.

El código Python a continuación ayuda a explicar el funcionamiento de *tee* (aunque la implementación real es mucho más compleja y usa sólo una cola FIFO subyacente).

Aproximadamente equivalente a:

```
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
                try:
                    newval = next(it)         # fetch a new value and
                except StopIteration:
                    return
            for d in deques:                  # load it to all the deques
                d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

Una vez que *tee()* ha hecho un corte, el *iterable* original no se debería usar en otro lugar. De lo contrario, el *iterable* podría avanzarse sin informar a los objetos *tee*.

Los iteradores *tee* no son *threadsafe*. *RuntimeError* puede ocurrir si se usan simultáneamente iteradores retornados por la misma llamada a *tee()* call, aún cuando el *iterable* original sea *threadsafe*.

Esta herramienta de iteración puede requerir almacenamiento auxiliar significativo (dependiendo de qué tantos datos necesitan ser almacenados). En general, si un iterador utiliza todos o la mayoría de los datos antes que

otro iterador comience, es más rápido utilizar `list()` en vez de `tee()`.

`itertools.zip_longest(*iterables, fillvalue=None)`

Crea un iterador que agrega elementos de cada uno de los iterables. Si los iterables tiene longitud impar, los valores sin encontrar serán iguales a *fillvalue*. La iteración continúa hasta que el iterable más largo sea consumido. Aproximadamente equivalente a:

```
def zip_longest(*args, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    iterators = [iter(it) for it in args]
    num_active = len(iterators)
    if not num_active:
        return
    while True:
        values = []
        for i, it in enumerate(iterators):
            try:
                value = next(it)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```

Si alguno de los iterables es potencialmente infinito, la función `zip_longest()` debería ser recubierta por otra que limite el número de llamadas (por ejemplo, `islice()` o `takewhile()`). Si no se especifica, *fillvalue* es `None` por defecto.

10.1.2 Fórmulas con `itertools`

Esta sección muestra fórmulas para crear un conjunto de herramientas extendido usando las herramientas de `itertools` como piezas básicas.

De manera considerable, todas estas fórmulas y muchos otras se pueden instalar desde el [proyecto more-itertools](#), ubicado en el Python Package Index:

```
pip install more-itertools
```

Las herramientas adicionales ofrecen el mismo alto rendimiento que las herramientas subyacentes. El rendimiento de memoria superior se mantiene al procesar los elementos uno a uno, y no cargando el iterable entero en memoria. El volumen de código se mantiene bajo al enlazar las herramientas en estilo funcional, eliminando variables temporales. La alta velocidad se retiene al preferir piezas «vectorizadas» sobre el uso de bucles `for` y *generators* que puedan incurrir en costos extra.

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def prepend(value, iterator):
    "Prepend a single value in front of an iterator"
    # prepend(1, [2, 3, 4]) -> 1 2 3 4
    return chain([value], iterator)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def tail(n, iterable):
    "Return last n items of the iterable as a list"
    # tail(3, [0, 1, 2, 3, 4, 5]) -> 3 4 5
    # tail(0, ...) -> []
    # tail(1, empty) -> []
    it = iter(iterable)
    if n == 0:
        return []
    if n == 1:
        return [next(it)]
    # ...
    return list(it)
```

(continué en la próxima página)

(proviene de la página anterior)

```

"Return an iterator over the last n items"
# tail(3, 'ABCDEFGG') --> E F G
return iter(collections.deque(iterable, maxlen=n))

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)
    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(map(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(map(operator.mul, vec1, vec2))

def flatten(list_of_lists):
    "Flatten one level of nesting"
    return chain.from_iterable(list_of_lists)

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 zip(a, b)

def grouper(iterable, n, fillvalue=None):

```

(continué en la próxima página)

(proviene de la página anterior)

```

"Collect data into fixed-length chunks or blocks"
# grouper('ABCDEFGF', 3, 'x') --> ABC DEF Gxx"
args = [iter(iterable)] * n
return zip_longest(*args, fillvalue=fillvalue)

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    num_active = len(iterables)
    nexts = cycle(iter(it).__next__ for it in iterables)
    while num_active:
        try:
            for next in nexts:
                yield next()
        except StopIteration:
            # Remove the iterator we just exhausted from the cycle.
            num_active -= 1
            nexts = cycle(islice(nexts, num_active))

def partition(pred, iterable):
    "Use a predicate to partition entries into false entries and true entries"
    # partition(is_odd, range(10)) --> 0 2 4 6 8 and 1 3 5 7 9
    t1, t2 = tee(iterable)
    return filterfalse(pred, t1), filter(pred, t2)

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 filterfalse(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 map(next, map(operator.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 builtins.iter(func, sentinel) but uses an exception instead
    of a sentinel to end the loop.

    Examples:

```

(continué en la próxima página)

(proviene de la página anterior)

```

        iter_except(functools.partial(heapop, h), IndexError) # priority queue
↪ iterator
        iter_except(d.popitem, KeyError) # non-blocking
↪ dict iterator
        iter_except(d.popleft, IndexError) # non-blocking
↪ deque iterator
        iter_except(q.get_nowait, Queue.Empty) # loop over a
↪ producer Queue
        iter_except(s.pop, KeyError) # non-blocking
↪ set iterator

    """
    try:
        if first is not None:
            yield first() # For database APIs needing an initial cast
↪ to db.first()
        while True:
            yield func()
    except exception:
        pass

def first_true(iterable, default=False, pred=None):
    """Returns the first true value in the iterable.

    If no true value is found, returns *default*

    If *pred* is not None, returns the first item
    for which pred(item) is true.

    """
    # first_true([a,b,c], x) --> a or b or c or x
    # first_true([a,b], x, f) --> a if f(a) else b if f(b) else x
    return next(filter(pred, iterable), default)

def random_product(*args, repeat=1):
    "Random selection from itertools.product(*args, **kwargs)"
    pools = [tuple(pool) for pool in args] * repeat
    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(range(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 range(r))
    return tuple(pool[i] for i in indices)

def nth_combination(iterable, r, index):
    'Equivalent to list(combinations(iterable, r))[index]'
```

(continué en la próxima página)

(proviene de la página anterior)

```

pool = tuple(iterable)
n = len(pool)
if r < 0 or r > n:
    raise ValueError
c = 1
k = min(r, n-r)
for i in range(1, k+1):
    c = c * (n - k + i) // i
if index < 0:
    index += c
if index < 0 or index >= c:
    raise IndexError
result = []
while r:
    c, n, r = c*r//n, n-1, r-1
    while index >= c:
        index -= c
        c, n = c*(n-r)//n, n-1
    result.append(pool[-1-n])
return tuple(result)

```

10.2 `functools` — Funciones de orden superior y operaciones sobre objetos invocables

Código fuente: [Lib/functools.py](#)

El módulo `functools` es para funciones de orden superior: funciones que actúan o retornan otras funciones. En general, cualquier objeto invocable puede ser tratado como una función para los propósitos de este módulo.

El módulo `functools` define las siguientes funciones:

`@functools.cached_property(func)`

Transforma un método de una clase en una propiedad cuyo valor se computa una vez y luego se almacena como un atributo normal durante la vida de la instancia. Similar a `property()`, con la adición de caching. Útil para propiedades calculadas costosas de instancias que de otra manera son efectivamente inmutables.

Ejemplo:

```

class DataSet:
    def __init__(self, sequence_of_numbers):
        self._data = sequence_of_numbers

    @cached_property
    def stdev(self):
        return statistics.stdev(self._data)

    @cached_property
    def variance(self):
        return statistics.variance(self._data)

```

Nuevo en la versión 3.8.

Nota: Este decorador requiere que el atributo `__dict__` en cada instancia sea un mapeo mutable. Esto significa que no funcionará con algunos tipos, como las metaclasses (ya que los atributos `__dict__` en las instancias de tipo son proxies sólo de lectura para el espacio de nombres de la clase), y los que especifican

`__slots__` sin incluir `__dict__` como uno de los slots definidos (ya que tales clases no proporcionan un atributo `__dict__` en absoluto).

`functools.cmp_to_key(func)`

Transformar una función de comparación de estilo antiguo en una *key function*. Se utiliza con herramientas que aceptan funciones clave (como `sorted()`, `min()`, `max()`, `heapq.nlargest()`, `heapq.nsmallest()`, `itertools.groupby()`). Esta función se utiliza principalmente como una herramienta de transición para los programas que se están convirtiendo a partir de Python 2, que soportaba el uso de funciones de comparación.

Una función de comparación es cualquier invocable que acepta dos argumentos, los compara y retorna un número negativo para diferencia, cero para igualdad o un número positivo para más. Una función clave es un invocable que acepta un argumento y retorna otro valor para ser usado como clave de ordenación.

Ejemplo:

```
sorted(iterable, key=cmp_to_key(locale.strcoll)) # locale-aware sort order
```

Para ejemplos de clasificación y un breve tutorial de clasificación, ver [sortinghowto](#).

Nuevo en la versión 3.2.

`@functools.lru_cache(user_function)`

`@functools.lru_cache(maxsize=128, typed=False)`

Decorador para envolver una función con un memorizador invocable que guarda hasta el *maxsize* de las llamadas más recientes. Puede salvar el tiempo cuando una función costosa o de E/S es llamada periódicamente con los mismos argumentos.

Dado que se utiliza un diccionario para guardar los resultados, los argumentos posicionales y de las palabras clave de la función deben ser hashable.

Los patrones de argumento distintos pueden considerarse como llamadas distintas con entradas de caché separadas. Por ejemplo, $f(a=1, b=2)$ y $f(b=2, a=1)$ difieren en el orden de los argumentos de las palabras clave y pueden tener dos entradas de caché separadas.

Si se especifica *user_function*, debe ser una llamada. Esto permite que el decorador *lru_cache* se aplique directamente a una función de usuario, dejando el *maxsize* en su valor por defecto de 128:

```
@lru_cache
def count_vowels(sentence):
    sentence = sentence.casefold()
    return sum(sentence.count(vowel) for vowel in 'aeiou')
```

Si *maxsize* está configurado como `None`, la función LRU está desactivada y la caché puede crecer sin límites.

Si *typed* se establece como verdadero, los argumentos de las funciones de diferentes tipos se almacenarán en la memoria caché por separado. Por ejemplo, $f(3)$ y $f(3.0)$ se tratarán como llamadas distintas con resultados distintos.

Para ayudar a medir la efectividad del cache y afinar el parámetro *maxsize*, la función envolvente está instrumentada con una función `cache_info()` que retorna un *named tuple* mostrando *hits*, *misses*, *maxsize* y *currsize*. En un entorno multi-hilo, los aciertos y los fallos son aproximados.

El decorador también proporciona una función `cache_clear()` para limpiar o invalidar el caché.

La función subyacente original es accesible a través del atributo `__wrapped__`. Esto es útil para la introspección, para evitar el caché, o para volver a envolver la función con un caché diferente.

Una caché LRU (la *menos usada recientemente*) funciona mejor cuando las llamadas más recientes son los mejores pronosticadores de las próximas llamadas (por ejemplo, los artículos más populares en un servidor de noticias tienden a cambiar cada día). El límite de tamaño de la caché asegura que ésta no crezca sin estar vinculada a procesos de larga duración como los servidores web.

En general, la caché de la LRU sólo debe utilizarse cuando se desea reutilizar valores previamente calculados. Por consiguiente, no tiene sentido almacenar en la caché funciones con efectos secundarios, funciones que necesitan crear distintos objetos mutables en cada llamada, o funciones impuras como `time()` o `random()`.

Ejemplo de un caché de la LRU para contenido web estático:

```
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = 'http://www.python.org/dev/peps/pep-%04d/' % num
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, cursize=8)
```

Ejemplo de computar eficientemente los «números de Fibonacci» <https://en.wikipedia.org/wiki/Fibonacci_number>_ usando un cache para implementar una «programación dinámica» <https://en.wikipedia.org/wiki/Dynamic_programming>_ técnica:

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, cursize=16)
```

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Añadida la opción *typed* option.

Distinto en la versión 3.8: Añadida la opción *user_function*.

@functools.total_ordering

Dada una clase que define uno o más métodos de ordenamiento de comparación ricos, este decorador de clase suministra el resto. Esto simplifica el esfuerzo de especificar todas las posibles operaciones de comparación rica:

La clase debe definir uno de `__lt__()`, `__le__()`, `__gt__()`, o `__ge__()`. Además, la clase debe suministrar un método `__eq__()`. Dada una clase que define uno o más métodos de ordenamiento de comparación ricos, este decorador de clase suministra el resto. Esto simplifica el esfuerzo de especificar todas las posibles operaciones de comparación rica.

Por ejemplo:

```
@total_ordering
class Student:
    def __is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self.__is_valid_operand(other):
```

(continué en la próxima página)

(proviene de la página anterior)

```

        return NotImplemented
    return ((self.lastname.lower(), self.firstname.lower()) ==
            (other.lastname.lower(), other.firstname.lower()))
def __lt__(self, other):
    if not self._is_valid_operand(other):
        return NotImplemented
    return ((self.lastname.lower(), self.firstname.lower()) <
            (other.lastname.lower(), other.firstname.lower()))

```

Nota: Mientras que este decorador facilita la creación de tipos bien comportados y totalmente ordenados, *does* a costa de una ejecución más lenta y de trazos de pila (*stack traces*) más complejos para los métodos de comparación derivados. Si la evaluación comparativa del rendimiento indica que se trata de un cuello de botella para una aplicación determinada, la aplicación de los seis métodos de comparación ricos en su lugar es probable que proporcione un fácil aumento de la velocidad.

Nuevo en la versión 3.2.

Distinto en la versión 3.4: Retornando `NotImplemented` de la función de comparación subyacente para los tipos no reconocidos está ahora soportado.

`functools.partial(func, /, *args, **keywords)`

Retorna un nuevo *partial object* que cuando sea llamado se comportará como *func* llamado con los argumentos posicionales *args* y los argumentos de palabras clave *keywords*. Si se suministran más argumentos a la llamada, se añaden a *args*. Si se suministran más argumentos de palabras clave, se extienden y anulan las *keywords*. Aproximadamente equivalente a:

```

def partial(func, /, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = {**keywords, **fkeywords}
        return func(*args, *fargs, **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc

```

El *partial()* se utiliza para la aplicación de funciones parciales que «congela» (*freezes*) alguna porción de los argumentos y/o palabras clave de una función dando como resultado un nuevo objeto con una firma simplificada. Por ejemplo, *partial()* puede usarse para crear una llamada que se comporte como la función *int()* donde el argumento *base* tiene un valor por defecto de dos:

```

>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18

```

class `functools.partialmethod(func, /, *args, **keywords)`

Retorna un nuevo descriptor *partialmethod* que se comporta como *partial* excepto que está diseñado para ser usado como una definición de método en lugar de ser directamente invocable.

func debe ser un *descriptor* o un invocable (los objetos que son ambos, como las funciones normales, se manejan como descriptores).

Cuando *func* es un descriptor (como una función Python normal, *classmethod()*, *staticmethod()*, *abstractmethod()* u otra instancia de *partialmethod*), las llamadas a `__get__` se delegan al descriptor subyacente, y se retorna un *partial object* apropiado como resultado.

Cuando *func* es una llamada no descriptiva, se crea dinámicamente un método de unión apropiado. Esto se comporta como una función Python normal cuando se usa como método: el argumento *self* se insertará como el primer argumento posicional, incluso antes de las *args* y *keywords* suministradas al constructor

partialmethod.

Ejemplo:

```
>>> class Cell(object):
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         return self._alive
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)
...
>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True
```

Nuevo en la versión 3.4.

`functools.reduce(function, iterable[, initializer])`

Aplicar una *función* de dos argumentos acumulativos a los elementos de *iterable*, de izquierda a derecha, para reducir los itables a un solo valor. Por ejemplo, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calcula `((((1+2)+3)+4)+5)`. El argumento de la izquierda, *x*, es el valor acumulado y el de la derecha, *y*, es el valor de actualización del *iterable*. Si el *initializer* opcional está presente, se coloca antes de los ítems de la *iterable* en el cálculo, y sirve como predeterminado cuando la *iterable* está vacía. Si no se da el *initializer* y el *iterable* contiene sólo un elemento, se retorna el primer elemento.

Aproximadamente equivalente a:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

Ver `itertools.accumulate()` para un iterador que produce todos los valores intermedios.

`@functools.singledispatch`

Transformar una función en una *single-dispatch generic function*.

Para definir la función genérica, decórela con el decorador `@singledispatch`. Ten en cuenta que el envío ocurre en el tipo del primer argumento, crea tu función en consecuencia:

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
```

Para añadir implementaciones sobrecargadas a la función, use el atributo `register()` de la función genérica. Es un decorador. Para las funciones anotadas con tipos, el decorador deducirá automáticamente el tipo del primer argumento:

```
>>> @fun.register
... def _(arg: int, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> @fun.register
... def _(arg: list, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

Para el código que no utiliza anotaciones de tipo, el argumento de tipo apropiado puede ser pasado explícitamente al propio decorador:

```
>>> @fun.register(complex)
... def _(arg, verbose=False):
...     if verbose:
...         print("Better than complicated.", end=" ")
...     print(arg.real, arg.imag)
...
>>>
```

Para permitir el registro de lambdas y funciones preexistentes, el atributo `register()` puede utilizarse de forma funcional:

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

El atributo `register()` retorna la función no decorada que permite al decorador apilar, decapar, así como crear pruebas de unidad para cada variante de forma independiente:

```
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
>>> fun_num is fun
False
```

Cuando se llama, la función genérica despacha sobre el tipo del primer argumento:

```
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615
```


Cuando no hay una implementación registrada para un tipo específico, su orden de resolución de método se utiliza para encontrar una implementación más genérica. La función original decorada con `@singledispatch` se registra para el tipo de `object` base, lo que significa que se usa si no se encuentra una mejor implementación.

Para comprobar qué implementación elegirá la función genérica para un tipo determinado, utilice el atributo `dispatch()`:

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)      # note: default implementation
<function fun at 0x103fe0000>
```

Para acceder a todas las implementaciones registradas, utilice el atributo `registry` de sólo lectura:

```
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>,
          <class 'decimal.Decimal'>, <class 'list'>,
          <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

Nuevo en la versión 3.4.

Distinto en la versión 3.7: El atributo `register()` soporta el uso de anotaciones de tipo.

class `functools.singledispatchmethod` (*func*)
Transformar un método en un *single-dispatch generic function*.

Para definir un método genérico, decóralo con el decorador `@singledispatchmethod`. Tenga en cuenta que el envío se produce en el tipo del primer argumento que no sea un atributo de instancias (*non-self*) ni un atributo de clases (*non-cls*), cree su función en consecuencia:

```
class Negator:
    @singledispatchmethod
    def neg(self, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    def _(self, arg: int):
        return -arg

    @neg.register
    def _(self, arg: bool):
        return not arg
```

El `@singledispatchmethod` apoya el anidamiento con otros decoradores como el `@classmethod`. Ten en cuenta que para permitir el `dispatcher.register`, `singledispatchmethod` debe ser el decorador *outer most*. Aquí está la clase `neg` con los métodos `Negator` limitados a la clase:

```
class Negator:
    @singledispatchmethod
    @classmethod
    def neg(cls, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    @classmethod
    def _(cls, arg: int):
        return -arg
```

(continué en la próxima página)

(proviene de la página anterior)

```

@neg.register
@classmethod
def _(cls, arg: bool):
    return not arg

```

El mismo patrón puede ser usado para otros decoradores similares: `staticmethod`, `abstractmethod`, y otros.

Nuevo en la versión 3.8.

`functools.update_wrapper(wrapper, wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)`

Actualizar una función envoltorio (*wrapper*) para que se parezca a la función de envoltura (*wrapped*). Los argumentos opcionales son tuplas para especificar qué atributos de la función original se asignan directamente a los atributos correspondientes de la función de envoltura y qué atributos de la función de envoltura se actualizan con los atributos correspondientes de la función original. Los valores por defecto de estos argumentos son las constantes de nivel de módulo `WRAPPER_ASSIGNMENTS` (que asigna al `__module__` de la función de envoltura, `__module__`, `__name__`, `__qualname__`, `__annotations__` y `__doc__`, la cadena de documentación) y `WRAPPER_UPDATES` (que actualiza el `__dict__` de la función de envoltura (*wrapped*), i. e. el diccionario de instancias).

Para permitir el acceso a la función original para la introspección y otros propósitos (por ejemplo, evitando un decorador de caché como `lru_cache()`), esta función añade automáticamente un atributo `__wrapped__` al envoltorio que se refiere a la función que se está envolviendo.

El principal uso previsto para esta función es en *decorator* functions que envuelven la función decorada y retornan el envoltorio. Si la función de envoltura no se actualiza, los metadatos de la función retornada reflejarán la definición de la envoltura en lugar de la definición de la función original, lo que normalmente no es de gran ayuda.

`update_wrapper()` puede ser usado con otros invocables que no sean funciones. Cualquier atributo nombrado en *assigned* o *updated* que falte en el objeto que se está invoca se ignora (es decir, esta función no intentará establecerlos en la función de envoltura (*wrapper*)). `AttributeError` sigue apareciendo si la propia función de envoltura no tiene ningún atributo nombrado en *updated*.

Nuevo en la versión 3.2: Adición automática de `__wrapped__` attribute.

Nuevo en la versión 3.2: Copia del atributo `__annotations__` por defecto.

Distinto en la versión 3.2: Los atributos faltantes ya no desencadenan un `AtributoError`.

Distinto en la versión 3.4: El atributo `__wrapped__` ahora siempre se refiere a la función envuelta, incluso si esa función definió un atributo `__wrapped__`. (see :issue:17482)

`@functools.wraps(wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)`

Esta es una función conveniente para invocar `update_wrapper()` como decorador de la función cuando se define una función de envoltura (*wrapper*). Es equivalente a `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)`. Por ejemplo:

```

>>> 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()

```

(continué en la próxima página)

(proviene de la página anterior)

```

Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'

```

Sin el uso de esta fábrica de decoradores, el nombre de la función de ejemplo habría sido `'wrapper'`, y el docstring de la `example()` se habría perdido.

10.2.1 `partial` Objetos

Los objetos `partial` son objetos invocables creados por `partial()`. Tienen tres atributos de sólo lectura:

`partial.func`

Un objeto o función invocable. Las llamadas al objeto `partial` serán reenviadas a `func` con nuevos argumentos y palabras clave.

`partial.args`

Los argumentos posicionales de la izquierda que se prepararán para los argumentos posicionales proporcionados un llamado al objeto `partial`.

`partial.keywords`

Los argumentos de la palabra clave que se suministrarán cuando se llame al objeto `partial`.

Los objetos `partial` son como los objetos `function` que son invocables, de referencia débil y pueden tener atributos. Hay algunas diferencias importantes. Por ejemplo, los atributos `__name__` y `__doc__` no se crean automáticamente. Además, los objetos `partial` definidos en las clases se comportan como métodos estáticos y no se transforman en métodos vinculados durante la búsqueda de atributos de la instancia.

10.3 `operator` — Operadores estándar como funciones

Código fuente: [Lib/operator.py](#)

El módulo `operator` exporta un conjunto de funciones eficientes correspondientes a los operadores intrínsecos de Python. Por ejemplo, `operator.add(x, y)` es equivalente a la expresión `x+y`. Muchos nombres de función son los utilizados para métodos especiales, sin los dobles guiones bajos. Por compatibilidad con versiones anteriores, muchos de estos tienen una variante que conserva los dobles guiones bajos. Se prefieren las variantes sin los dobles guiones bajos para mayor claridad.

Las funciones se dividen en categorías que realizan comparaciones de objetos, operaciones lógicas, operaciones matemáticas y operaciones sobre secuencias.

Las funciones de comparación de objetos son útiles para todos los objetos, y llevan el nombre de los operadores de comparación enriquecida que soportan:

```

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)`

Realiza «comparaciones enriquecidas» entre *a* y *b*. Específicamente, `lt(a, b)` es equivalente a `a < b`, `le(a, b)` es equivalente a `a <= b`, `eq(a, b)` es equivalente a `a == b`, `ne(a, b)` es equivalente a `a != b`, `gt(a, b)` es equivalente a `a > b` y `ge(a, b)` es equivalente a `a >= b`. Tenga en cuenta que estas funciones pueden retornar cualquier valor, que puede o no ser interpretable como un valor booleano. Consulte `comparisons` para obtener más información sobre las comparaciones enriquecidas.

Las operaciones lógicas también son aplicables a todos los objetos, y admiten pruebas de verdad, pruebas de identidad y operaciones booleanas:

`operator.not_(obj)`

`operator.__not__(obj)`

Retorna el resultado de `not obj`. (Tenga en cuenta que no hay ningún método `__not__()` para las instancias de objeto; solo el núcleo del intérprete define esta operación. El resultado se ve afectado por los métodos `__bool__()` y `__len__()`.)

`operator.truth(obj)`

Retorna `True` si *obj* es verdadero, y `False` de lo contrario. Esto equivale a usar el constructor `bool`.

`operator.is_(a, b)`

Retorna `a is b`. Chequea la identidad del objeto.

`operator.is_not(a, b)`

Retorna `a is not b`. Chequea la identidad del objeto.

Las operaciones matemáticas y a nivel de bits son las más numerosas:

`operator.abs(obj)`

`operator.__abs__(obj)`

Retorna el valor absoluto de *obj*.

`operator.add(a, b)`

`operator.__add__(a, b)`

Retorna `a + b`, para los números *a* y *b*.

`operator.and_(a, b)`

`operator.__and__(a, b)`

Retorna la «conjunción bit a bit» (*bitwise and*) de *a* y *b*.

`operator.floordiv(a, b)`

`operator.__floordiv__(a, b)`

Retorna `a // b`.

`operator.index(a)`

`operator.__index__(a)`

Retorna *a* convertido a un número entero. Equivalente a `a.__index__()`.

`operator.inv(obj)`

`operator.invert(obj)`

`operator.__inv__(obj)`

`operator.__invert__(obj)`

Retorna el «inverso bit a bit» (*bitwise inverse*) del número *obj*. Esto es equivalente a `~obj`.

`operator.lshift(a, b)`

`operator.__lshift__(a, b)`

Retorna *a* desplazado a izquierda (*shift left*) *b* bits.

`operator.mod(a, b)`

`operator.__mod__(a, b)`

Retorna `a % b`.

`operator.mul(a, b)`

`operator.__mul__(a, b)`

Retorna `a * b`, para los números *a* y *b*.

`operator.matmul(a, b)`

`operator.matmul(a, b)`

Retorna $a @ b$.

Nuevo en la versión 3.5.

`operator.neg(obj)`

`operator.__neg__(obj)`

Retorna *obj* negado ($-obj$).

`operator.or_(a, b)`

`operator.__or__(a, b)`

Retorna la «disyunción bit a bit» (*bitwise or*) de *a* y *b*.

`operator.pos(obj)`

`operator.__pos__(obj)`

Retorna *obj* positivo ($+obj$).

`operator.pow(a, b)`

`operator.__pow__(a, b)`

Retorna $a ** b$, para los números *a* y *b*.

`operator.rshift(a, b)`

`operator.__rshift__(a, b)`

Retorna *a* desplazado a derecha (*shift right*) *b* bits.

`operator.sub(a, b)`

`operator.__sub__(a, b)`

Retorna $a - b$.

`operator.truediv(a, b)`

`operator.__truediv__(a, b)`

Retorna a / b donde $2/3$ es $.66$ en lugar de 0 . Esto también se conoce como división «real» (*true division*).

`operator.xor(a, b)`

`operator.__xor__(a, b)`

Retorna la disyunción exclusiva bit a bit (*bitwise exclusive or*) de *a* y *b*.

Las operaciones que funcionan con secuencias (y algunas de ellas también con mapeos) incluyen:

`operator.concat(a, b)`

`operator.__concat__(a, b)`

Retorna $a + b$ para las secuencias *a* y *b*.

`operator.contains(a, b)`

`operator.__contains__(a, b)`

Retorna el resultado del chequeo $b \text{ in } a$. Notar que los operandos se invirtieron.

`operator.countOf(a, b)`

Retorna el número de ocurrencias de *b* en *a*.

`operator.delitem(a, b)`

`operator.__delitem__(a, b)`

Remueve el valor de *a* en el índice *b*.

`operator.getitem(a, b)`

`operator.__getitem__(a, b)`

Retorna el valor de *a* en el índice *b*.

`operator.indexOf(a, b)`

Retorna el índice de la primera ocurrencia de *b* en *a*.

`operator.setitem(a, b, c)`

`operator.__setitem__(a, b, c)`

Establece el valor de *a* en el índice *b* a *c*.

`operator.length_hint(obj, default=0)`

Retorna un largo estimativo del objeto *o*. Primero intenta retornar su largo real, luego un estimativo usando `object.__length_hint__()`, y finalmente retorna un valor predeterminado.

Nuevo en la versión 3.4.

El módulo `operator` también define herramientas para la obtención generalizada de atributos e ítems. Estas herramientas son útiles para utilizar rápidamente extractores de campos como argumentos de `map()`, `sorted()`, `itertools.groupby()`, u otras funciones que esperan una función como argumento.

`operator.attrgetter(attr)`

`operator.attrgetter(*attrs)`

Retorna un objeto invocable que obtiene *attr* de su operando. Si se solicita más de un atributo, retorna una tupla de los atributos. Los nombres de los atributos también pueden contener puntos. Por ejemplo:

- Después de `f = attrgetter('name')`, la llamada `f(b)` retorna `b.name`.
- Después de `f = attrgetter('name', 'date')`, la llamada `f(b)` retorna `(b.name, b.date)`.
- Después de `f = attrgetter('name.first', 'name.last')`, la llamada `f(b)` retorna `(b.name.first, b.name.last)`.

Equivalente a:

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    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
```

`operator.itemgetter(item)`

`operator.itemgetter(*items)`

Retorna un objeto invocable que obtiene *item* de su operando utilizando sobre el mismo el método `__getitem__()`. Si se especifican múltiples ítems, retorna una tupla con los valores obtenidos. Por ejemplo:

- Después de `f = itemgetter(2)`, la llamada `f(r)` retorna `r[2]`.
- Después de `g = itemgetter(2, 5, 3)`, la llamada `g(r)` retorna `(r[2], r[5], r[3])`.

Equivalente a:

```
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
```

Los ítems pueden ser de cualquier tipo aceptado por el método `__getitem__()` del operando. Los diccionarios aceptan cualquier valor *hasheable*. Las listas, las tuplas y las cadenas de caracteres aceptan un índice o un segmento:

```
>>> itemgetter(1)('ABCDEFGH')
'B'
>>> itemgetter(1, 3, 5)('ABCDEFGH')
('B', 'D', 'F')
>>> itemgetter(slice(2, None))('ABCDEFGH')
'CDEFG'
>>> soldier = dict(rank='captain', name='dotterbart')
>>> itemgetter('rank')(soldier)
'captain'
```

Ejemplos que utilizan `itemgetter()` para obtener campos específicos de un registro tupla:

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

`operator.methodcaller(name, /, *args, **kwargs)`

Retorna un objeto invocable que a su vez invoca al método `name` sobre su operando. Si se pasan argumentos adicionales y/o argumentos por palabra clave, estos serán a su vez pasados al método. Por ejemplo:

- Después de `f = methodcaller('name')`, la llamada `f(b)` retorna `b.name()`.
- Después de `f = methodcaller('name', 'foo', bar=1)`, la llamada `f(b)` retorna `b.name('foo', bar=1)`.

Equivalente a:

```
def methodcaller(name, /, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

10.3.1 Asignación de operadores a funciones

Esta tabla muestra cómo operaciones abstractas se corresponden con operadores simbólicos en la sintaxis de Python y las funciones en el módulo `operator`.

Operación	Sintaxis	Función
Adición	<code>a + b</code>	<code>add(a, b)</code>
Concatenación	<code>seq1 + seq2</code>	<code>concat(seq1, seq2)</code>
Chequeo de pertenencia	<code>obj in seq</code>	<code>contains(seq, obj)</code>
División	<code>a / b</code>	<code>truediv(a, b)</code>
División	<code>a // b</code>	<code>floordiv(a, b)</code>
Conjunción lógica bit a bit (<i>bitwise and</i>)	<code>a & b</code>	<code>and_(a, b)</code>
Disyunción lógica bit a bit (<i>bitwise exclusive or</i>)	<code>a ^ b</code>	<code>xor(a, b)</code>
Inversión bit a bit (<i>bitwise inversion</i>)	<code>~ a</code>	<code>invert(a)</code>
Disyunción lógica bit a bit (<i>bitwise or</i>)	<code>a b</code>	<code>or_(a, b)</code>
Exponenciación	<code>a ** b</code>	<code>pow(a, b)</code>
Identidad	<code>a is b</code>	<code>is_(a, b)</code>
Identidad	<code>a is not b</code>	<code>is_not(a, b)</code>
Asignación indexada	<code>obj[k] = v</code>	<code>setitem(obj, k, v)</code>
Eliminación indexada	<code>del obj[k]</code>	<code>delitem(obj, k)</code>
Indexado	<code>obj[k]</code>	<code>getitem(obj, k)</code>
Desplazamiento a izquierda (<i>left shift</i>)	<code>a << b</code>	<code>lshift(a, b)</code>
Módulo	<code>a % b</code>	<code>mod(a, b)</code>

Continúa en la página siguiente

Tabla 1 – proviene de la página anterior

Operación	Sintaxis	Función
Multiplicación	<code>a * b</code>	<code>mul(a, b)</code>
Multiplicación de matrices	<code>a @ b</code>	<code>matmul(a, b)</code>
Negación (aritmética)	<code>- a</code>	<code>neg(a)</code>
Negación (lógica)	<code>not a</code>	<code>not_(a)</code>
Positivo	<code>+ a</code>	<code>pos(a)</code>
Desplazamiento a derecha (<i>right shift</i>)	<code>a >> b</code>	<code>rshift(a, b)</code>
Asignación por segmento	<code>seq[i:j] = values</code>	<code>setitem(seq, slice(i, j), values)</code>
Eliminación por segmento	<code>del seq[i:j]</code>	<code>delitem(seq, slice(i, j))</code>
Segmentación	<code>seq[i:j]</code>	<code>getitem(seq, slice(i, j))</code>
Formateo de cadenas	<code>s % obj</code>	<code>mod(s, obj)</code>
Sustracción	<code>a - b</code>	<code>sub(a, b)</code>
Chequeo de verdad	<code>obj</code>	<code>truth(obj)</code>
Ordenado	<code>a < b</code>	<code>lt(a, b)</code>
Ordenado	<code>a <= b</code>	<code>le(a, b)</code>
Igualdad	<code>a == b</code>	<code>eq(a, b)</code>
Diferencia	<code>a != b</code>	<code>ne(a, b)</code>
Ordenado	<code>a >= b</code>	<code>ge(a, b)</code>
Ordenado	<code>a > b</code>	<code>gt(a, b)</code>

10.3.2 Operadores *In-place*

Muchas operaciones tienen una versión «*in-place*». Abajo se listan las funciones que proveen un acceso más primitivo a operadores *in-place* que la sintaxis usual; por ejemplo, el *statement* `x += y` es equivalente a `x = operator.iadd(x, y)`. Otra forma de decirlo es que `z = operator.iadd(x, y)` es equivalente a la sentencia (*statement*) compuesta `z = x; z += y`.

En esos ejemplos, notar que cuando se invoca un método *in-place*, el cómputo y la asignación se realizan en dos pasos separados. Las funciones *in-place* que se listan aquí debajo solo hacen el primer paso, llamar al método *in-place*. El segundo paso, la asignación, no se gestiona.

Para objetivos inmutables como cadenas de caracteres, números, y tuplas, el valor actualizado es computado, pero no es asignado de nuevo a la variable de entrada:

```
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

Para objetivos mutables como listas y diccionarios, el método *in-place* realiza la actualización, así que no es necesaria una asignación subsiguiente:

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

```
operator.iadd(a, b)
operator.__iadd__(a, b)
    a = iadd(a, b) es equivalente a a += b.
```

```
operator.iand(a, b)
operator.__iand__(a, b)
    a = iand(a, b) es equivalente a a &= b.
```

```
operator.iconcat(a, b)
```


`operator.__iconcat__(a, b)`
`a = iconcat(a, b)` es equivalente a `a += b` para dos *a* y *b* secuencias.

`operator.ifloordiv(a, b)`
`operator.__ifloordiv__(a, b)`
`a = ifloordiv(a, b)` es equivalente a `a //= b`.

`operator.ilshift(a, b)`
`operator.__ilshift__(a, b)`
`a = ilshift(a, b)` es equivalente a `a <<= b`.

`operator.imod(a, b)`
`operator.__imod__(a, b)`
`a = imod(a, b)` es equivalente a `a %= b`.

`operator.imul(a, b)`
`operator.__imul__(a, b)`
`a = imul(a, b)` es equivalente a `a *= b`.

`operator.imatmul(a, b)`
`operator.__imatmul__(a, b)`
`a = imul(a, b)` es equivalente a `a *= b`.

Nuevo en la versión 3.5.

`operator.ior(a, b)`
`operator.__ior__(a, b)`
`a = ior(a, b)` es equivalente a `a |= b`.

`operator.ipow(a, b)`
`operator.__ipow__(a, b)`
`a = ipow(a, b)` es equivalente a `a **= b`.

`operator.irshift(a, b)`
`operator.__irshift__(a, b)`
`a = irshift(a, b)` es equivalente a `a >>= b`.

`operator.isub(a, b)`
`operator.__isub__(a, b)`
`a = isub(a, b)` es equivalente a `a -= b`.

`operator.itruediv(a, b)`
`operator.__itruediv__(a, b)`
`a = itrueidiv(a, b)` es equivalente a `a /= b`.

`operator.ixor(a, b)`
`operator.__ixor__(a, b)`
`a = ixor(a, b)` es equivalente a `a ^= b`.

Acceso a archivos y directorios

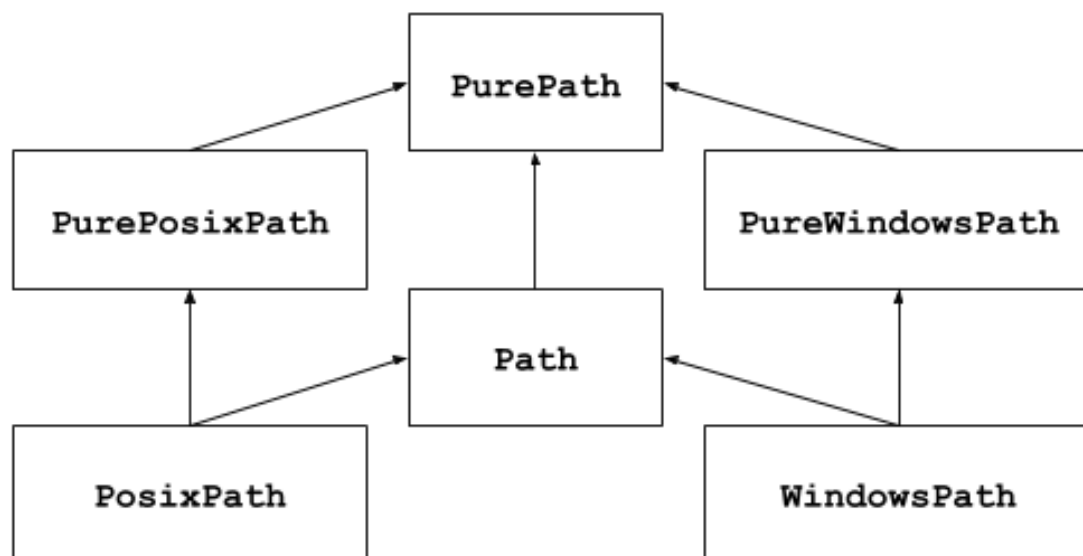
Los módulos descritos en este capítulo tratan de archivos de disco y directorios. Por ejemplo, hay módulos para leer las propiedades de los archivos, manipular rutas de acceso de forma portátil y crear archivos temporales. La lista completa de módulos en este capítulo es:

11.1 `pathlib` — Object-oriented filesystem paths

Nuevo en la versión 3.4.

Código fuente: [Lib/pathlib.py](#)

Este módulo ofrece clases que representan rutas del sistema de archivos con semántica apropiada para diferentes sistemas operativos. Las clases ruta se dividen entre *rutas puras*, que proporcionan operaciones puramente computacionales sin E/S; y *rutas concretas*, que heredan de rutas puras pero también proporcionan operaciones de E/S.



Si nunca has usado este módulo o simplemente no estás seguro de qué clase es la adecuada para tu tarea, `Path` es probablemente lo que necesitas. Crea una instancia *ruta concreta* para la plataforma en la que se ejecuta el código.

Las rutas puras son útiles en algunos casos especiales, por ejemplo:

1. Si deseas manipular las rutas de Windows en una máquina Unix (o viceversa). No puedes crear una instancia de `WindowsPath` cuando se ejecuta en Unix, pero puedes crear una instancia de `PureWindowsPath`.
2. Desea asegurar que su código solo manipule rutas sin acceder realmente al sistema operativo. En este caso, crear instancias de una de las clases puras puede ser útil, ya que simplemente no tienen ninguna operación de acceso al sistema operativo.

Ver también:

PEP 428: El módulo `pathlib` – rutas de sistema orientadas a objetos.

Ver también:

Para la manipulación de rutas de bajo nivel en cadenas, también puede usar el módulo `os.path`.

11.1.1 Uso básico

Importar la clase principal:

```
>>> from pathlib import Path
```

Listado de subdirectorios:

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

Listing Python source files in this directory tree:

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

Navegar dentro de un árbol de directorios:

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

Consultar propiedades de ruta:

```
>>> q.exists()
True
>>> q.is_dir()
False
```

Abrir un archivo:

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2 Rutas puras

Los objetos ruta pura proporcionan operaciones de manejo de rutas que en realidad no acceden al sistema de archivos. Hay tres formas de acceder a estas clases, que llamaremos *familias*:

class `pathlib.PurePath(*pathsegments)`

Una clase genérica que representa la familia de rutas del sistema (al crear una instancia se crea *PurePosixPath* o *PureWindowsPath*):

```
>>> PurePath('setup.py')           # Running on a Unix machine
PurePosixPath('setup.py')
```

Each element of *pathsegments* can be either a string representing a path segment, an object implementing the *os.PathLike* interface which returns a string, or another path object:

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

Cuando *pathsegments* está vacío, se asume el directorio actual:

```
>>> PurePath()
PurePosixPath('.')
```

Cuando se proporcionan varias rutas absolutas, la última se toma como un ancla (copiando el comportamiento de *os.path.join()*):

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

Sin embargo, en una ruta de Windows, cambiar la raíz local no elimina la configuración de la unidad anterior:

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

Spurious slashes and single dots are collapsed, but double dots ('..') are not, since this would change the meaning of a path in the face of symbolic links:

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

(un enfoque naif haría pensar que `PurePosixPath('foo/../bar')` es equivalente a `PurePosixPath('bar')`, lo cual es incorrecto si `foo` es un enlace simbólico a otro directorio)

Los objetos ruta pura implementan la interfaz `os.PathLike`, lo que permite su uso en cualquier lugar donde se acepte la interfaz.

Distinto en la versión 3.6: Se agregó soporte para la interfaz `os.PathLike`.

class `pathlib.PurePosixPath(*pathsegments)`

Una subclase de `PurePath`, esta familia representa rutas que no son del sistema de archivos de Windows:

```
>>> PurePosixPath('/etc')
PurePosixPath('/etc')
```

`pathsegments` se especifica de manera similar a `PurePath`.

class `pathlib.PureWindowsPath(*pathsegments)`

A subclass of `PurePath`, this path flavour represents Windows filesystem paths:

```
>>> PureWindowsPath('c:/Program Files/')
PureWindowsPath('c:/Program Files')
```

`pathsegments` se especifica de manera similar a `PurePath`.

Independientemente del sistema en el que se encuentre, puede crear instancias de todas estas clases, ya que no proporcionan ninguna operación que llame al sistema operativo.

Propiedades generales

Paths are immutable and hashable. Paths of a same flavour are comparable and orderable. These properties respect the flavour's case-folding semantics:

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

Rutas de diferentes familias no son iguales y no se pueden ordenar:

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and
↪ 'PurePosixPath'
```

Operadores

El operador barra ayuda a crear rutas secundarias, similar a `os.path.join()`:

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
```

Se puede usar un objeto ruta en cualquier lugar donde se acepte un objeto que implemente `os.PathLike`:

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

The string representation of a path is the raw filesystem path itself (in native form, e.g. with backslashes under Windows), which you can pass to any function taking a file path as a string:

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\\Program Files'
```

Del mismo modo, llamar a `bytes` en una ruta proporciona la ruta cruda del sistema de archivos como un objeto bytes, codificado por `os.fsencode()`:

```
>>> bytes(p)
b'/etc'
```

Nota: Llamar a `bytes` solo se recomienda en Unix. En Windows, la forma unicode es la representación canónica de las rutas del sistema de archivos.

Acceso a partes individuales

Para acceder a las «partes» (componentes) individuales de una ruta, use la siguiente propiedad:

`PurePath.parts`

Una tupla que da acceso a los diversos componentes de la ruta:

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

(obsérvese cómo la unidad y la raíz local se reagrupan en una sola parte)

Métodos y propiedades

Las rutas puras proporcionan los siguientes métodos y propiedades:

`PurePath.drive`

Una cadena que representa la letra o el nombre de la unidad, si corresponde:

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

UNC shares are also considered drives:

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

`PurePath.root`

Una cadena que representa la raíz (local o global), si la hay:

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

Las localizaciones UNC siempre tienen una raíz:

```
>>> PureWindowsPath('//host/share').root
'\\'
```

`PurePath.anchor`

La concatenación de la unidad y la raíz:

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

`PurePath.parents`

Una secuencia inmutable que proporciona acceso a los ancestros lógicos de la ruta:

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

`PurePath.parent`

El padre lógico de la ruta

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```


No puede ir más allá de un ancla o una ruta vacía:

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

Nota: Esta es una operación puramente léxica, de ahí el siguiente comportamiento:

```
>>> p = PurePosixPath('foo/..')
>>> p.parent
PurePosixPath('foo')
```

Si desea explorar una ruta arbitraria del sistema de archivos, se recomienda llamar primero a `Path.resolve()` para resolver los enlaces simbólicos y eliminar los componentes «..».

`PurePath.name`

Una cadena que representa el componente final de la ruta, excluyendo la unidad y la raíz, si hay alguna:

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

Los nombres de unidad UNC no se consideran:

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

`PurePath.suffix`

La extensión del archivo del componente final, si lo hay:

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

`PurePath.suffixes`

Una lista de las extensiones de archivo de la ruta:

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

`PurePath.stem`

El componente final de la ruta, sin su sufijo:

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

`PurePath.as_posix()`

Retorna una cadena que representa la ruta con barras invertidas (/):

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

`PurePath.as_uri()`

Representa la ruta como un file URI. *ValueError* se genera si la ruta no es absoluta.

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = PureWindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

`PurePath.is_absolute()`

Retorna si la ruta es absoluta o no. Una ruta se considera absoluta si tiene una raíz y (si la familia lo permite) una unidad:

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

`PurePath.is_reserved()`

Con *PureWindowsPath*, retorna True si la ruta se considera reservada en Windows, False en caso contrario. Con *PurePosixPath*, siempre retorna False.

```
>>> PureWindowsPath('nul').is_reserved()
True
>>> PurePosixPath('nul').is_reserved()
False
```

Las llamadas al sistema de archivos en rutas reservadas pueden fallar inesperadamente o tener efectos no deseados.

`PurePath.joinpath(*other)`

Llamar a este método es equivalente a combinar la ruta con cada uno de los *other* argumentos:

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

`PurePath.match(pattern)`

Match this path against the provided glob-style pattern. Return `True` if matching is successful, `False` otherwise.

Si *pattern* es relativo, la ruta puede ser relativa o absoluta, y la coincidencia se realiza desde la derecha:

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

Si *pattern* es absoluto, la ruta debe ser absoluta y toda la ruta debe coincidir:

```
>>> PurePath('/a.py').match('/*.py')
True
>>> PurePath('a/b.py').match('/*.py')
False
```

Al igual que con otros métodos, la distinción entre mayúsculas y minúsculas sigue los valores predeterminados de la plataforma:

```
>>> PurePosixPath('b.py').match('*.PY')
False
>>> PureWindowsPath('b.py').match('*.PY')
True
```

`PurePath.relative_to(*other)`

Compute a version of this path relative to the path represented by *other*. If it's impossible, `ValueError` is raised:

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 694, in relative_to
    .format(str(self), str(formatted))
ValueError: '/etc/passwd' does not start with '/usr'
```

`PurePath.with_name(name)`

Retorna una nueva ruta con *name* cambiado. Si la ruta original no tiene nombre, se genera `ValueError`:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

`PurePath.with_suffix(suffix)`

Retorna una nueva ruta con *suffix* cambiado. Si la ruta original no tiene un sufijo, se agrega el nuevo *suffix*. Si *suffix* es una cadena vacía, el sufijo original se elimina:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
```

(continué en la próxima página)

(proviene de la página anterior)

```
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```

11.1.3 Rutas concretas

Las rutas concretas son subclases de las rutas puras. Además de las operaciones proporcionadas por estas últimas, también proporcionan métodos realizar llamadas del sistema a objetos ruta. Hay tres formas de crear instancias de rutas concretas:

class `pathlib.Path(*pathsegments)`

Una subclase de `PurePath`, esta clase representa rutas concretas de la *familia* ruta del sistema (al crear una instancia crea ya sea `PosixPath` o `WindowsPath`):

```
>>> Path('setup.py')
PosixPath('setup.py')
```

`pathsegments` se especifica de manera similar a `PurePath`.

class `pathlib.PosixPath(*pathsegments)`

Una subclase de `Path` y `PurePosixPath`, esta clase representa rutas concretas de sistemas de archivos que no son de Windows:

```
>>> PosixPath('/etc')
PosixPath('/etc')
```

`pathsegments` se especifica de manera similar a `PurePath`.

class `pathlib.WindowsPath(*pathsegments)`

Una subclase de `Path` y `PureWindowsPath`, esta clase representa rutas concretas del sistema de archivos de Windows:

```
>>> WindowsPath('c:/Program Files/')
WindowsPath('c:/Program Files')
```

`pathsegments` se especifica de manera similar a `PurePath`.

Solo puedes crear instancias de la *familia* de clase que corresponde a su sistema operativo (permitir llamadas del sistema no compatibles podría provocar errores o fallas en su aplicación):

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
NotImplementedError: cannot instantiate 'WindowsPath' on your system
```

Métodos

Las rutas concretas proporcionan los siguientes métodos -además de los métodos de ruta puros-. Muchos de estos métodos pueden generar un `OSError` si falla una llamada al sistema (por ejemplo, porque la ruta no existe).

Distinto en la versión 3.8: `exists()`, `is_dir()`, `is_file()`, `is_mount()`, `is_symlink()`, `is_block_device()`, `is_char_device()`, `is_fifo()`, `is_socket()` ahora retorna “False” en lugar de generar una excepción para las rutas que contienen caracteres que no se pueden representar a nivel del sistema operativo.

classmethod `Path.cwd()`

Retorna un nuevo objeto ruta que representa el directorio actual (como lo retorna `os.getcwd()`):

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

classmethod `Path.home()`

Retorna un nuevo objeto ruta que representa el directorio de inicio del usuario (como lo retorna `os.path.expanduser()` con el agregado `~`):

```
>>> Path.home()
PosixPath('/home/antoine')
```

Nuevo en la versión 3.5.

`Path.stat()`

Retorna un objeto `os.stat_result` que contiene información sobre esta ruta, del mismo modo que `os.stat()`. El resultado se consulta en cada llamada a este método.

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

`Path.chmod(mode)`

Cambia el modo y los permisos de archivo, como `os.chmod()`:

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

`Path.exists()`

Si la ruta apunta a un archivo o directorio existente:

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

Nota: Si la ruta apunta a un enlace simbólico, `exist()` retorna si el enlace simbólico *apunta a* un archivo o directorio existente.

`Path.expanduser()`

Retorna una nueva ruta con las construcciones `~` y `~user` expandidas, como lo retorna `os.path.expanduser()`:

```
>>> p = PosixPath('~ /films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

Nuevo en la versión 3.5.

`Path.glob(pattern)`

Glob the given relative *pattern* in the directory represented by this path, yielding all matching files (of any kind):

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('docs/conf.py')]
```

El patrón «`**`» significa «este directorio y todos los subdirectorios de forma recursiva». En otras palabras, habilita el comodín recursivo:

```
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

Nota: El uso del patrón «`**`» en grandes árboles de directorios puede consumir una cantidad excesiva de tiempo.

`Path.group()`

Retorna el nombre del grupo propietario del archivo. `KeyError` se genera si el *gid* del archivo no se encuentra en la base de datos del sistema.

`Path.is_dir()`

Retorna `True` si la ruta apunta a un directorio (o un enlace simbólico que apunta a un directorio), `False` si apunta a otro tipo de archivo.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.is_file()`

Retorna `True` si la ruta apunta a un archivo normal (o un enlace simbólico que apunta a un archivo normal), `Falso` si apunta a otro tipo de archivo.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.is_mount()`

Return `True` if the path is a *mount point*: a point in a file system where a different file system has been mounted. On POSIX, 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. Not implemented on Windows.

Nuevo en la versión 3.7.

`Path.is_symlink()`

Retorna `True` si la ruta apunta a un enlace simbólico, `False` de lo contrario.

“`False`” también se retorna si la ruta no existe; se extiende a otros errores (como errores de permiso).

`Path.is_socket()`

Retorna `True` si la ruta apunta a un *socket* Unix (o un enlace simbólico que apunta a uno), `False` si apunta a otro tipo de archivo.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.is_fifo()`

Retorna `True` si la ruta apunta a un *FIFO* (o un enlace simbólico que apunta a un *FIFO*), `False` si apunta a otro tipo de archivo.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.is_block_device()`

Retorna `True` si la ruta apunta a un dispositivo de bloques (o un enlace simbólico que apunta a uno), `False` si apunta a otro tipo de archivo.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.is_char_device()`

Retorna `True` si la ruta apunta a un dispositivo de caracteres (o un enlace simbólico que apunta a uno), `False` si apunta a otro tipo de archivo.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.iterdir()`

Cuando la ruta apunta a un directorio, produce objetos de ruta del contenido del directorio:

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

The children are yielded in arbitrary order, and the special entries `'.'` and `'..'` are not included. If a file is removed from or added to the directory after creating the iterator, whether an path object for that file be included is unspecified.

`Path.lchmod(mode)`

Del mismo modo que `Path.chmod()` pero si la ruta apunta a un enlace simbólico, el modo del enlace simbólico cambia en lugar del de su objetivo.

`Path.lstat()`

Del mismo modo que `Path.stat()` pero si la ruta apunta a un enlace simbólico, retorna la información del enlace simbólico en lugar de la de su objetivo.

`Path.mkdir(mode=0o777, parents=False, exist_ok=False)`

Crea un nuevo directorio en la ruta dada. Si se proporciona *mode*, se combina con el valor del proceso *umask* para determinar el modo de archivo y los derechos de acceso. Si la ruta ya existe, se genera `FileExistsError`.

Si *parents* es verdadero, los padres que faltan de esta ruta se crean según sea necesario; se crean con los permisos predeterminados sin tener en cuenta *mode* (imitando el comando POSIX `mkdir -p`).

Si *parents* es falso (el valor predeterminado), se genera un padre que falta `FileNotFoundError`.

Si *exist_ok* es falso (el valor predeterminado), se genera `FileExistsError` si el directorio de destino ya existe.

Si `exist_ok` es verdadero, se ignorarán las excepciones `FileExistsError` (el mismo comportamiento que el comando POSIX `mkdir -p`), pero solo si el último componente de ruta no es un archivo existente que no sea de directorio.

Distinto en la versión 3.5: Se agregó el parámetro `exist_ok`.

`Path.open(mode='r', buffering=-1, encoding=None, errors=None, newline=None)`

Abre el archivo señalado por la ruta, como lo hace la función incorporada `open()`:

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

`Path.owner()`

Retorna el nombre del usuario propietario del archivo. `KeyError` se genera si el `uid` del archivo no se encuentra en la base de datos del sistema.

`Path.read_bytes()`

Retorna el contenido binario del archivo apuntado como un objeto bytes:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

Nuevo en la versión 3.5.

`Path.read_text(encoding=None, errors=None)`

Retorna el contenido decodificado del archivo apuntado como una cadena:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

El archivo se abre y luego se cierra. Los parámetros opcionales funcionan de la misma manera que en `open()`.

Nuevo en la versión 3.5.

`Path.rename(target)`

Cambia el nombre del archivo o directorio apuntado al de `target` y retorna una nueva instancia de `Path` que apunte a `target`. En Unix, si `target` existe, es un archivo y el usuario tiene permisos, será reemplazado sin notificar. `target` puede ser una cadena u otro objeto ruta:

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
PosixPath('bar')
>>> target.open().read()
'some text'
```

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the Path object.

Distinto en la versión 3.8: Valor de retorno agregado, retorna la nueva instancia de `Path`.

`Path.replace(target)`

Cambia el nombre del archivo o directorio al de `target` y retorna una nueva instancia de `Path` que apunte a `target`. Si `target` apunta a un archivo o directorio existente, será reemplazado incondicionalmente.

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the Path object.

Distinto en la versión 3.8: Valor de retorno agregado, retorna la nueva instancia de *Path*.

`Path.resolve(strict=False)`

Hace que la ruta sea absoluta, resolviendo los enlaces simbólicos. Se retorna un nuevo objeto ruta:

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

Los componentes «. .» también se eliminan (este es el único método para hacerlo):

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

Si la ruta no existe y *strict* es *True*, se genera *FileNotFoundError*. Si *strict* es *False*, la ruta se resuelve en la medida de lo posible y se agrega el resto sin verificar si existe. Si se encuentra un bucle infinito en la resolución de la ruta se genera *RuntimeError*.

Nuevo en la versión 3.6: The *strict* argument (pre-3.6 behavior is strict).

`Path.rglob(pattern)`

Idéntico a llamar a *Path.glob()* con «**/» agregado delante del *pattern* relativo:

```
>>> sorted(Path().rglob("*.py"))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

`Path.rmdir()`

Elimina el directorio. El directorio debe estar vacío.

`Path.samefile(other_path)`

Retorna si la ruta apunta al mismo archivo que *other_path*, que puede ser un objeto *Path* o una cadena. La semántica es similar a *os.path.samefile()* y *os.path.samestat()*.

Se puede generar *OSError* si no se accede a alguno de los archivos por algún motivo.

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

Nuevo en la versión 3.5.

`Path.symlink_to(target, target_is_directory=False)`

Hace de la ruta un enlace simbólico a *target*. En Windows, *target_is_directory* debe ser verdadero si el destino del enlace es un directorio (*False* es predeterminado). En POSIX, el valor de *target_is_directory* se ignora.

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> p.lstat().st_size
8
```

Nota: El orden de los argumentos (*link*, *target*) es el reverso de `os.symlink()`'s.

`Path.link_to(target)`
Make *target* a hard link to this path.

Advertencia: This function does not make this path a hard link to *target*, despite the implication of the function and argument names. The argument order (target, link) is the reverse of `Path.symlink_to()`, but matches that of `os.link()`.

Nuevo en la versión 3.8.

`Path.touch(mode=0o666, exist_ok=True)`

Crea un archivo en la ruta dada. Si se proporciona *mode*, se combina con el valor del proceso `umask` para determinar el modo de archivo y los indicadores de acceso. Si el archivo ya existe, la función tiene éxito si *exist_ok* es verdadero (y su hora de modificación se actualiza a la hora actual), de lo contrario se genera `FileExistsError`.

`Path.unlink(missing_ok=False)`

Elimine el archivo o enlace simbólico. Si la ruta apunta a un directorio, use `Path.rmdir()` en su lugar.

Si *missing_ok* es falso (el valor predeterminado), se genera `FileNotFoundError` si la ruta no existe.

Si *missing_ok* es verdadero, las excepciones `FileNotFoundError` serán ignoradas (el mismo comportamiento que el comando POSIX `rm -f`).

Distinto en la versión 3.8: Se agregó el parámetro *missing_ok*.

`Path.write_bytes(data)`

Abre el archivo apuntado en modo bytes, escribe *data* y cierra el archivo:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

Se sobrescribe un archivo existente con el mismo nombre.

Nuevo en la versión 3.5.

`Path.write_text(data, encoding=None, errors=None)`

Abre el archivo apuntado en modo texto, escribe *data* y cierra el archivo:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

Se sobrescribe un archivo existente con el mismo nombre. Los parámetros opcionales tienen el mismo significado que en `open()`.

Nuevo en la versión 3.5.

11.1.4 Correspondencia a herramientas en el módulo `os`

A continuación se muestra una tabla que asigna varias funciones `os` a sus equivalentes en `PurePath/Path`.

Nota: Aunque `os.path.relpath()` y `PurePath.relative_to()` tienen algunos casos de uso superpuestos, su semántica difiere lo suficiente como para justificar no considerarlos equivalentes.

os y os.path	pathlib
<code>os.path.abspath()</code>	<code>Path.resolve()</code>
<code>os.chmod()</code>	<code>Path.chmod()</code>
<code>os.mkdir()</code>	<code>Path.mkdir()</code>
<code>os.rename()</code>	<code>Path.rename()</code>
<code>os.replace()</code>	<code>Path.replace()</code>
<code>os.rmdir()</code>	<code>Path.rmdir()</code>
<code>os.remove()</code> , <code>os.unlink()</code>	<code>Path.unlink()</code>
<code>os.getcwd()</code>	<code>Path.cwd()</code>
<code>os.path.exists()</code>	<code>Path.exists()</code>
<code>os.path.expanduser()</code>	<code>Path.expanduser()</code> y <code>Path.home()</code>
<code>os.listdir()</code>	<code>Path.iterdir()</code>
<code>os.path.isdir()</code>	<code>Path.is_dir()</code>
<code>os.path.isfile()</code>	<code>Path.is_file()</code>
<code>os.path.islink()</code>	<code>Path.is_symlink()</code>
<code>os.link()</code>	<code>Path.link_to()</code>
<code>os.symlink()</code>	<code>Path.symlink_to()</code>
<code>os.stat()</code>	<code>Path.stat()</code> , <code>Path.owner()</code> , <code>Path.group()</code>
<code>os.path.isabs()</code>	<code>PurePath.is_absolute()</code>
<code>os.path.join()</code>	<code>PurePath.joinpath()</code>
<code>os.path.basename()</code>	<code>PurePath.name</code>
<code>os.path.dirname()</code>	<code>PurePath.parent</code>
<code>os.path.samefile()</code>	<code>Path.samefile()</code>
<code>os.path.splitext()</code>	<code>PurePath.suffix</code>

11.2 `os.path` — Manipulaciones comunes de nombre de ruta

Código fuente: `Lib/posixpath.py` (para POSIX) y `Lib/ntpath.py` (para Windows NT).

Este módulo implementa algunas funciones útiles en nombres de ruta. Para leer o escribir archivos consulta `open()`, y para acceder al sistema de archivos consulta el módulo `os`. Los parámetros de ruta puede ser pasados tanto siendo cadenas o bytes. Se recomienda que las aplicaciones representen los nombres de archivo como caracteres de cadena (Unicode). Desafortunadamente, algunos nombres de archivo puede que no sean representables como cadenas en Unix, así que las aplicaciones que necesiten tener soporte para nombres de archivo arbitrarios deberían usar objetos byte para representar nombres de archivo. Viceversa, el uso de objetos de bytes no puede representar todos los nombres de archivo en Windows (en la codificación estándar `mbcs`), por lo tanto, las aplicaciones de Windows deben usar objetos de cadena para acceder a todos los archivos.

A diferencia de un shell Unix, Python no realiza ninguna expansión de ruta *automática*. Las funciones como `expanduser()` y `expandvars()` puede ser invocadas de manera explícita cuando una aplicación desea realizar una expansión de ruta *shell-like*. (Consulta el módulo `glob`)

Ver también:

El módulo `pathlib` ofrece objetos de ruta de alto nivel.

Nota: Todas estas funciones aceptan solo bytes o solo objetos de cadena como sus parámetros. El resultado es un objeto del mismo tipo, si se retorna una ruta o un nombre de archivo.

Nota: Dado que los diferentes sistemas operativos tienen diferentes convenciones de nombres de ruta, existen varias versiones de este módulo en la librería estándar. El módulo `os.path` siempre es el módulo adecuado para el sistema operativo en el cual Python está operando, y por lo tanto es utilizable para rutas locales. Sin embargo, también puedes importar y utilizar los módulos individuales si deseas manipular una ruta que *siempre* está en uno de los diferentes formatos. Todos tienen la misma interfaz:

- `posixpath` para rutas con estilo UNIX
 - `ntpath` para rutas Windows
-

Distinto en la versión 3.8: `exists()`, `lexists()`, `isdir()`, `isfile()`, `islink()`, y `ismount()` ahora retornan `False` en lugar de lanzar una excepción para rutas que contienen caracteres o bytes que no se puedan representar a nivel de sistema operativo.

`os.path.abspath(path)`

Retorna una versión normalizada y absoluta del nombre de ruta `path`. En la mayoría de las plataformas esto es el equivalente a invocar la función `normpath` de la siguiente manera: `normpath(join(os.getcwd(), path))`()`.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.basename(path)`

Retorna un nombre base de nombre de ruta `path`. Este es el segundo elemento del par retornado al pasar `path` a la función `split()`. Toma en cuenta que el resultado de esta función es diferente a la del programa de Unix **basename**; donde **basename** para `'/foo/bar/'` retorna `'bar'`, la función `basename()` retorna una cadena vacía `('')`.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.commonpath(paths)`

Retorna la sub-ruta común más larga de cada nombre de ruta en la secuencia `paths`. Lanza una excepción `ValueError` si `paths` contiene nombres de ruta absolutos y relativos, o los `paths` están en discos diferentes o si `paths` está vacío. A diferencia de `commonprefix()`, retorna una ruta válida.

Availability: Unix, Windows.

Nuevo en la versión 3.5.

Distinto en la versión 3.6: Acepta una secuencia de *path-like objects*.

`os.path.commonprefix(list)`

Retorna el prefijo de ruta más largo (tomado carácter por carácter) que es un prefijo de todas las rutas en `list`. Si `list` está vacía, retorna la cadena vacía `('')`.

Nota: Esta función puede que retorne rutas inválidas porque trabaja un carácter a la vez. Para obtener una ruta válida, consulta `commonpath()`.

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
'/usr/l'

>>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.dirname(path)`

Retorna el nombre del directorio de la ruta *path*. Es el primer elemento del par retornado al pasar *path* a la función `split()`.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.exists(path)`

Retorna `True` si *path* se refiere a una ruta existente o un descriptor de archivo abierto. Retorna `False` para enlaces simbólicos rotos. En algunas plataformas, esta función puede retornar `False` si no se concede permiso para ejecutar `os.stat()` en el archivo solicitado, incluso la ruta *path* existe físicamente.

Distinto en la versión 3.3: *path* ahora puede ser un valor entero: retorna `True` si es un descriptor de archivo abierto, de otro modo retorna `False`.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.lexists(path)`

Retorna `True` si *path* se refiere a un camino existente. Retorna `True` para los enlaces simbólicos rotos. Equivalente a `exists()` en plataformas que carecen de `os.lstat()`.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.expanduser(path)`

En Unix y Windows, retorna el argumento con un componente inicial de `~` o `~user` reemplazado por el directorio *home* de *user*.

En Unix, el `~` inicial es reemplazado por la variable de entorno `HOME` si está activada; si no, el directorio principal del usuario actual se busca en el directorio de contraseñas a través del módulo incorporado `pwd`. Un `~user` inicial es buscado directamente en el directorio de contraseñas.

En Windows, se usará `USERPROFILE` si está configurado, si no, se usará una combinación de `HOME` y `HOMEDRIVE`. Un `~user` inicial se maneja quitando el último componente del directorio de la ruta de usuario creada derivada arriba.

Si la expansión falla o si la ruta no comienza con una tilde, la ruta se retorna sin cambios.

Distinto en la versión 3.6: Acepta un *path-like object*.

Distinto en la versión 3.8: Ya no utiliza `HOME` en Windows.

`os.path.expandvars(path)`

Retorna el argumento con variables de entorno expandidas. Las sub-cadenas de la forma `$name` or `${name}` se reemplazan por el valor de la variable de entorno *name*. Los nombres de variables mal formadas y las referencias a variables no existentes se dejan sin cambios.

En Windows, las expansiones `%name%` están soportadas además de `$name` y `${name}`.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.getatime(path)`

Retorna la hora del último acceso de *path*. El valor de retorno es un número de punto flotante que da el número de segundos desde la época (Consulta el módulo `time`). Lanza una excepción `OSError` si el archivo no existe o es inaccesible.

`os.path.getmtime(path)`

Retorna el tiempo de la última modificación de *path*. El valor de retorno es un número de punto flotante que da el número de segundos desde la época (consulta el módulo `time`). lanza una excepción `OSError` si el archivo no existe o es inaccesible.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.getctime(path)`

Retorna el *ctime* del sistema que, en algunos sistemas (como Unix) es la hora del último cambio de metadatos y, en otros (como Windows), es el tiempo de creación de *path*. El valor retornado es un número que da el número de segundos desde la época (consulta el módulo `time`). Lanza una excepción `OSError` si el archivo no existe o es inaccesible.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.getsize(path)`

Retorna el tamaño en bytes de *path*, Lanza una excepción *OSError* si el archivo no existe o es inaccesible.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.isabs(path)`

Retorna True si *path* es un nombre de ruta de acceso absoluto. En Unix, eso significa que comienza con una barra diagonal, en Windows que comienza con una barra diagonal (invertida) después de cortar una letra de unidad potencial.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.isfile(path)`

Retorna True si *path* es un archivo *existing*. Esto sigue los enlaces simbólicos, por lo que tanto *islink()* como *isfile()* pueden ser verdaderos para la misma ruta.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.isdir(path)`

Retorna True si *path* es un directorio *existing*. Esto sigue los enlaces simbólicos, por lo que tanto *islink()* como *isdir()* pueden ser verdaderos para la misma ruta.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.islink(path)`

Retorna True si *path* hace referencia a una entrada de directorio *existing* que es un enlace simbólico. Siempre False si el entorno de ejecución de Python no admite vínculos simbólicos..

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.ismount(path)`

Retorna True si el nombre de ruta *path* es un *mount point*: un punto en un sistema de archivos donde se ha montado un sistema de archivos diferente. En POSIX, la función comprueba si el elemento primario de *path*, *path/. . .*, se encuentra en un dispositivo diferente de *path*, o si *path/. . .* y *path* apuntan al mismo *i-node* en el mismo dispositivo — esto debería detectar puntos de montaje para todas las variantes Unix y POSIX. No es capaz de detectar de forma fiable los montajes de enlace en el mismo sistema de archivos. En Windows, una raíz de letra de unidad y un recurso compartido UNC siempre son puntos de montaje, y para cualquier otra ruta de acceso `GetVolumePathName` se llama para ver si es diferente de la ruta de acceso de entrada.

Nuevo en la versión 3.4: Soporte para detectar puntos de montaje *non-root* en Windows.

Distinto en la versión 3.6: Acepta un *path-like object*.

`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 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.

En Windows, la letra de la unidad no se restablece cuando se encuentra un componente de ruta absoluta (por ejemplo, `r'\foo'`). Si un componente contiene una letra de unidad, todos los componentes anteriores se desechan y la letra de unidad se restablece. Ten en cuenta que, dado que hay un directorio actual para cada unidad, “`os.path.join(«c:», «foo»)`” “representa una ruta relativa al directorio actual en la unidad C: (`c:\foo`), no `c:\foo`.”

Distinto en la versión 3.6: Acepta un objeto *path-like object* para *path* y *paths*.

`os.path.normcase(path)`

Normaliza las mayúsculas y minúsculas de un nombre de ruta. En Windows convierte todos los caracteres en el nombre de ruta a minúsculas y también convierte las barras inclinadas hacia atrás en barras inclinadas hacia atrás. En otros sistemas operativos, retorna la ruta sin cambios.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.normpath(path)`

Normaliza un nombre de ruta colapsando separadores redundantes y referencias de nivel superior para que `A//B`, `A/B/`, `A/. /B` y `A/foo/. /B` se transformen en “`A/B`”. Esta modificación de cadena puede que modifique

el significado de la ruta que contenga enlaces simbólicos. En Windows, convierte las barras inclinadas hacia adelante en barras hacia atrás. Para normalizar mayúsculas y minúsculas, utiliza `normcase()`.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.realpath(path)`

Retorna la ruta canónica del nombre de archivo especificado, eliminando cualquier enlace simbólico encontrado en la ruta (si es que tienen soporte por el sistema operativo).

Nota: Cuando ocurren ciclos de enlaces simbólicos, la ruta retornada será un miembro del ciclo, pero no se garantiza cuál miembro será.

Distinto en la versión 3.6: Acepta un *path-like object*.

Distinto en la versión 3.8: Los enlaces y uniones simbólicos ahora se resuelven en Windows.

`os.path.relpath(path, start=os.curdir)`

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*. On Windows, `ValueError` is raised when *path* and *start* are on different drives.

start toma de forma predeterminada el valor de `os.curdir`.

Availability: Unix, Windows.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.samefile(path1, path2)`

Retorna `True` si ambos argumentos de nombre de ruta refieren al mismo archivo o directorio. Esto se determina por el número de dispositivo y el número de *i-node* y lanza una excepción si una llamada de `os.stat()` en alguno de los nombres de ruta falla.

Availability: Unix, Windows.

Distinto en la versión 3.2: Añadido soporte para Windows.

Distinto en la versión 3.4: Windows ahora utiliza la misma implementación que en el resto de las plataformas.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.sameopenfile(fp1, fp2)`

Retorna `True` si los descriptores de archivo *fp1* y *fp2* se refieren al mismo archivo.

Availability: Unix, Windows.

Distinto en la versión 3.2: Añadido soporte para Windows.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.samestat(stat1, stat2)`

Retorna `True` si las tuplas de *stat* (*stat1* y *stat2*) refieren al mismo archivo. Estas estructuras pueden haber sido retornadas por `os.fstat()`, `os.lstat()`, o `os.stat()`. Esta función implementa la comparación subyacente utilizada por: `samefile()` y `sameopenfile()`.

Availability: Unix, Windows.

Distinto en la versión 3.4: Añadido soporte para Windows.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.split(path)`

Divide el nombre de la ruta *path* * en un par, “(*head*, *tail*)” “donde **tail* es el último componente del nombre de la ruta y *head* es todo lo que conduce a eso. La parte *head* nunca contendrá una barra; si *head* termina en una barra, *tail* estará vacía. Si no hay barra inclinada en *path*, *head* estará vacío. Si *path* está vacía, tanto *head* como *tail* estarán vacíos. Las barras diagonales finales se eliminan de *head* a menos que sea la raíz (solo una o más barras). En todos los casos, `join(head, tail)` retorna una ruta a la misma ubicación que *path* (pero las cadenas pueden diferir). Consulta las funciones `dirname()` y `basename()`.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.splitdrive(path)`

Divide el nombre de ruta *path* en un par (*drive*, *tail*) donde *drive* es un punto de montaje o una cadena vacía. En sistemas que no utilizan especificaciones de unidad, *drive* siempre será una cadena vacía. En todos los casos, *drive* + *tail* será lo mismo que *path*.

En Windows, divide un nombre de ruta en unidad / punto compartido UNC y ruta relativa.

Si la ruta contiene una letra de unidad, la unidad contendrá todo hasta los dos puntos inclusive. p.ej. `splitdrive("c:/dir")` retorna `("c:", "/dir")`

Si la ruta contiene una ruta UNC, *drive* contendrá el nombre de host y el recurso compartido, hasta el cuarto separador, pero sin incluirlo. p.ej. `splitdrive("//host/computer/dir")` retorna `("//host/computer", "/dir")`

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.splitext(path)`

Divide el nombre de ruta *path* en un par (*root*, *ext*) de tal forma que `root + ext == path`, y *ext* queda vacío o inicia con un punto y contiene a lo mucho un punto. Se ignoran los puntos iniciales del nombre base; `.splitext('.cshrc')` retorna `('cshrc', '')`.

Distinto en la versión 3.6: Acepta un *path-like object*.

`os.path.supports_unicode_filenames`

True si se pueden utilizar cadenas Unicode arbitrarias como nombres de archivo (dentro de las limitaciones impuestas por el sistema de archivos).

11.3 fileinput — Iterar sobre líneas de múltiples flujos de entrada

Código fuente: [Lib/fileinput.py](#)

Este módulo implementa una clase auxiliar y funciones para escribir rápidamente un bucle sobre una entrada estándar o una lista de archivos. Si solo quiere leer o escribir un archivo, vea `open()`.

El uso común es:

```
import fileinput
for line in fileinput.input():
    process(line)
```

Esto itera sobre las líneas de todos los archivos enumerados en `sys.argv[1:]`, por defecto a `sys.stdin` si la lista está vacía. Si un nombre de archivo es ```'-'`, también se reemplaza por `sys.stdin` y los argumentos opcionales *mode* y *openhook* se ignoran. Para especificar una lista alternativa de nombres de archivo, se pasa como primer argumento a `input()`. También se permite un único nombre de archivo.

Todos los archivos se abren en modo texto de manera predeterminada, pero puede anular esto especificando el parámetro *mode* en la llamada a `input()` o `FileInput`. Si se produce un error de E/S durante la apertura o lectura de un archivo, se lanza `OSError`.

Distinto en la versión 3.3: `IOError` solía ser lanzado; ahora es un alias de `OSError`.

Si `sys.stdin` se usa más de una vez, el segundo y siguientes usos no retornarán líneas, excepto tal vez para uso interactivo, o si se ha reiniciado explícitamente (por ejemplo, usando `sys.stdin.seek(0)`).

Los archivos vacíos se abren e inmediatamente se cierran; la única vez que su presencia en la lista de nombres de archivo es notable es cuando el último archivo abierto está vacío.

Las líneas se retornan con cualquier nueva línea intacta, lo que significa que la última línea en un archivo puede no tener una.

Puede controlar cómo se abren los archivos proporcionando un enlace de apertura a través del parámetro *openhook* a `fileinput.input()` o `FileInput()`. El enlace debe ser una función que tome dos argumentos, *filename* y *mode*, y retorna un objeto similar a un archivo abierto. Este módulo ya proporciona dos enlaces útiles.

La siguiente función es la interfaz principal de este módulo:

```
fileinput.input(files=None, inplace=False, backup="", *, mode='r', openhook=None)
```

Crea una instancia de la clase `FileInput`. La instancia se usará como estado global para las funciones de este módulo y también se volverá a usar durante la iteración. Los parámetros de esta función se pasarán al constructor de la clase `FileInput`.

La instancia `FileInput` se puede usar como gestor de contexto en la declaración `with`. En este ejemplo, `input` se cierra después de salir de la instrucción `with`, incluso si se produce una excepción:

```
with fileinput.input(files=('spam.txt', 'eggs.txt')) as f:
    for line in f:
        process(line)
```

Distinto en la versión 3.2: Se puede usar como gestor de contexto.

Distinto en la versión 3.8: Los parámetros de palabras clave *mode* y *openhook* ahora son solo palabras clave.

Las siguientes funciones utilizan el estado global creado por `fileinput.input()`; si no hay estado activo, es lanzado `RuntimeError`.

```
fileinput.filename()
```

Retorna el nombre del archivo que se está leyendo actualmente. Antes de leer la primera línea, retorna `None`.

```
fileinput.fileno()
```

Retorna el entero «file descriptor» para el archivo actual. Cuando no se abre ningún archivo (antes de la primera línea y entre archivos), retorna `-1`.

```
fileinput.lineno()
```

Retorna el número de línea acumulativa de la línea que se acaba de leer. Antes de que se haya leído la primera línea, retorna `0`. Después de leer la última línea del último archivo, retorna el número de línea de esa línea.

```
fileinput.filelineno()
```

Retorna el número de línea en el archivo actual. Antes de que se haya leído la primera línea, retorna `0`. Después de leer la última línea del último archivo, retorna el número de línea de esa línea dentro del archivo.

```
fileinput.isfirstline()
```

Retorna `True` si la línea que acaba de leer es la primera línea de su archivo; de lo contrario, retorna `False`.

```
fileinput.isstdin()
```

Retorna `True` si la última línea se leyó de `sys.stdin`, de lo contrario, retorna `False`.

```
fileinput.nextfile()
```

Cierra el archivo actual para que la próxima iteración lea la primera línea del siguiente archivo (si corresponde); las líneas no leídas del archivo no contarán para el recuento de líneas acumuladas. El nombre del archivo no se cambia hasta que se haya leído la primera línea del siguiente archivo. Antes de que se haya leído la primera línea, esta función no tiene efecto; no se puede usar para omitir el primer archivo. Después de leer la última línea del último archivo, esta función no tiene efecto.

```
fileinput.close()
```

Cierra la secuencia.

La clase que implementa el comportamiento de secuencia proporcionado por el módulo también está disponible para la subclasificación:

```
class fileinput.FileInput(files=None, inplace=False, backup="", *, mode='r', openhook=None)
```

La Clase `FileInput` es la implementación; sus métodos `filename()`, `fileno()`, `lineno()`, `filelineno()`, `isfirstline()`, `isstdin()`, `nextfile()` and `close()` corresponden a las funciones del mismo nombre en el módulo. Además tiene un método `readline()` que retorna la siguiente línea de entrada, y un método `__getitem__()` que implementa el comportamiento de secuencia. Se debe acceder a la secuencia en orden estrictamente secuencial; acceso aleatorio y `readline()` no se pueden mezclar.

Con *mode* puede especificar a qué modo de archivo se pasará `open()`. Debe ser uno de `'r'`, `'rU'`, `'U'` and `'rb'`.

El *openhook*, cuando se proporciona, debe ser una función que tome dos argumentos, *filename* y *mode*, y retorne un objeto similar a un archivo abierto en consecuencia. No puede usar *inplace* y *openhook* juntos.

Una instancia *FileInput* se puede usar como gestor de contexto en la instrucción `with`. En este ejemplo, *input* se cierra después de salir de la palabra clave `with`, incluso si se produce una excepción:

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

Distinto en la versión 3.2: Se puede usar como gestor de contexto.

Obsoleto desde la versión 3.4: Los modos `'rU'` and `'U'`.

Obsoleto desde la versión 3.8: Soporte para el método `__getitem__()` está discontinuado.

Distinto en la versión 3.8: El parámetro de palabra clave *mode* y *openhook* ahora son solo palabras clave.

Filtrado al instante opcional: si el argumento de la palabra clave `inplace=True` se pasa a *fileinput.input()* o al constructor *FileInput*, el archivo se mueve a una copia de seguridad y la salida estándar es dirigida al archivo de entrada (si ya existe un archivo con el mismo nombre que el archivo de copia de seguridad, se reemplazará en silencio). Esto hace posible escribir un filtro que reescribe su archivo de entrada en su lugar. Si se proporciona el parámetro *backup* (generalmente como `backup='.<some extension>'`), este especifica la extensión para el archivo de respaldo y el archivo de respaldo permanece; de forma predeterminada, la extensión es `' .bak '` y se elimina cuando se cierra el archivo de salida. El filtrado en el lugar se desactiva cuando se lee la entrada estándar.

Este módulo proporciona los dos enlaces de apertura siguientes:

`fileinput.hook_compressed(filename, mode)`

Abre de forma transparente archivos comprimidos con *gzip* y *bzip2* (reconocidos por las extensiones `' .gz '` and `' .bz2 '`) utilizando los módulos *gzip* y *bz2*. Si la extensión del nombre de archivo no es `' .gz '` or `' .bz2 '`, el archivo se abre normalmente (es decir, usando `open()` sin descompresión).

Ejemplo de uso: `fi = fileinput.FileInput(openhook=fileinput.hook_compressed)`

`fileinput.hook_encoded(encoding, errors=None)`

Retorna un enlace que abre cada archivo con `open()`, usando el *encoding* y *errors* dados para leer el archivo.

Ejemplo de uso: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))`

Distinto en la versión 3.6: Se agregó el parámetro opcional *errors*.

11.4 stat — Interpretación de los resultados de stat()

Código fuente: [Lib/stat.py](#)

El módulo *stat* define constantes y funciones para interpretar los resultados de `os.stat()`, `os.fstat()` y `os.lstat()` (si existen). Para obtener los detalles completos sobre las llamadas a `stat()`, `fstat()` y `lstat()`, consulta la documentación de tu sistema.

Distinto en la versión 3.4: El módulo *stat* se apoya en una implementación en C.

El módulo *stat* define las siguientes funciones para comprobar tipos de archivo específicos:

`stat.S_ISDIR(mode)`

Retorna un valor no nulo si el modo es de un directorio.

`stat.S_ISCHR(mode)`

Retorna un valor no nulo si el modo es de un archivo de un dispositivo especial de caracteres.

`stat.S_ISBLK(mode)`
Retorna un valor no nulo si el modo es de un archivo de un dispositivo especial de bloques.

`stat.S_ISREG(mode)`
Retorna un valor no nulo si el modo es de un archivo normal.

`stat.S_ISFIFO(mode)`
Retorna un valor no nulo si el modo es de un *FIFO* (tubería con nombre).

`stat.S_ISLNK(mode)`
Retorna un valor no nulo si el modo es de un enlace simbólico.

`stat.S_ISSOCK(mode)`
Retorna un valor no nulo si el modo es de un socket.

`stat.S_ISDOOR(mode)`
Retorna un valor no nulo si el modo es de un *door*.

Nuevo en la versión 3.4.

`stat.S_ISPORT(mode)`
Retorna un valor no nulo si el modo es de un *event port*.

Nuevo en la versión 3.4.

`stat.S_ISWHT(mode)`
Retorna un valor no nulo si el modo es de un *whiteout*.

Nuevo en la versión 3.4.

Se definen dos funciones adicionales para una manipulación más general del modo del archivo:

`stat.S_IMODE(mode)`
Retorna la porción del modo del archivo que puede ser establecida por `os.chmod()` — esto es, los bits de los permisos del archivo más los bits *sticky bit*, *set-group-id* y *set-user-id* (en los sistemas que lo soporten).

`stat.S_IFMT(mode)`
Retorna la porción del modo del archivo que describe el tipo de archivo (usado por las funciones `S_IS*` () de más arriba).

Normalmente se usarían las funciones `os.path.is*()` para comprobar el tipo de un archivo; estas funciones de aquí son útiles cuando se hacen múltiples comprobaciones sobre el mismo archivo y se desea evitar la sobrecarga causada por la llamada al sistema `stat()` en cada comprobación. También son útiles cuando se comprueba información de un archivo que no es gestionada por `os.path`, como buscar dispositivos de bloques o caracteres.

Ejemplo:

```
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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
def visitfile(file):  
    print('visiting', file)  
  
if __name__ == '__main__':  
    walktree(sys.argv[1], visitfile)
```

Se proporciona una función de utilidad adicional para convertir el modo del archivo en una cadena de caracteres legible por humanos:

`stat.filemode(mode)`

Convierte el modo del archivo a una cadena de caracteres de la forma “-rwxrwxrwx”.

Nuevo en la versión 3.3.

Distinto en la versión 3.4: La función soporta `S_IFDOOR`, `S_IFPORT` y `S_IFWHT`.

Todas las variables de debajo son simplemente índices simbólicos sobre la tupla de 10 elementos retornada por `os.stat()`, `os.fstat()` o `os.lstat()`.

`stat.ST_MODE`

Modo de protección del *inode*.

`stat.ST_INO`

Número del *inode*.

`stat.ST_DEV`

Dispositivo en el que reside el *inode*.

`stat.ST_NLINK`

Número de enlaces al *inode*.

`stat.ST_UID`

Id de usuario del propietario.

`stat.ST_GID`

Id del grupo del propietario.

`stat.ST_SIZE`

Tamaño en bytes de un archivo normal; cantidad de datos esperando en algunos archivos especiales.

`stat.ST_ATIME`

Momento del último acceso.

`stat.ST_MTIME`

Momento de la última modificación.

`stat.ST_CTIME`

El «ctime» reportado por el sistema operativo. En algunos sistemas (como Unix) es el momento del último cambio en los metadatos, y en otros (como Windows), es el momento de creación (véase la documentación de la plataforma para más detalles).

La interpretación de «tamaño de archivo» cambia dependiendo del tipo de archivo. Para archivos normales es el tamaño del archivo en bytes. En la mayoría de sistemas Unix (incluyendo Linux en particular), para *FIFOs* y sockets es el número de bytes que esperan ser leídos en el momento de la llamada a `os.stat()`, `os.fstat()`, o `os.lstat()`; en ocasiones, esto puede ser útil, especialmente para sondear uno de estos archivos especiales después de una apertura no bloqueante. Para otros dispositivos de caracteres y bloques el significado del campo *size* es más variado, dependiendo de la implementación de la llamada al sistema subyacente.

Las variables de debajo definen los flags usados en el campo `ST_MODE`.

El uso de las funciones de arriba es más portable que el uso del primer juego de flags:

`stat.S_IFSOCK`

Socket.

`stat.S_IFLNK`

Enlace simbólico.

`stat.S_IFREG`
Archivo normal.

`stat.S_IFBLK`
Dispositivo de bloques.

`stat.S_IFDIR`
Directorio.

`stat.S_IFCHR`
Dispositivo de caracteres.

`stat.S_IFIFO`
FIFO.

`stat.S_IFDOOR`
Door.

Nuevo en la versión 3.4.

`stat.S_IFPORT`
Event port.

Nuevo en la versión 3.4.

`stat.S_IFWHT`
Whiteout.

Nuevo en la versión 3.4.

Nota: `S_IFDOOR`, `S_IFPORT` o `S_IFWHT` se definen como 0 cuando la plataforma no soporta los tipos de archivo.

Los siguientes flags también pueden usarse en el argumento *mode* de `os.chmod()`:

`stat.S_ISUID`
Establecer el bit *UID*.

`stat.S_ISGID`
Bit *Set-group-ID*. Este bit tiene varios usos especiales. Para un directorio indica que la semántica BSD debe usarse para ese directorio: los archivos creados ahí heredan el *ID* de grupo del directorio, no del *ID* de grupo efectivo del proceso que los crea, y los directorios creados ahí también tendrán activado el bit `S_ISGID`. Para un archivo que no tiene activado el bit de ejecución de grupo (`S_IXGRP`), el bit *Set-group-ID* indica el bloqueo obligatorio del archivo/registro (véase también `S_ENFMT`).

`stat.S_ISVTX`
Sticky bit. Cuando este bit está activado en un directorio, significa que un archivo dentro de ese directorio puede ser renombrado o borrado sólo por el propietario del archivo, por el propietario del directorio, o por un proceso con privilegios.

`stat.S_IRWXU`
Máscara para los permisos del propietario del archivo.

`stat.S_IRUSR`
El propietario tiene permiso de lectura.

`stat.S_IWUSR`
El propietario tiene permiso de escritura.

`stat.S_IXUSR`
El propietario tiene permiso de ejecución.

`stat.S_IRWXG`
Máscara para los permisos del grupo.

`stat.S_IRGRP`
El grupo tiene permiso de lectura.

`stat.S_IWGRP`

El grupo tiene permiso de escritura.

`stat.S_IXGRP`

El grupo tiene permiso de ejecución.

`stat.S_IRWXO`

Máscara para permisos de los otros (no en el grupo).

`stat.S_IROTH`

Los otros tienen permiso de lectura.

`stat.S_IWOTH`

Los otros tienen permiso de escritura.

`stat.S_IXOTH`

Los otros tienen permiso de ejecución.

`stat.S_ENFMT`

Imposición del bloqueo de archivos de System V. Este flag se comparte con `S_ISGID`: se impone el bloqueo de archivos/registros en archivos que no tengan activado el bit de ejecución por el grupo (`S_IXGRP`).

`stat.S_IREAD`

Sinónimo de `S_IRUSR` en Unix V7.

`stat.S_IWRITE`

Sinónimo de `S_IWUSR` en Unix V7.

`stat.S_IEXEC`

Sinónimo de `S_IXUSR` en Unix V7.

Los siguientes flags pueden usarse como argumento *flags* de `os.chflags()`:

`stat.UF_NODUMP`

No volcar el archivo.

`stat.UF_IMMUTABLE`

El archivo no puede ser modificado.

`stat.UF_APPEND`

Sólo se puede añadir al archivo.

`stat.UF_OPAQUE`

El directorio es opaco cuando se mira a través de un *union stack*.

`stat.UF_NOUNLINK`

El archivo no puede ser renombrado o borrado.

`stat.UF_COMPRESSED`

El archivo se almacena comprimido (Mac OS X 10.6+).

`stat.UF_HIDDEN`

El archivo no debe ser mostrado en una *GUI* (Mac OS X 10.5+).

`stat.SF_ARCHIVED`

El archivo puede ser archivado.

`stat.SF_IMMUTABLE`

El archivo no puede ser modificado.

`stat.SF_APPEND`

Sólo se puede añadir al archivo.

`stat.SF_NOUNLINK`

El archivo no puede ser renombrado o borrado.

`stat.SF_SNAPSHOT`

El archivo es una instantánea.

Véase la página de *man chflags (2)* de los sistemas *BSD o Mac OS para más información.

En Windows, las siguientes constantes de atributos de fichero están disponibles para ser usadas al comprobar los bits del miembro `st_file_attributes` retornado por `os.stat()`. Véase [Windows API documentation](#) para más detalles sobre el significado de estas constantes.

```
stat.FILE_ATTRIBUTE_ARCHIVE
stat.FILE_ATTRIBUTE_COMPRESSED
stat.FILE_ATTRIBUTE_DEVICE
stat.FILE_ATTRIBUTE_DIRECTORY
stat.FILE_ATTRIBUTE_ENCRYPTED
stat.FILE_ATTRIBUTE_HIDDEN
stat.FILE_ATTRIBUTE_INTEGRITY_STREAM
stat.FILE_ATTRIBUTE_NORMAL
stat.FILE_ATTRIBUTE_NOT_CONTENT_INDEXED
stat.FILE_ATTRIBUTE_NO_SCRUB_DATA
stat.FILE_ATTRIBUTE_OFFLINE
stat.FILE_ATTRIBUTE_READONLY
stat.FILE_ATTRIBUTE_REPARSE_POINT
stat.FILE_ATTRIBUTE_SPARSE_FILE
stat.FILE_ATTRIBUTE_SYSTEM
stat.FILE_ATTRIBUTE_TEMPORARY
stat.FILE_ATTRIBUTE_VIRTUAL
```

Nuevo en la versión 3.5.

En Windows, las siguientes constantes están disponibles para la comparación con el miembro `st_reparse_tag` retornado por `os.lstat()`. Estas constantes son muy conocidas, pero no se trata de una lista exhaustiva.

```
stat.IO_REPARSE_TAG_SYMLINK
stat.IO_REPARSE_TAG_MOUNT_POINT
stat.IO_REPARSE_TAG_APPEXECLINK
```

Nuevo en la versión 3.8.

11.5 filecmp— Comparaciones de Archivo y Directorio

Código fuente: `Lib/filecmp.py`

El módulo `filecmp` define funciones para comparar ficheros y directorios, con varias compensaciones de tiempo/corrección. Para comparar ficheros, vea también el módulo `difflib`.

El módulo `filecmp` define las siguientes funciones:

`filecmp.cmp(f1, f2, shallow=True)`

Compara los ficheros llamados `f1` y `f2`, retornando `True` si son iguales, `False` en caso contrario.

Si `shallow` es verdadero, ficheros con firmas idénticas `os.stat()` se consideran iguales. En caso contrario, el contenido de los ficheros es comparado.

Note que ningún programa externo es llamado desde esta función, dándole portabilidad y eficiencia.

La función utiliza un caché para comparaciones pasadas y los resultados, con entradas de caché invalidadas si la información de `os.stat()` para el fichero cambia. El caché entero puede ser limpiado utilizando `clear_cache()`.

`filecmp.cmpfiles(dir1, dir2, common, shallow=True)`

Compara los ficheros en los dos directorios `dir1` y `dir2` cuyos nombres son dados por `common`.

Retorna tres listas de nombres de fichero: `match`, `mismatch`, `errors`. `match` contiene la lista de ficheros que coinciden, `mismatch` contiene los nombres de aquellos que no, y `errors` lista los nombres de los ficheros que no deberían ser comparados. Los ficheros son listados en `errors` si no existen en uno de los directorios, el usuario carece de permisos para leerlos o si la comparación no puede hacerse por alguna razón.

El parámetro *shallow* tiene el mismo significado y valor por defecto en cuanto a `filecmp.cmp()`.

Por ejemplo, `cmpfiles('a', 'b', ['c', 'd/e'])` comparará `a/c` con `b/c` y `a/d/e` con `b/d/e`. `'c'` y `'d/e'` estará cada uno en una de las tres listas retornadas.

`filecmp.clear_cache()`

Limpia el caché de `filecmp`. Esto podría ser útil si un fichero es comparado muy rápido después de que es modificado que está dentro de la resolución mtime del sistema de archivos subyacente.

Nuevo en la versión 3.4.

11.5.1 La clase `dircmp`

class `filecmp.dircmp`(*a*, *b*, *ignore=None*, *hide=None*)

Construye un nuevo objeto de comparación de directorio, para comparar los directorios *a* y *b*. *ignore* es una lista de nombres a ignorar, y predetermina a `filecmp.DEFAULT_IGNORES`. *hide* es una lista de nombres a ocultar, y predetermina a `[os.curdir, os.pardir]`.

La clase `dircmp` compara ficheros haciendo comparaciones *shallow* como está descrito en `filecmp.cmp()`.

La clase `dircmp` provee los siguientes métodos:

report()

Imprime (a `sys.stdout`) una comparación entre *a* y *b*.

report_partial_closure()

Imprime una comparación entre *a* y *b* y subdirectorios inmediatos comunes.

report_full_closure()

Imprime una comparación entre *a* y *b* y subdirectorios comunes (recursivamente).

La clase `dircmp` ofrece un número de atributos interesantes que pueden ser utilizados para obtener varios bits de información sobre los árboles de directorio que están siendo comparados.

Note que vía los hooks `__getattr__()`, todos los atributos son perezosamente computados, así que no hay penalización de velocidad si sólo esos atributos que son ligeros de computar son utilizados.

left

El directorio *a*.

right

El directorio *b*.

left_list

Ficheros y subdirectorios en *a*, filtrados por *hide* e *ignore*.

right_list

Ficheros y subdirectorios en *b*, filtrados por *hide* e *ignore*.

common

Ficheros y subdirectorios en *a* y *b*.

left_only

Ficheros y subdirectorios sólo en *a*.

right_only

Ficheros y subdirectorios sólo en *b*.

common_dirs

Subdirectorios en *a* y *b*.

common_files

Ficheros en *a* y *b*.

common_funny

Nombres en *a* y *b*, de forma que el tipo difiera entre los directorios, o los nombres por los que `os.stat()` reporta un error.

same_files

Ficheros que son idénticos en *a* y *b*, utilizando el operador de comparación de fichero de la clase.

diff_files

Ficheros que están en *a* y *b*, cuyos contenidos difieren acorde al operador de comparación del fichero de clase.

funny_files

Ficheros que están en *a* y *b*, pero no deberían ser comparados.

subdirs

Un diccionario mapeando nombres en objetos de `common_dirs` a `dircmp`.

filecmp.DEFAULT_IGNORES

Nuevo en la versión 3.4.

Lista de directorios ignorados por `dircmp` por defecto.

Aquí hay un ejemplo simplificado de uso del atributo `subdirs` para buscar recursivamente a través de dos directorios para mostrar diferentes ficheros comunes:

```
>>> 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)
```

11.6 tempfile — Generar archivos y directorios temporales

Código fuente: `Lib/tempfile.py`

Este módulo crea archivos y directorios temporales. Funciona en todas las plataformas soportadas. `TemporaryFile`, `NamedTemporaryFile`, `TemporaryDirectory`, y `SpooledTemporaryFile` son interfaces de alto nivel que proporcionan limpieza automática y se pueden utilizar como administradores de contexto. `mkstemp()` y `mkdtemp()` son funciones de nivel inferior que requieren limpieza manual.

Todas las funciones y constructores invocables por el usuario toman argumentos adicionales que permiten el control directo sobre la ubicación y el nombre de los archivos y directorios temporales. Los nombres de archivos utilizados por este módulo incluyen una cadena de caracteres aleatorios que permite que esos archivos se creen de forma segura en directorios temporales compartidos. Para mantener la compatibilidad con versiones anteriores, el orden de argumentos es algo extraño; se recomienda utilizar argumentos nombrados para mayor claridad.

El módulo define los siguientes elementos invocables por el usuario:

`tempfile.TemporaryFile(mode='w+b', buffering=-1, encoding=None, newline=None, suffix=None, prefix=None, dir=None, *, errors=None)`

Retorna un *file-like object* que se puede usar como área de almacenamiento temporal. El archivo se crea de forma segura, usando las mismas reglas que `mkstemp()`. Este se destruirá tan pronto como se cierre (incluido un cierre implícito cuando el objeto es recolectado como basura). Bajo Unix, la entrada de directorio para el archivo no se crea en lo absoluto o se elimina inmediatamente después de crear el archivo. Otras plataformas no soportan esto; tu código no debería depender en un archivo temporal creado con esta función teniendo o no un nombre visible en el sistema de archivos.

El objeto resultante puede usarse como un administrador de contexto (ver *Ejemplos*). Al completar el contexto o la destrucción del objeto de archivo, el archivo temporal se eliminará del sistema de archivos.

El parámetro *mode* por defecto es `'w+b'` para que el archivo creado pueda leerse y escribirse sin cerrarse. El modo binario se utiliza para que se comporte consistentemente en todas las plataformas sin tener en cuenta los datos que se almacenan. *buffering*, *encoding*, *errors* y *newline* se interpretan como en `open()`.

Los parámetros *dir*, *prefix* y *suffix* tienen el mismo significado y valores predeterminados de `mkstemp()`.

El objeto retornado es un objeto de archivo verdadero en las plataformas POSIX. En otras plataformas, es un objeto similar a un archivo cuyo atributo `file` es el objeto de archivo verdadero subyacente.

El indicador `os.O_TMPFILE` se usa si está disponible (específico de Linux, requiere el kernel de Linux 3.11 o posterior).

Lanza un *evento de auditoria* `tempfile.mkstemp` con argumento `fullpath`.

Distinto en la versión 3.5: El indicador `os.O_TMPFILE` ahora se usa si está disponible.

Distinto en la versión 3.8: Se agregó el parámetro *errors*.

`tempfile.NamedTemporaryFile(mode='w+b', buffering=-1, encoding=None, newline=None, suffix=None, prefix=None, dir=None, delete=True, *, errors=None)`

Esta función opera exactamente como lo hace `TemporaryFile()`, excepto que el archivo está garantizado para tener un nombre visible en el sistema de archivos (en Unix, el directorio de entrada no está desvinculado). El nombre se puede obtener del atributo `name` del objeto tipo archivo retornado. Aunque el nombre se puede usar para abrir el archivo por segunda vez, mientras el archivo temporal nombrado sigue abierto, esto varía según las plataformas (se puede usar en Unix; no se puede en Windows NT o posteriores). Si *delete* es verdadero (por defecto), el archivo se elimina tan pronto como se cierra. El objeto retornado siempre es un objeto similar a un archivo cuyo atributo `file` es el objeto de archivo verdadero subyacente. Este objeto similar a un archivo se puede usar con una sentencia `with`, al igual que un archivo normal.

Lanza un *evento de auditoria* `tempfile.mkstemp` con argumento `fullpath`.

Distinto en la versión 3.8: Se agregó el parámetro *errors*.

`tempfile.SpooledTemporaryFile(max_size=0, mode='w+b', buffering=-1, encoding=None, newline=None, suffix=None, prefix=None, dir=None, *, errors=None)`

Esta función opera exactamente como lo hace `TemporaryFile()`, excepto que los datos se almacenan en la memoria hasta que el tamaño del archivo excede *max_size*, o hasta que el método del archivo `fileno()` se llama, en ese momento los contenidos se escriben en el disco y la operación continúa como con `TemporaryFile()`.

El archivo resultante tiene un método adicional, `rollover()`, que hace que el archivo se transfiera a un archivo en el disco, independientemente de su tamaño.

El objeto retornado es un objeto tipo archivo cuyo atributo `_file` es un objeto `io.BytesIO` o `io.TextIOWrapper` (dependiendo de si se especificó binario o texto *mode*) o un objeto de archivo verdadero, dependiendo de si: se ha llamado a `rollover()`. Este objeto similar a un archivo se puede usar en una sentencia `with`, al igual que un archivo normal.

Distinto en la versión 3.3: el método *truncate* ahora acepta un argumento *size*.

Distinto en la versión 3.8: Se agregó el parámetro *errors*.

`tempfile.TemporaryDirectory(suffix=None, prefix=None, dir=None)`

Esta función crea de forma segura un directorio temporal utilizando las mismas reglas que `mkdtemp()`. El objeto resultante puede usarse como administrador de contexto (ver *Ejemplos*). Al finalizar el contexto o la destrucción del objeto de directorio temporal, el directorio temporal recién creado y todo su contenido se eliminan del sistema de archivos.

El nombre del directorio se puede obtener del atributo `name` del objeto retornado. Cuando el objeto retornado se usa como administrador de contexto, el `name` se asignará al objetivo de la cláusula *as* en la sentencia `with`, si hay una.

El directorio se puede limpiar explícitamente llamando al método `cleanup()`.

Lanza un *evento de auditoria* `tempfile.mkdtemp` con argumento `fullpath`.

Nuevo en la versión 3.2.

`tempfile.mkstemp(suffix=None, prefix=None, 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.

A diferencia de `TemporaryFile()`, el usuario de `mkstemp()` es responsable de eliminar el archivo temporal cuando haya terminado con él.

Si `suffix` no es `None`, el nombre del archivo terminará con ese sufijo, de lo contrario no habrá sufijo. `mkstemp()` no pone un punto entre el nombre del archivo y el sufijo; si necesita uno, póngalo al comienzo del `suffix`.

Si `prefix` no es `None`, el nombre del archivo comenzará con ese prefijo; de lo contrario, se usa un prefijo predeterminado. El valor predeterminado es el valor de retorno de `gettempprefix()` o `gettempprefixb()`, según corresponda.

Si `dir` no es `None`, el archivo se creará en ese directorio; de lo contrario, se usa un directorio predeterminado. El directorio predeterminado se elige de una lista dependiente de la plataforma, pero el usuario de la aplicación puede controlar la ubicación del directorio configurando las variables de entorno `TMPDIR`, `TEMP` o `TMP`. Por lo tanto, no hay garantía de que el nombre de archivo generado tenga buenas propiedades, como no requerir comillas cuando se pasa a comandos externos a través de `os.popen()`.

Si alguno de los argumentos `suffix`, `prefix`, y `dir` no son `None`, estos deben ser del mismo tipo. Si son bytes, el nombre retornado será bytes en lugar de str. Si desea forzar un valor de retorno de bytes con un comportamiento predeterminado, pase `suffix=b''`.

If `text` is specified and true, the file is opened in text mode. Otherwise, (the default) the file is opened in binary mode.

`mkstemp()` retorna una tupla que contiene un controlador de nivel de sistema operativo a un archivo abierto (como sería retornado por `os.open()`) y la ruta absoluta de ese archivo, en ese orden.

Lanza un *evento de auditoria* `tempfile.mkstemp` con argumento `fullpath`.

Distinto en la versión 3.5: `suffix`, `prefix`, y `dir` ahora se pueden suministrar en bytes para obtener el valor de retorno en bytes. Antes de esto, solo str se permitía. `suffix` y `prefix` ahora aceptan y por defecto `None` para hacer que se use un valor predeterminado apropiado.

Distinto en la versión 3.6: El parámetro `dir` ahora acepta un *path-like object*.

`tempfile.mkdtemp(suffix=None, prefix=None, dir=None)`

Crea un directorio temporal de la manera más segura posible. No hay condiciones de carrera en la creación del directorio. El directorio se puede leer, escribir y buscar solo por el ID del usuario creador.

El usuario de `mkdtemp()` es responsable de eliminar el directorio temporal y su contenido cuando haya terminado con él.

Los argumentos `prefix`, `suffix`, y `dir` son los mismos que para `mkstemp()`.

`mkdtemp()` retorna la ruta absoluta del nuevo directorio.

Lanza un *evento de auditoria* `tempfile.mkdtemp` con argumento `fullpath`.

Distinto en la versión 3.5: `suffix`, `prefix`, y `dir` ahora se pueden suministrar en bytes para obtener el valor de retorno en bytes. Antes de esto, solo str se permitía. `suffix` y `prefix` ahora aceptan y por defecto `None` para hacer que se use un valor predeterminado apropiado.

Distinto en la versión 3.6: El parámetro `dir` ahora acepta un *path-like object*.

`tempfile.gettempdir()`

Retorna el nombre del directorio utilizado para archivos temporales. Esto define el valor predeterminado para el argumento `dir` para todas las funciones en este módulo.

Python busca en una lista estándar de directorios para encontrar uno dentro del cual el usuario pueda crear archivos. La lista es:

1. El directorio nombrado por la variable de entorno `TMPDIR`.
2. El directorio nombrado por la variable de entorno `TEMP`.
3. El directorio nombrado por la variable de entorno `TMP`.
4. Una ubicación específica de la plataforma:
 - En Windows, los directorios `C:\TEMP`, `C:\TMP`, `\TEMP`, y `\TMP`, en ese orden.
 - En todas las otras plataformas, los directorios `/tmp`, `/var/tmp`, y `/usr/tmp`, en ese orden.
5. Y como última alternativa, el directorio de trabajo actual.

El resultado de la búsqueda es cacheada, vea la descripción de `tempdir` abajo.

`tempfile.gettempdir()`

Igual a `gettempdir()` pero el valor retornado es en bytes.

Nuevo en la versión 3.5.

`tempfile.gettempprefix()`

Retorna el prefijo del nombre de archivo utilizado para crear archivos temporales. Este no contiene el componente de directorio.

`tempfile.gettempprefixb()`

Igual que `gettempprefix()` pero el valor retornado es en bytes.

Nuevo en la versión 3.5.

El módulo utiliza una variable global para almacenar el nombre del directorio utilizado para los archivos temporales retornados por `gettempdir()`. Se puede configurar directamente para anular el proceso de selección, pero esto no se recomienda. Todas las funciones en este módulo toman un argumento `dir` que puede usarse para especificar el directorio y este es el enfoque recomendado.

`tempfile.tempdir`

Cuando se establece en un valor distinto de `None`, esta variable define el valor predeterminado para el argumento `dir` para las funciones definidas en este módulo.

Si `tempdir` es `None` (por defecto) en cualquier llamada a cualquiera de las funciones anteriores excepto `gettempprefix()` se inicializa siguiendo el algoritmo descrito en `gettempdir()`.

11.6.1 Ejemplos

Estos son algunos ejemplos del uso típico del módulo `tempfile`:

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# file is now closed and removed
```

(continué en la próxima página)

(proviene de la página anterior)

```
# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

11.6.2 Funciones y variables deprecadas

Una forma histórica de crear archivos temporales era generar primero un nombre de archivo con la función `mktemp()` y luego crear un archivo con este nombre. Desafortunadamente, esto no es seguro, porque un proceso diferente puede crear un archivo con este nombre en el tiempo entre la llamada a `mktemp()` y el intento posterior de crear el archivo mediante el primer proceso. La solución es combinar los dos pasos y crear el archivo de inmediato. Este enfoque es utilizado por `mkstemp()` y las otras funciones descritas anteriormente.

`tempfile.mktemp(suffix="", prefix='tmp', dir=None)`

Obsoleto desde la versión 2.3: Utilice `mkstemp()` en su lugar.

Retorna el nombre de la ruta absoluta de un archivo que no existía en el momento en que se realiza la llamada. Los argumentos `prefix`, `suffix` y `dir` son similares a los de `mkstemp()`, excepto los nombres de archivo de bytes, `suffix=None` y `prefix=None` no son soportados.

Advertencia: El uso de esta función puede introducir un agujero de seguridad en su programa. Para cuando llegues a hacer algo con el nombre de archivo que retorna, alguien más pudo haberse adelantado. El uso de `mktemp()` se puede reemplazar fácilmente con `NamedTemporaryFile()`, pasándole el parámetro `delete=False`:

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmpjtjujtt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

11.7 glob — Expansión del patrón de nombres de ruta de estilo Unix

Código fuente: [Lib/glob.py](#)

El módulo `glob` encuentra todos los nombres de rutas que se asemejan a un patrón especificado de acuerdo a las reglas que se siguen en una terminal Unix, aunque los resultados se retornan con un orden arbitrario. No se realiza expansión de virgulilla (*tilde* («~») *expansion* en inglés) pero `*`, `?` y caracteres de rango expresados mediante `[]` serán emparejados correctamente. Esto se realiza usando las funciones `os.scandir()` y `fnmatch.fnmatch()` de forma concertada sin invocar, realmente, a una *subshell*. Nótese que, a diferencia de `fnmatch.fnmatch()`, `glob` trata a los nombres de ficheros que comienzan con `(.)` como casos especiales. (Para la expansión de virgulilla o variable de terminal, usa `os.path.expanduser()` y `os.path.expandvars()`.)

Para una coincidencia literal, envuelve los meta-caracteres en corchetes. Por ejemplo, `'[?]'` empareja el carácter `'?'`.

Ver también:

El módulo `pathlib` ofrece objetos ruta de alto nivel.

`glob.glob(pathname, *, recursive=False)`

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). Whether or not the results are sorted depends on the file system. If a file that satisfies conditions is removed or added during the call of this function, whether a path name for that file be included is unspecified.

Si *recursive* es verdadero, el patrón «**» coincidirá con cualquier fichero y cero o más directorios, subdirectorios y enlaces simbólicos a directorios. Si el patrón va seguido de un `os.sep` o `os.altsep` los ficheros no coincidirán.

Lanza un *evento de auditoría* `glob.glob` con los argumentos `pathname`, `recursive`.

Nota: El uso del patrón «**» en árboles de directorios grandes podría consumir una cantidad de tiempo excesiva.

Distinto en la versión 3.5: Soporte para globs recursivos usando «**».

`glob.iglob(pathname, *, recursive=False)`

Retorna un *iterador* el cual produce los mismos valores que `glob()` sin necesidad de almacenarlos todos de forma simultánea.

Lanza un *evento de auditoría* `glob.glob` con los argumentos `pathname`, `recursive`.

`glob.escape(pathname)`

Escapa todos los caracteres especiales ('?', '*', y '['). Esto es útil si deseas coincidir una cadena de caracteres literal arbitraria que podría contener caracteres especiales. Los caracteres especiales en unidades compartidas/UNC no se eluden, e.g. en Windows `escape('///?/c:/Quo vadis?.txt')` retorna `'//?/c:/Quo vadis[?].txt'`.

Nuevo en la versión 3.4.

Por ejemplo, considera un directorio que contenga los siguientes ficheros: `1.gif`, `2.txt`, `card.gif` y un subdirectorio `sub` el cual contenga solo el fichero `3.txt`. `glob()` producirá los siguientes resultados. Nótese como se preservará cualquier componente inicial de la ruta.

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

Si un directorio contiene ficheros que comienzan con `.` no coincidirá por defecto. Por ejemplo, considera un directorio que contiene `card.gif` y `.card.gif`:

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

Ver también:

Módulo `fnmatch` Expansión de nombre de fichero (no de ruta) al estilo de la terminal

11.8 fnmatch — Coincidencia de patrones de nombre de archivos Unix

Código fuente: [Lib/fnmatch.py](#)

Este módulo proporciona soporte para comodines de estilo shell de Unix, que *no* son lo mismo que las expresiones regulares (que se documentan en el módulo [re](#)). Los caracteres especiales utilizados en los comodines de estilo shell son:

Patrón	Significado
*	coincide con todo
?	coincide con un solo carácter
[seq]	coincide con cualquier carácter presente en <i>seq</i>
[!seq]	coincide con cualquier carácter ausente en <i>seq</i>

Para una coincidencia literal, envuelva los meta-caracteres entre paréntesis. Por ejemplo, '[?]' coincide con el carácter '?'.
 Ten en cuenta que el separador de nombre de archivo ('/' en Unix) *no* es tratado de forma especial en este módulo. Consulta el módulo [glob](#) para realizar expansiones de nombre de ruta ([glob](#) usa [filter\(\)](#) para hacer coincidir con los componentes del nombre de ruta). Del mismo modo, los nombres de archivo que comienzan con un punto tampoco son tratados de forma especial por este módulo y se corresponden con los patrones * y ?.

`fnmatch.fnmatch(filename, pattern)`

Prueba si la cadena de caracteres *filename* coincide con la cadena *pattern*, retornando *True* o *False*. Ambos parámetros se normalizan entre mayúsculas y minúsculas usando [os.path.normcase\(\)](#). [fnmatchcase\(\)](#) se puede utilizar para realizar una comparación que distingue entre mayúsculas y minúsculas, independientemente de si es estándar para el sistema operativo.

Este ejemplo imprimirá todos los nombres de archivo en el directorio actual con la extensión `.txt`:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

`fnmatch.fnmatchcase(filename, pattern)`

Prueba si *filename* coincide con *pattern*, retornando *True* o *False*; la comparación distingue entre mayúsculas y minúsculas y no aplica [os.path.normcase\(\)](#).

`fnmatch.filter(names, pattern)`

Construct a list from those elements of the iterable *names* that match *pattern*. It is the same as `[n for n in names if fnmatch(n, pattern)]`, but implemented more efficiently.

`fnmatch.translate(pattern)`

Retorna el *pattern* estilo shell convertido a una expresión regular para usar con [re.match\(\)](#).

Ejemplo:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\.\.txt)\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<re.Match object; span=(0, 10), match='foobar.txt'>
```


Ver también:

Módulo `glob` Expansión de ruta estilo shell en Unix.

11.9 `linecache` — Acceso aleatorio a líneas de texto

Código fuente: `Lib/linecache.py`

El módulo `linecache` permite obtener cualquier línea de un archivo fuente Python, mientras se intenta optimizar internamente, usando una caché, el caso común en el que se leen muchas líneas de un solo archivo. Esto es utilizado por el módulo `traceback` para recuperar líneas de código fuente para incluirlas en el seguimiento de pila formateado.

La función `tokenize.open()` se utiliza para abrir archivos. Esta función usa `tokenize.detect_encoding()` para obtener la codificación del archivo; en la ausencia de un token de codificación, la codificación del archivo es por defecto UTF-8.

El módulo `linecache` define las siguientes funciones:

`linecache.getline(filename, lineno, module_globals=None)`

Obtiene la línea `lineno` del archivo llamado `filename`. Esta función nunca lanzará una excepción — retornará `' '` en los errores (el carácter de la nueva línea de terminación se incluirá para las líneas que se encuentren).

Si un archivo llamado `filename` no se encuentra, la función primero verifica un `__loader__` en `module_globals` [PEP 302](#). Si existe tal cargador y define un método `get_source`, entonces eso determina las líneas de origen (si `get_source()` retorna `None`, entonces se retorna `' '`). Finalmente, si `filename` es un nombre de fichero relativo, se busca en relación a las entradas en la ruta de búsqueda del módulo, `sys.path`.

`linecache.clearcache()`

Borra el caché. Use esta función si ya no necesita las líneas de archivos leídos previamente usando `getline()`.

`linecache.checkcache(filename=None)`

Comprueba la validez de la caché. Use esta función si los archivos de la caché pueden haber cambiado en disco y necesita la versión actualizada. Si se omite `filename`, comprobará todas las entradas en la caché.

`linecache.lazycache(filename, module_globals)`

Captura suficientes detalles sobre un módulo no basado en archivos para permitir obtener sus líneas más tarde a través de `getline()` incluso si `module_globals` es `None` en la llamada posterior. Esto evita hacer E/S hasta que una línea es realmente necesaria, sin tener que llevar los módulos globales indefinidamente.

Nuevo en la versión 3.5.

Ejemplo:

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.10 `shutil` — Operaciones de archivos de alto nivel

Código fuente: `Lib/shutil.py`

El módulo `shutil` ofrece varias operaciones de alto nivel en archivos y colecciones de archivos. En particular, provee funciones que dan soporte a la copia y remoción de archivos. Para operaciones en archivos individuales, véase también el módulo `os`.

Advertencia: Incluso las funciones de copia de archivos de nivel superior (`shutil.copy()`, `shutil.copy2()`) no pueden copiar todos los metadatos del archivo.

En las plataformas POSIX, esto significa que tanto el propietario como el grupo del archivo se pierden, al igual que las ACLs (*access control lists* o lista de control de acceso). En Mac OS, la bifurcación (*fork*) de recursos y de otros metadatos no se utilizan. Esto quiere decir que los recursos se perderán y que el tipo de archivo y los códigos de creador no serán correctos. En Windows, los propietarios de archivos, las ACLs y secuencias de datos alternativas no se copian.

11.10.1 Operaciones de directorios y archivos

`shutil.copyfileobj(fsrc, fdst[, length])`

Copia los contenidos del objeto de tipo archivo *fsrc* en el objeto *fdst*. El valor entero *length*, si está indicado, es el tamaño del búfer. En particular, un valor *length* negativo significa copiar los datos sin recorrer los datos de origen en fragmentos; los datos se leen en fragmentos en forma predeterminada para evitar el consumo de memoria incontrolado. Nótese que si la posición actual del archivo del objeto *fsrc* es distinto de cero, solo se copiarán el contenido desde la posición actual del archivo hasta el final.

`shutil.copyfile(src, dst, *, follow_symlinks=True)`

Copia los contenidos (sin metadatos) del archivo *src* en un archivo denominado *dst* y retorna *dst* de la manera más eficaz posible. *src* y *dst* son objetos de tipo ruta o nombres de ruta dados como cadenas de caracteres.

dst debe ser el nombre completo del archivo destino; véase `shutil.copy()` para una copia que acepta una ruta de directorio destino. Si *src* y *dst* refieren al mismo archivo, se lanza `SameFileError`.

El local de destino debe tener permisos de escritura; de lo contrario, se genera una excepción `OSError`. Si *dst* ya existe, se reemplazará. Los archivos especiales, como los dispositivos de caracteres o de bloques y *pipes*, no se pueden copiar con esta función.

Si *follow_symlinks* es falso y *src* es un enlace simbólico, un enlace simbólico nuevo se creará en lugar de copiar el archivo al que *src* apunta.

Se genera un *evento de auditoría* `shutil.copyfile` con argumentos *src*, *dst*.

Distinto en la versión 3.3: Solía generarse `IOError` en lugar de `OSError`. Se añadió el argumento *follow_symlinks*. Ahora retorna *dst*.

Distinto en la versión 3.4: Genera `SameFileError` en lugar de `Error`. Dado que la primera es una subclase de la segunda, el cambio es compatible con versiones anteriores.

Distinto en la versión 3.8: Los llamadas al sistema de copia rápida específicos de la plataforma se pueden usar internamente para copiar el archivo de manera más eficiente. Véase la sección *Operaciones de copia eficientes dependientes de la plataforma*.

exception `shutil.SameFileError`

Esta excepción se genera si el origen y destino en `copyfile()` son el mismo archivo.

Nuevo en la versión 3.4.

`shutil.copymode(src, dst, *, follow_symlinks=True)`

Copia los bits de permiso de *src* a *dst*. Los contenidos, el propietario y el grupo no se ven afectados. *src* y *dst* son objetos tipo ruta o nombres de ruta dados como cadenas de caracteres. Si *follow_symlinks* es falso, y tanto *src* como *dst* son enlaces simbólicos, `copymode()` intentará modificar el modo de *dst* (y no el archivo al que apunta). Esta funcionalidad no está disponible en todas las plataformas; véase `copystat()` para mayor información. Si `copymode()` no puede modificar los enlaces simbólicos de la plataforma local y se le solicita hacerlo, no hará nada y retornará.

Genera un *evento de auditoría* `shutil.copymode` con argumentos *src*, *dst*.

Distinto en la versión 3.3: Se añadió el argumento *follow_symlinks*.

`shutil.copystat(src, dst, *, follow_symlinks=True)`

Copia los bits de permiso, la última hora de acceso, la última hora de modificación y las *flags* desde *src* a *dst*.

En Linux, `copystat()` también copia los «atributos extendidos» siempre que sea posible. Los contenidos, el propietario y el grupo del archivo no se ven afectados. `src` y `dst` son objetos de tipo a ruta o nombres de ruta dados como cadenas de caracteres.

Si `follow_symlinks` es falso, y `src` y `dst` hacen referencia los enlaces simbólicos, `copystat()` funcionará con los enlaces simbólicos en lugar de en los archivos a los que estos se refieren: leer la información del enlace simbólico `src` y escribirla en el enlace simbólico `dst`.

Nota: No todas las plataformas proporcionan la capacidad de examinar y modificar enlaces simbólicos. Python puede indicarte qué funcionalidad está disponible localmente.

- Si `os.chmod` in `os.supports_follow_symlinks` es True, `copystat()` puede modificar los bits de permiso de un enlace simbólico.
- Si `os.utime` in `os.supports_follow_symlinks` es True, `copystat()` puede modificar el último acceso y las veces que un enlace simbólico fue modificado.
- Si `os.chflags` in `os.supports_follow_symlinks` es True, `copystat()` puede modificar las flags de un enlace simbólico. (`os.chflags` no está disponible en todas las plataformas.)

En plataformas donde parte o toda esta funcionalidad no está disponible, cuando se le pida modificar un enlace simbólico, `copystat()` copiará todo lo que pueda. `copystat()` nunca retorna un error.

Véase `os.supports_follow_symlinks` para más información.

Genera un *evento de auditoría* `shutil.copystat` con argumentos `src`, `dst`.

Distinto en la versión 3.3: Se ha añadido el argumento `follow_symlinks` y el soporte para atributos extendidos de Linux.

`shutil.copy(src, dst, *, follow_symlinks=True)`

Copies the file `src` to the file or directory `dst`. `src` and `dst` should be *path-like objects* or strings. If `dst` specifies a directory, the file will be copied into `dst` using the base filename from `src`. Returns the path to the newly created file.

Si `follow_symlinks` es falso, y `src` es un enlace simbólico, `dst` se creará como enlace simbólico. Si `follow_symlinks` es verdadero y `src` es un enlace simbólico, `dst` será una copia del archivo al que `src` hace referencia.

`copy()` copia los datos del archivo y el modo de permiso del archivo (véase `os.chmod()`). Otros metadatos, como los tiempos de creación y modificación de archivos, no se preservan. Para preservar todos los metadatos del archivo, usa `copy2()` en su lugar.

Se genera un *evento de auditoría* `shutil.copyfile` con argumentos `src`, `dst`.

Genera un *evento de auditoría* `shutil.copymode` con argumentos `src`, `dst`.

Distinto en la versión 3.3: Se ha añadido el argumento `follow_symlinks`. Ahora retorna la ruta de acceso al archivo recién creado.

Distinto en la versión 3.8: Los llamadas al sistema de copia rápida específicos de la plataforma se pueden usar internamente para copiar el archivo de manera más eficiente. Véase la sección *Operaciones de copia eficientes dependientes de la plataforma*.

`shutil.copy2(src, dst, *, follow_symlinks=True)`

Idéntico a `copy()`, excepto que `copy2()` también intenta conservar los metadatos del archivo.

Cuando `follow_symlinks` es falso, y `src` es un enlace simbólico, `copy2()` intenta copiar todos los metadatos del enlace simbólico `src` en el enlace simbólico `dst` recién creado. Sin embargo, esta funcionalidad no está disponible en todas las plataformas. En las plataformas donde parte o toda esta funcionalidad no está disponible, `copy2()` conservará todos los metadatos que pueda; `copy2()` nunca genera una excepción porque no puede conservar los metadatos del archivo.

`copy2()` usa `copystat()` para copiar todos los metadatos del archivo. Véase `copystat()` para más información sobre la compatibilidad con la plataforma para modificar los metadatos del enlace simbólico.

Se genera un *evento de auditoría* `shutil.copyfile` con argumentos `src`, `dst`.

Genera un *evento de auditoría* `shutil.copystat` con argumentos `src`, `dst`.

Distinto en la versión 3.3: Añadido el argumento `follow_symlinks`, intenta copiar también los atributos del sistema de archivos extendidos (actualmente solo en Linux). Ahora retorna la ruta de acceso al archivo recién creado.

Distinto en la versión 3.8: Los llamadas al sistema de copia rápida específicos de la plataforma se pueden usar internamente para copiar el archivo de manera más eficiente. Véase la sección *Operaciones de copia eficientes dependientes de la plataforma*.

`shutil.ignore_patterns(*patterns)`

Esta función de fábrica crea una función que puede ser usada como un invocable para el argumento `ignore` de `copytree()`, ignorando los archivos y directorios que coinciden con uno de los patrones `patterns` de estilo glob provistos. Véase el ejemplo siguiente.

`shutil.copytree(src, dst, symlinks=False, ignore=None, copy_function=copy2, ignore_dangling_symlinks=False, dirs_exist_ok=False)`

Copia recursivamente un directorio `tree` entero con raíz en `src` en un directorio llamado `dst` y retorna un directorio destino. `dirs_exist_ok` dicta si debe generar una excepción en caso de que `dst` o cualquier directorio principal que falte ya exista.

Los permisos y los tiempos de directorios son copiados con `copystat()`; los archivos individuales son copiados usando `copy2()`.

Si `symlinks` es verdadero, los enlaces simbólicos en el árbol de origen son representados como enlaces simbólicos en el árbol nuevo y los metadatos de los enlaces originales serán copiados mientras la plataforma lo permita; si es falso o se omite, los contenidos y los metadatos de los archivos vinculados se copiarán en el árbol nuevo.

Cuando `symlinks` es falso y el archivo al que apunta no exista, se agrega una excepción a la lista de errores generados en una excepción `Error` al final del proceso de copia. Puedes establecer la marca opcional `ignore_dangling_symlinks` en verdadero si deseas silenciar esa excepción. Ten en cuenta que esta opción no tiene efecto en plataformas que no admiten `os.symlink()`.

Si se proporciona `ignore`, debe ser un invocable que recibirá como argumentos el directorio visitado por `copytree()` y una lista de sus contenidos, tal como los retorna `os.listdir()`. Ya que `copytree()` se invoca recursivamente, el invocable `ignore` se llamará una vez por cada directorio que se copia. El invocable debe retornar una secuencia de directorio y de nombres de archivo en relación con el directorio actual (es decir, un subconjunto de los elementos en su segundo argumento); estos nombres serán ignorados en el proceso de copia. `ignore_patterns()` se pueden usar para crear un invocable que ignora los nombres basados en patrones de estilo glob.

Si ocurren excepciones, se genera, un `Error` con una lista de razones.

Si `copy_function` está dado, debe ser un invocable que se usará para copiar cada archivo. Se le invocará con la ruta de origen y la ruta de destino como argumentos. Por defecto, se usa `copy2()`, pero se puede usar cualquier función que admita la misma (como `copy()`).

Se genera un *evento de auditoría* `shutil.copytree` con argumentos `src`, `dst`.

Distinto en la versión 3.3: Copia los metadatos cuando `symlinks` es falso. Ahora retorna `dst`.

Distinto en la versión 3.2: Se añadió el argumento `copy_function` para poder proporcionar una función de copia personalizada. Se añadió el argumento `ignore_dangling_symlinks` a los errores de enlaces colgantes silenciosos cuando `symlinks` es falso.

Distinto en la versión 3.8: Los llamadas al sistema de copia rápida específicos de la plataforma se pueden usar internamente para copiar el archivo de manera más eficiente. Véase la sección *Operaciones de copia eficientes dependientes de la plataforma*.

Nuevo en la versión 3.8: El parámetro `dirs_exist_ok`.

`shutil.rmtree(path, ignore_errors=False, onerror=None)`

Elimina un árbol de directorios entero; `path` debe apuntar a un directorio (pero no a un enlace simbólico a un directorio). Si `ignore_errors` es verdadero, se eliminarán los errores que resulten de eliminaciones fallidas; si

es falso u omitido, estos errores se controlan invocando un controlador especificado por *onerror* o, si este es omitido, se genera una excepción.

Nota: En plataformas que admiten las funciones basadas en descriptores de archivo necesarias, una versión resistente de `rmtree()` al ataque de enlace simbólico se usa por defecto. En otras plataformas, la implementación `rmtree()` es susceptible a un ataque de enlace simbólico; dados el tiempo y las circunstancias adecuados, los atacantes pueden manipular los enlaces simbólicos en el sistema de archivos para eliminar archivos a los que no podrían acceder de otra manera. Las aplicaciones pueden usar el atributo de función `rmtree.avoids_symlink_attacks` para determinar qué caso aplica.

Si se *onerror* está dado, debe ser un invocable que acepte tres parámetros: *function*, *path*, y *excinfo*.

El primer parámetro, *function*, es la función que generó la excepción; depende de la plataforma y de la implementación. El segundo parámetro, *path*, será el nombre de ruta pasado a *function*. El tercer parámetro, *excinfo*, será la información de la excepción retornada por `sys.exc_info()`. Las excepciones generadas por *onerror* no serán tomadas.

Genera un *evento de auditoría* `shutil.rmtree` con argumento *path*.

Distinto en la versión 3.3: Se añadió una versión resistente a los ataques de enlaces simbólicos que se usan automáticamente si la plataforma admite funciones pasadas en descriptores de archivo.

Distinto en la versión 3.8: En Windows, ya no eliminará los contenidos de un cruce de directorios antes de quitar el cruce.

`rmtree.avoids_symlink_attacks`

Indica si la plataforma actual y la implementación proporciona una versión de `rmtree()` resistente al ataque de enlace simbólico. Actualmente, esto solo sucede para plataformas que admitan funciones de acceso a directorios basadas en descriptores de archivo.

Nuevo en la versión 3.3.

`shutil.move(src, dst, copy_function=copy2)`

Mueve de forma recursiva un archivo o directorio (*src*) a otra ubicación (*dst*) y retorna el destino.

Si el destino es un directorio existente, entonces *src* se mueve dentro de ese directorio. Si el destino existe pero no es un directorio, puede sobrescribirse dependiendo de la semántica `os.rename()`.

Si el destino está en el sistema de archivos actual, entonces se usa `os.rename()`. De lo contrario, *src* se copia en *dst* usando *copy_function* y luego se elimina. En el caso de los enlaces simbólicos, se creará un enlace simbólico nuevo que apunta al destino de **src* en o como *dst* y será eliminado.

If *copy_function* is given, it must be a callable that takes two arguments *src* and *dst*, and will be used to copy *src* to *dst* if `os.rename()` cannot be used. If the source is a directory, `copytree()` is called, passing it the `copy_function()`. The default *copy_function* is `copy2()`. Using `copy()` as the *copy_function* allows the move to succeed when it is not possible to also copy the metadata, at the expense of not copying any of the metadata.

Genera un *evento de auditoría* `shutil.move` con argumentos *src*, *dst*.

Distinto en la versión 3.3: Se añadió el manejo explícito de enlaces simbólicos para sistemas de archivos extranjeros, de manera que se adapta al comportamiento del **mv** del GNU. Ahora retorna *dst*.

Distinto en la versión 3.5: Se agregó el argumento de keyword *copy_function*.

Distinto en la versión 3.8: Los llamadas al sistema de copia rápida específicos de la plataforma se pueden usar internamente para copiar el archivo de manera más eficiente. Véase la sección *Operaciones de copia eficientes dependientes de la plataforma*.

`shutil.disk_usage(path)`

Retorna las estadísticas de uso de disco sobre la ruta de acceso dada como un *named tuple* con los atributos *total*, *used* y *free*, que son las cantidades del espacio total, el usado y el libre, en bytes. *path* puede ser un archivo o un directorio.

Nuevo en la versión 3.3.

Distinto en la versión 3.8: En Windows, *path* ahora puede ser un archivo o un directorio.

Disponibilidad: Unix, Windows.

`shutil.chown(path, user=None, group=None)`

Cambia el propietario *user* y/o *group* de la *ruta* dada.

user puede ser un nombre usuario de sistema o un UID (identificador de usuario); lo mismo aplica a *group*. Se requiere al menos un argumento.

Véase también `os.chown()`, la función subyacente.

Genera un *evento de auditoría* `shutil.chown` con argumentos *path*, *user*, *group*.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

`shutil.which(cmd, mode=os.F_OK | os.X_OK, path=None)`

Retorna la ruta de acceso a un ejecutable que se ejecutaría si el *cmd* dado se invoca. Si no se invoca a *cmd*, retorna `None`.

mode es una máscara de permiso que se pasa a `os.access()`, que determinar por defecto si el archivo existe y es ejecutable.

Cuando no se especifica *path*, se usan los resultados de `os.environ()` y se retorna el valor de «PATH» o una reserva de `os.defpath`.

En Windows, el directorio actual siempre se antepone a *path*, independientemente de si usa el valor predeterminado o si tú provees el tuyo propio, que es el comportamiento que el comando *shell* usa cuando encuentra ejecutables. Además, al encontrar el *cmd* en el *path*, se comprueba la variable de entorno `PATHEXT`. Por ejemplo, si invocas `shutil.which("python")`, `which()` buscará `PATHEXT` para saber si debe buscar `python.exe` dentro de los directorios *path*. Por ejemplo, en Windows:

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

Nuevo en la versión 3.3.

Distinto en la versión 3.8: Ahora se acepta el tipo *bytes*. Si el tipo *cmd* es *bytes*, el tipo resultante también es *bytes*.

exception `shutil.Error`

Esta excepción recopila las excepciones que se generan durante una operación de varios archivos. Para `copytree()`, el argumento de excepción es una lista de 3 tuplas (*srcname*, *dstname*, *exception*).

Operaciones de copia eficientes dependientes de la plataforma

Starting from Python 3.8, all functions involving a file copy (`copyfile()`, `copy()`, `copy2()`, `copytree()`, and `move()`) may use platform-specific «fast-copy» syscalls in order to copy the file more efficiently (see [bpo-33671](#)). «fast-copy» means that the copying operation occurs within the kernel, avoiding the use of userspace buffers in Python as in «`outfd.write(infd.read())`».

En macOS, se usar `fcopyfile` para copiar el contenido del archivo (pero no los metadatos).

En Linux, se usa `os.sendfile()`.

En Windows, `shutil.copyfile()` usa un tamaño de búfer predeterminado más grande (1 MiB en lugar de 64 KiB) y se usa una variante de `shutil.copyfileobj()` basada en `memoryview()`.

Si ocurre un error en la operación de copia rápida y no se ha escrito ningún dato en el archivo de destino, entonces `shutil` recurrirá silenciosamente a usar la función `copyfileobj()` menos eficiente internamente.

Distinto en la versión 3.8.

ejemplo de *copytree*

Este ejemplo es la implementación de la función *copytree()*, descrita arriba, con la cadena de documentación omitida. Demuestra muchas de las otras funciones provistas por este módulo.

```
def copytree(src, dst, symlinks=False):
    names = os.listdir(src)
    os.makedirs(dst)
    errors = []
    for name in names:
        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)
            else:
                copy2(srcname, dstname)
                # XXX What about devices, sockets etc.?
        except OSError 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 OSError as why:
        # can't copy file access times on Windows
        if why.winerror is None:
            errors.extend((src, dst, str(why)))
    if errors:
        raise Error(errors)
```

Otro ejemplo que usa el auxiliar *ignore_patterns()*:

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

Esto copiará todo excepto archivos *.pyc* y archivos o directorios cuyo nombre comience con *tmp*.

Otro ejemplo que usa el argumento *ignore* para agregar una llamada a iniciar sesión:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return [] # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```


ejemplo de rmtree

Este ejemplo muestra como eliminar un árbol de directorio en Windows donde algunos de los archivos tienen configurado su *bit* de sólo lectura. Usa la *onerror callback* para limpiar el *bit* de sólo lectura y reintentar eliminarlo. Cualquier fallo que le siga, se propagará.

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onerror=remove_readonly)
```

11.10.2 Operaciones de archivado

Nuevo en la versión 3.2.

Distinto en la versión 3.5: Se agregó soporte para el formato *xztar*.

Se proveen también las utilidades de alto nivel para crear y leer archivos comprimidos y archivados. Utilizan para esto los módulos *zipfile* y *tarfile*.

```
shutil.make_archive(base_name, format[, root_dir[, base_dir[, verbose[, dry_run[, owner[, group[,
logger]]]]]])
```

Crea un documento de archivado (como zip o tar) y retorna su nombre.

base_name es el nombre del archivo a crear, incluyendo la ruta, menos cualquier extensión que sea específica de formato. *format* es el formato de archivo: uno de *zip* (si el módulo *zlib* está disponible), «tar», «gztar» (si el módulo *zlib* está disponible), «bztar» (si el módulo *bz2* está disponible), o «xztar» (si el módulo *lzma*).

root_dir is a directory that will be the root directory of the archive, all paths in the archive will be relative to it; for example, we typically `chdir` into *root_dir* before creating the archive.

base_dir is the directory where we start archiving from; i.e. *base_dir* will be the common prefix of all files and directories in the archive. *base_dir* must be given relative to *root_dir*. See *Archiving example with base_dir* for how to use *base_dir* and *root_dir* together.

root_dir y *base_dir* irán por defecto al directorio actual.

If *dry_run* es verdadero, no se crea un archivo, pero las operaciones que se ejecutarán están registradas en *logger*.

owner y *group* se usan al crear un archivador tar. Por defecto, usa el propietario y el grupo actual.

logger debe ser un objeto compatible con **PEP 282**, usualmente una instancia de *logging.Logger*.

El argumento *verbose* está fuera de uso y es obsoleto.

Genera un *evento de auditoría* `shutil.make_archive` con argumentos *base_name*, *format*, *root_dir*, *base_dir*.

Nota: This function is not thread-safe.

Distinto en la versión 3.8: El formato de *pax* moderno (POSIX.1-2001) se usa ahora en lugar del formato de legado GNU para archivadores creados con `format="tar"`.

```
shutil.get_archive_formats()
```

Retorna una lista de formatos admitidos para archivado. Cada elemento de una secuencia retornada es una tupla (name, description).

Por defecto, *shutil* provee los siguientes formatos:

- *zip*: archivo ZIP (si el módulo *zlib* está disponible).
- *tar*: archivo tar sin comprimir. Utiliza POSIX.1-2001 formato pax para archivos nuevos.
- *gztar*: archivo tar comprimido con *gzip* (si el módulo *zlib* está disponible).
- *bztar*: archivo tar comprimido con *bzip2* (si el módulo *bz2* está disponible).
- *xztar*: archivo tar comprimido con *xz* (si el módulo *lzma* está disponible).

Puedes registrar formatos nuevos o proporcionar tu propio archivador para cualquier formato existente, usando `register_archive_format()`.

`shutil.register_archive_format(name, function[, extra_args[, description]])`

Registra un archivador para el formato *name*.

function es el invocable que se usará para desempacar archivos. El invocable recibirá el *base_name* del archivo para crearlo, seguido por *base_dir* (que cae por defecto en `os.curdir`) para comenzar a archivar desde allí. Otros argumentos se pasan como argumentos de palabras clave: *owner*, *group*, *dry_run* y *logger* (como se pasa en `make_archive()`).

Si está dado, *extra_args* es una secuencia de pares (*name*, *value*) que se usarán como argumentos de palabras clave extra cuando se usa el archivador invocable.

description se usa por `get_archive_formats()` que retorna la lista de archivadores. Cae por defecto en una cadena de caracteres vacía.

`shutil.unregister_archive_format(name)`

Elimina el formato de archivo *name* de la lista de formatos admitidos.

`shutil.unpack_archive(filename[, extract_dir[, format[, filter]]])`

Desempaqueta un archivo. *filename* es la ruta completa del archivo.

extract_dir es el nombre del directorio donde se desempaca el archivo. Si no está provisto, se usa el nombre del directorio actual.

format es el formato de archivo: uno de «zip», «tar», «gztar», «bztar», or «xztar». O cualquier otra formato registrado con `register_unpack_format()`. Si no está provisto, `unpack_archive()` usará la extensión de archivo del archivador y verá si se registró un desempacador para esa extensión. En caso de que no se encuentre ninguno, se genera un *ValueError*.

The keyword-only *filter* argument, which was added in Python 3.8.17, is passed to the underlying unpacking function. For zip files, *filter* is not accepted. For tar files, it is recommended to set it to 'data', unless using features specific to tar and UNIX-like filesystems. (See *Extraction filters* for details.) The 'data' filter will become the default for tar files in Python 3.14.

Se genera un *evento de auditoría* `shutil.unpack_archive` con argumentos *filename*, *extract_dir*, *format*.

Advertencia: Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of the path specified in the *extract_dir* argument, e.g. members that have absolute filenames starting with «/» or filenames with two dots «..».

Distinto en la versión 3.7: Acepta un *path-like object* como *filename* y *extract_dir*.

Distinto en la versión 3.8.17: Added the *filter* argument.

`shutil.register_unpack_format(name, extensions, function[, extra_args[, description]])`

Registra un formato de desempaquete. *name* es el nombre del formato y *extensions* es una lista de extensiones que corresponden al formato, como `.zip` para archivos zip.

function is the callable that will be used to unpack archives. The callable will receive:

- the path of the archive, as a positional argument;
- the directory the archive must be extracted to, as a positional argument;

- possibly a *filter* keyword argument, if it was given to `unpack_archive()`;
- additional keyword arguments, specified by *extra_args* as a sequence of (name, value) tuples.

description se puede proporcionar para describir el formato y será retornado por la función `get_unpack_formats()`.

`shutil.unregister_unpack_format(name)`

Anula el registro del formato de desempaque. *name* es el nombre del formato.

`shutil.get_unpack_formats()`

Retorna una lista de todos los formatos registrados para desempaquetar. Cada elemento de la secuencia es una tupla (name, extensions, description).

Por defecto, *shutil* provee los siguientes formatos:

- *zip*: archivo zip (desempaquetar archivos comprimidos funciona solo si el módulo correspondiente está disponible).
- *tar*: archivo tar sin comprimir.
- *gztar*: archivo tar comprimido con gzip (si el módulo *zlib* está disponible).
- *bztar*: archivo tar comprimido con bzip2 (si el módulo *bz2* está disponible).
- *xztar*: archivo tar comprimido con xz (si el módulo *lzma* está disponible).

Puedes registrar formatos nuevos o proporcionar tu propio desempaketado para cualquier formato existente, usando `register_unpack_format()`.

Ejemplo de archivado

En este ejemplo, creamos un archivo tar de tipo gzip que contiene todos los archivos que se encuentran en el directorio `.ssh` del usuario:

```
>>> 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'
```

El archivo resultante contiene:

```
$ 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
```

Archiving example with *base_dir*

In this example, similar to the *one above*, we show how to use `make_archive()`, but this time with the usage of *base_dir*. We now have the following directory structure:

```
$ tree tmp
tmp
├── root
│   └── structure
│       ├── content
│       │   └── please_add.txt
│       └── do_not_add.txt
```

In the final archive, `please_add.txt` should be included, but `do_not_add.txt` should not. Therefore we use the following:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> make_archive(
...     archive_name,
...     'tar',
...     root_dir='tmp/root',
...     base_dir='structure/content',
... )
'/Users/tarek/my_archive.tar'
```

Listing the files in the resulting archive gives us:

```
$ python -m tarfile -l /Users/tarek/myarchive.tar
structure/content/
structure/content/please_add.txt
```

11.10.3 Consulta el tamaño de la terminal de salida

`shutil.get_terminal_size (fallback=(columns, lines))`

Obtén el tamaño de la ventana de la terminal.

Para cada una de las dos dimensiones, se comprueba la variable de entorno `COLUMNS` y `LINES` respectivamente. Si la variable está definida y el valor es un entero positivo, se utiliza.

Cuando `COLUMNS` o `LINES` no está definido, como suele ocurrir, la terminal conectada a `sys.__stdout__` se consulta invocando `os.get_terminal_size()`.

Si el tamaño de la terminal no se puede consultar correctamente, ya sea porque el sistema no admite consultas o porque no estamos conectados a una terminal, se usa el valor dado en el parámetro `fallback`. `fallback` tiene por defecto `(80, 24)`, que es el tamaño por defecto que se usa en muchos emuladores de terminal.

El valor retornado es una tupla con su respectivo nombre, de tipo `os.terminal_size`.

See also: The Single UNIX Specification, Version 2, [Other Environment Variables](#).

Nuevo en la versión 3.3.

Ver también:

Módulo `os` Interfaces del sistema operativo, incluidas las funciones para trabajar con archivos en un nivel inferior al de Python *file objects*.

Módulo `io` La biblioteca de I/O integrada de Python, incluidas las clases abstractas y algunas clases concretas, como la I/O de archivos.

Función incorporada `open()` La forma estándar de abrir archivos para leer y escribir con Python.

Los módulos descritos en este capítulo soportan el almacenamiento de datos de Python de forma persistente en el disco. Los módulos *pickle* y *marshal* pueden convertir muchos tipos de datos de Python en un flujo de bytes y luego recrear los objetos a partir de los bytes. Los diversos módulos relacionados con DBM admiten una familia de formatos de archivo basados en hash que almacenan un mapeo de cadenas a otras cadenas.

La lista de módulos descritos en este capítulo es:

12.1 *pickle* — Serialización de objetos Python

Código fuente: [Lib/pickle.py](#)

El módulo *pickle* implementa protocolos binarios para serializar y deserializar una estructura de objetos Python. «*Pickling*» es el proceso mediante el cual una jerarquía de objetos de Python se convierte en una secuencia de bytes, y el «*unpickling*» es la operación inversa, mediante la cual una secuencia de bytes de un archivo binario (*binary file*) ó un objeto tipo binario (*bytes-like object*) es convertido nuevamente en una jerarquía de objetos. *Pickling* (y *unpickling*) son alternativamente conocidos como «serialización», «ensamblaje»,¹ o «aplanamiento»; sin embargo, para evitar confusiones, los términos utilizados aquí son «pickling» y «unpickling».

Advertencia: El módulo *pickle* **no es seguro**. Solo deserialice con *pickle* los datos en los que confía.

Es posible construir datos maliciosos con *pickle* que **ejecuten código arbitrario durante el proceso de ‘unpickling’**. Nunca deserialice datos con *pickle* que podrían haber venido de una fuente no confiable, o que podrían haber sido manipulados.

Considere firmar los datos con *hmac* si necesita asegurarse de que no hayan sido alterados.

Los formatos de serialización más seguros como *json* pueden ser más apropiados si está procesando datos no confiables. Ver *Comparación con json*.

¹ No confunda esto con el módulo *marshal*

12.1.1 Relación con otros módulos de Python

Comparación con `marshal`

Python tiene un módulo de serialización más primitivo llamado `marshal`, pero en general `pickle` debería ser siempre la forma preferida de serializar objetos de Python. `marshal` existe principalmente para soportar archivos Python `.pyc`.

El módulo `pickle` difiere de `marshal` en varias formas significativas:

- El módulo `pickle` realiza un seguimiento de los objetos que ya ha serializado, para que las referencias posteriores al mismo objeto no se serialicen nuevamente. `marshal` no hace esto.

Esto tiene implicaciones tanto para los objetos recursivos como para compartir objetos. Los objetos recursivos son objetos que contienen referencias a sí mismos. `Marshal` no los maneja y, de hecho, intentar agrupar objetos recursivos bloqueará su intérprete de Python. El intercambio de objetos ocurre cuando hay múltiples referencias al mismo objeto en diferentes lugares de la jerarquía de objetos que se serializan. `pickle` almacena dichos objetos solo una vez y garantiza que todas las demás referencias apunten a la copia maestra. Los objetos compartidos permanecen compartidos, lo cual puede ser muy importante para los objetos mutables.

- `marshal` no se puede usar para serializar clases definidas por el usuario y sus instancias. `pickle` puede guardar y restaurar instancias de clase de forma transparente, sin embargo, la definición de clase debe ser importable y vivir en el mismo módulo que cuando se almacenó el objeto.
- No se garantiza que el formato de serialización `marshal` sea portable a través de todas las versiones de Python. Debido a que su trabajo principal es dar soporte a archivos `.pyc`, los implementadores de Python se reservan el derecho de cambiar el formato de serialización de formas no compatibles con versiones anteriores si surge la necesidad. El formato de serialización `pickle` está garantizado para ser compatible con versiones anteriores de Python siempre que se elija un protocolo de `pickle` compatible y el serializado y deserializado de código con `pickle` se encargue de lidiar con las diferencias de tipos entre Python 2 y Python 3 si sus datos están cruzando ese límite único entre las versiones del lenguaje.

Comparación con `json`

Existen diferencias fundamentales entre los protocolos de `pickle` y JSON (acrónimo de JavaScript Object Notation, «notación de objeto de JavaScript»):

- JSON es un formato de serialización de texto (genera texto unicode, aunque la mayoría de las veces se codifica a utf-8), mientras que `pickle` es un formato de serialización binario;
- JSON es legible por humanos, mientras que `pickle` no lo es;
- JSON es interoperable y ampliamente utilizado fuera del ecosistema de Python, mientras que `pickle` es específico de Python;
- JSON, por defecto, solo puede representar un subconjunto de los tipos integrados de Python, y no clases personalizadas; `pickle` puede representar un número extremadamente grande de tipos de Python (muchos de ellos automáticamente, mediante el uso inteligente de la introspección de objetos en Python; los casos complejos se pueden abordar implementando API de objetos específicos, *specific object APIs*);
- A diferencia de `pickle`, deserializar JSON no confiable no crea en sí mismo una vulnerabilidad de ejecución de código arbitraria.

Ver también:

El módulo `json`: un módulo de la biblioteca estándar que permite la serialización y deserialización de JSON.

12.1.2 Formato de flujo de datos

El formato de datos utilizado por *pickle* es específico de Python. Esto tiene la ventaja de que no hay restricciones impuestas por estándares externos como JSON o XDR (que no pueden representar el uso compartido de punteros); sin embargo, significa que los programas que no son de Python pueden no ser capaces de reconstruir objetos Python serializados con *pickle*.

Por defecto, el formato de datos *pickle* utiliza una representación binaria relativamente compacta. Si necesita características de tamaño óptimas, puede eficientemente *comprimir* datos serializados con *pickle*.

El módulo *pickletools* contiene herramientas para analizar flujos de datos generados por *pickle*. El código fuente de *pickletools* tiene comentarios extensos sobre los códigos de operación utilizados por los protocolos de *pickle*.

Actualmente hay 6 protocolos diferentes que se pueden utilizar para serializar con *pickle*. Cuanto mayor sea el protocolo utilizado, más reciente será la versión de Python necesaria para leer el *pickle* producido.

- La versión 0 del protocolo es el protocolo original «legible para humanos» y es compatible con versiones anteriores de Python.
- La versión 1 del protocolo es un formato binario antiguo que también es compatible con versiones anteriores de Python.
- La versión 2 del protocolo se introdujo en Python 2.3. Proporciona un serializado con *pickle* mucho más eficiente de clases de estilo nuevo (*new-style class*). Consulte [PEP 307](#) para obtener información sobre las mejoras aportadas por el protocolo 2.
- Se agregó la versión 3 del protocolo en Python 3.0. Tiene soporte explícito para objetos *bytes* y no puede ser deserializado con *pickle* por Python 2.x. Este era el protocolo predeterminado en Python 3.0–3.7.
- Se agregó la versión 4 del protocolo en Python 3.4. Agrega soporte para objetos muy grandes, *pickling* de mas tipos de objetos y algunas optimizaciones de formato de datos. Es el protocolo predeterminado que comienza con Python 3.8. Consulte [PEP 3154](#) para obtener información sobre las mejoras aportadas por el protocolo 4.
- Se agregó la versión 5 del protocolo en Python 3.8. Agrega soporte para datos fuera de banda y aceleración para datos dentro de banda. Consulte [PEP 574](#) para obtener información sobre las mejoras aportadas por el protocolo 5.

Nota: La serialización es una noción más primitiva que la persistencia; aunque *pickle* lee y escribe objetos de archivo, no maneja el problema de nombrar objetos persistentes, ni el problema (aún más complicado) de acceso concurrente a objetos persistentes. El módulo *pickle* puede transformar un objeto complejo en una secuencia de bytes y puede transformar la secuencia de bytes en un objeto con la misma estructura interna. Quizás lo más obvio que hacer con estos flujos de bytes es escribirlos en un archivo, pero también es concebible enviarlos a través de una red o almacenarlos en una base de datos. El módulo *shelve* proporciona una interfaz simple para serializar y deserializar objetos con *pickle* en archivos de bases de datos de estilo DBM.

12.1.3 Interfaz del módulo

Para serializar una jerarquía de objetos, simplemente llame a la función *dumps()*. De manera similar, para deserializar un flujo de datos, llama a la función *loads()*. Sin embargo, si desea tener más control sobre la serialización y la deserialización, puede crear un objeto *Pickler* o *Unpickler*, respectivamente.

El módulo *pickle* proporciona las siguientes constantes:

`pickle.HIGHEST_PROTOCOL`

Un entero, la versión de protocolo (*protocol version*) más alta disponible. Este valor se puede pasar como un valor de *protocolo* a las funciones *dump()* y *dumps()* así como al constructor *Pickler*.

`pickle.DEFAULT_PROTOCOL`

Un entero, la versión de protocolo (*protocol version*) predeterminada utilizada para el serializado con *pickle*. Puede ser menor que *HIGHEST_PROTOCOL*. Actualmente, el protocolo predeterminado es 4, introducido por primera vez en Python 3.4 e incompatible con versiones anteriores.

Distinto en la versión 3.0: El protocolo predeterminado es 3.

Distinto en la versión 3.8: El protocolo predeterminado es 4.

El módulo `pickle` proporciona las siguientes funciones para que el proceso de *pickling* sea más conveniente:

`pickle.dump(obj, file, protocol=None, *, fix_imports=True, buffer_callback=None)`

Escribe la representación *pickle* del objeto *obj* en el archivo abierto *file object*. Esto es equivalente a `Pickler(file, protocol).dump(obj)`.

Los argumentos *file*, *protocol*, *fix_imports* y *buffer_callback* tienen el mismo significado que en el constructor `Pickler`.

Distinto en la versión 3.8: Se agregó el argumento *buffer_callback*.

`pickle.dumps(obj, protocol=None, *, fix_imports=True, buffer_callback=None)`

Retorna la representación *pickle* del objeto *obj* como un objeto *bytes*, en lugar de escribirlo en un archivo.

Los argumentos *protocol*, *fix_imports* y *buffer_callback* tienen el mismo significado que en el constructor `Pickler`.

Distinto en la versión 3.8: Se agregó el argumento *buffer_callback*.

`pickle.load(file, *, fix_imports=True, encoding="ASCII", errors="strict", buffers=None)`

Lee la representación *pickle* de un objeto desde un archivo abierto *file object* y retorna la jerarquía de objetos reconstituidos especificada en el mismo. Esto es equivalente a `Unpickler(file).load()`.

La versión de protocolo del *pickle* se detecta automáticamente, por lo que no se necesita ningún argumento de protocolo. Los bytes más allá de la representación empaquetada son ignorados.

Los argumentos *file*, *fix_imports*, *encoding*, *errors*, *strict* y *buffers* tienen el mismo significado que en el constructor `Unpickler`.

Distinto en la versión 3.8: Se agregó el argumento *buffers*.

`pickle.loads(data, *, fix_imports=True, encoding="ASCII", errors="strict", buffers=None)`

Retorna la jerarquía de objetos reconstruida de la representación *pickle data* de un objeto. *data* debe ser un objeto tipo binario (*bytes-like object*).

La versión de protocolo del *pickle* se detecta automáticamente, por lo que no se necesita ningún argumento de protocolo. Los bytes más allá de la representación empaquetada son ignorados.

Los argumentos *file*, *fix_imports*, *encoding*, *errors*, *strict* y *buffers* tienen el mismo significado que en el constructor `Unpickler`.

Distinto en la versión 3.8: Se agregó el argumento *buffers*.

El módulo `pickle` define tres excepciones:

exception `pickle.PickleError`

Clase base común para las otras excepciones de *pickling*. Hereda de `Exception`.

exception `pickle.PicklingError`

Error generado cuando `Pickler` encuentra un objeto que no se puede serializar con *pickle*. Hereda de `PickleError`.

Consulte *¿Qué se puede serializar (pickled) y deserializar (unpickled) con pickle?* para aprender qué tipos de objetos se pueden serializar con *pickle*.

exception `pickle.UnpicklingError`

Se produce un error cuando hay un problema al deserializar un objeto con *pickle*, por ejemplo como una corrupción de datos o una violación de seguridad. Hereda de `PickleError`.

Tenga en cuenta que también se pueden generar otras excepciones durante la deserialización con *pickle*, incluyendo (pero no necesariamente limitado a) `AttributeError`, `EOFError`, `ImportError`, e `IndexError`.

El módulo `pickle` exporta tres clases, `Pickler`, `Unpickler` y `PickleBuffer`:

class `pickle.Pickler` (*file*, *protocol=None*, *, *fix_imports=True*, *buffer_callback=None*)

Esto toma un archivo binario para escribir un flujo de datos de *pickle*.

El argumento opcional *protocol*, un entero, le dice al *pickler* que use el protocolo dado; los protocolos admitidos son 0 para *HIGHEST_PROTOCOL*. Si no se especifica, el valor predeterminado es *DEFAULT_PROTOCOL*. Si se especifica un número negativo, *HIGHEST_PROTOCOL* es seleccionado.

El argumento *file* debe tener un método *write()* que acepte un argumento de bytes individuales. Por lo tanto, puede ser un archivo en disco abierto para escritura binaria, una instancia *io.BytesIO*, o cualquier otro objeto personalizado que cumpla con esta interfaz.

Si *fix_imports* es verdadero y *protocol* es menor que 3, *pickle* intentará asignar los nuevos nombres de Python 3 a los nombres de módulos antiguos utilizados en Python 2, de modo que la secuencia de datos de *pickle* sea legible con Python 2.

Si *buffer_callback* es *None* (el valor predeterminado), las vistas de búfer se serializan en *file* como parte de la secuencia de *pickle*.

Si *buffer_callback* no es *None*, entonces se puede llamar cualquier número de veces con una vista de búfer. Si la *callback* retorna un valor falso (como *None*), el búfer dado está fuera de banda (*out-of-band*); de lo contrario, el búfer se serializa en banda, es decir, dentro del flujo de *pickle*.

Es un error si *buffer_callback* no es *None* y *protocol* es *None* o menor que 5.

Distinto en la versión 3.8: Se agregó el argumento *buffer_callback*.

dump (*obj*)

Escribe la representación serializada con *pickle* del objeto *obj* en el objeto archivo abierto dado en el constructor.

persistent_id (*obj*)

No hacer nada por defecto. Esto existe para que una subclase pueda sobrescribirlo.

Si *persistent_id()* retorna *None*, *obj* es serializado con *pickle* como siempre. Cualquier otro valor hace que *Pickler* emita el valor retornado como un ID persistente para *obj*. El significado de este ID persistente debe definirse por *Unpickler.persistent_load()*. Tenga en cuenta que el valor retornado por *persistent_id()* no puede tener una ID persistente.

Ver *Persistencia de objetos externos* para detalles y ejemplos de uso.

dispatch_table

La tabla de envío de un objeto *Pickler* es un registro de *funciones de reducción* del tipo que se puede declarar usando *copyreg.pickle()*. Es un mapeo cuyas claves son clases y cuyos valores son funciones de reducción. Una función de reducción toma un solo argumento de la clase asociada y debe ajustarse a la misma interfaz que un método *__reduce__()*.

Por defecto, un objeto de *pickle* no tendrá un atributo *dispatch_table*, y en su lugar utilizará la tabla de despacho global administrada por el módulo *copyreg*. Sin embargo, para personalizar el *pickling* para un objeto de *pickle* específico, se puede establecer el atributo *dispatch_table* en un objeto tipo dict. Alternativamente, si una subclase de *Pickler* tiene un atributo *dispatch_table* esto se usará como la tabla de despacho predeterminada para instancias de esa clase.

Ver *Tablas de despacho* para ejemplos de uso.

Nuevo en la versión 3.3.

reducer_override (*self*, *obj*)

Reductor especial que se puede definir en subclases de *Pickler*. Este método tiene prioridad sobre cualquier reductor en *dispatch_table*. Debe cumplir con la misma interfaz que un método *__reduce__()*, y opcionalmente puede retornar *NotImplemented* para recurrir a reductores registrados en *dispatch_table* el objeto *pickle obj*.

Para un ejemplo detallado, ver *Reducción personalizada para tipos, funciones y otros objetos*.

Nuevo en la versión 3.8.

fast

Obsoleto. Habilite el modo rápido si se establece en un valor verdadero. El modo rápido deshabilita

el uso de memo, por lo tanto, acelera el proceso de *pickling* al no generar códigos de operación PUT superfluos. No debe usarse con objetos autorreferenciales; de lo contrario, la clase *Pickler* se repetirá infinitamente.

Use *pickletools.optimize()* si necesita *pickles* más compactos.

class *pickle.Unpickler* (*file*, *, *fix_imports*=True, *encoding*="ASCII", *errors*="strict", *buffers*=None)

Esto toma un archivo binario para leer un flujo de datos de *pickle*.

La versión de protocolo de *pickle* se detecta automáticamente, por lo que no se necesita ningún argumento de protocolo.

El argumento *file* debe tener tres métodos, un método *read()* que toma un argumento entero, un método *readinto()* que toma un argumento búfer y un método *readline()* que no requiere argumentos, como en la interfaz *io.BufferedIOBase*. Por lo tanto *file* puede ser un archivo en disco abierto para lectura binaria, un objeto *io.BytesIO*, o cualquier otro objeto personalizado que cumpla con esta interfaz.

Los argumentos opcionales *fix_imports*, *encoding* and *errors* se utilizan para controlar el soporte de compatibilidad para el flujo de *pickle* generado por Python 2. Si *fix_imports* es verdadero, *pickle* intentará asignar los nombres antiguos de Python 2 a los nuevos nombres utilizados en Python 3. Tanto *encoding* como *errors* le indican a *pickle* cómo decodificar instancias de cadenas de 8 bits seleccionadas por Python 2; estos son predeterminados a "ASCII" y "strict", respectivamente. *encoding* puede ser "bytes" para leer estas instancias de cadena de 8 bits como objetos de bytes. Se requiere el uso de *encoding='latin1'* para realizar el *unpickling* de arreglos de NumPy e instancias de *datetime*, *date* y *time* serializados con *pickle* por Python 2.

Si *buffers* es None (el valor predeterminado), todos los datos necesarios para la deserialización deben estar contenidos en el flujo de *pickle*. Esto significa que el argumento *buffer_callback* era None cuando se instanciaba una clase *Pickler* (o cuando se llamaba a *dump()* o *dumps()*).

Si *buffers* no es None, debería ser un iterable de objetos habilitados para almacenamiento intermedio que se consumen cada vez que el flujo de *pickle* hace referencia a una vista de buffer fuera de banda (*out-of-band*). Tales buffers se han dado para el *buffer_callback* de un objeto *Pickler*.

Distinto en la versión 3.8: Se agregó el argumento *buffers*.

load()

Lee la representación serializada con *pickle* de un objeto desde el objeto de archivo abierto dado en el constructor, y retorne la jerarquía de objetos reconstituídos especificada allí. Los Bytes más allá de la representación serializada con *pickle* del objeto se ignoran.

persistent_load(pid)

Lanza un *UnpicklingError* de forma predeterminada.

Si se define, *persistent_load()* debería retornar el objeto especificado por el ID persistente *pid*. Si se encuentra un ID persistente no válido, se debe lanzar un *UnpicklingError*.

Ver *Persistencia de objetos externos* para detalles y ejemplos de uso.

find_class(module, name)

Importa *module* si es necesario y retorna el objeto llamado *name* desde el, donde los argumentos *module* y *name* son objetos de *str*. Tenga en cuenta que, a diferencia de lo que sugiere su nombre, *find_class()* también se usa para buscar funciones.

Las subclases pueden sobrescribir esto para obtener control sobre qué tipo de objetos y cómo se pueden cargar, reduciendo potencialmente los riesgos de seguridad. Consulte *Restricción de Globals* para obtener más detalles.

Lanza un *auditing event* *pickle.find_class* con argumentos *module*, *name*.

class *pickle.PickleBuffer* (*buffer*)

Un envoltorio (*wrapper*) para un búfer que representa datos serializables con *pickle* (*picklable data*). *buffer* debe ser un objeto que proporciona un búfer (buffer-providing), como objeto tipo binario (*bytes-like object*) o un arreglo N-dimensional.

PickleBuffer es en sí mismo un proveedor de búfer, por lo que es posible pasarlo a otras API que esperan un objeto que provea un búfer, como *memoryview*.

Los objetos `PickleBuffer` solo se pueden serializar usando el protocolo `pickle` 5 o superior. Son elegibles para serialización fuera de banda (*out-of-band serialization*).

Nuevo en la versión 3.8.

raw()

Retorna un `memoryview` del área de memoria subyacente a este búfer. El objeto retornado es una vista de memoria unidimensional, C-contigua con formato B (bytes sin firmar). `BufferError` es lanzado si el búfer no es contiguo a C ni a Fortran.

release()

Libera el búfer subyacente expuesto por el objeto `PickleBuffer`.

12.1.4 ¿Qué se puede serializar (pickled) y deserializar (unpickled) con `pickle`?

Los siguientes tipos se pueden serializar con `pickle` (pickled):

- `None`, `True`, y `False`
- enteros, números de coma flotante, números complejos
- cadenas, bytes, bytearray
- tuplas, listas, conjuntos y diccionarios que contiene solo objetos serializables con `pickle`
- funciones definidas en el nivel superior de un módulo (usando `def`, no `lambda`)
- funciones integradas definidas en el nivel superior de un módulo
- clases que se definen en el nivel superior de un módulo
- instancias de tales clases cuyo `__dict__` o el resultado de llamar a `__getstate__()` es serializable con `pickle` (picklable) (consulte la sección *Pickling de Instancias de clases* para obtener más detalles).

Los intentos de serializar objetos no serializables con `pickle` lanzarán la excepción `PicklingError`; cuando esto sucede, es posible que ya se haya escrito una cantidad no especificada de bytes en el archivo subyacente. Intentar serializar con `pickle` una estructura de datos altamente recursiva puede exceder la profundidad máxima de recursividad, en este caso se lanzará un `RecursionError`. Puede aumentar cuidadosamente este límite con `sys.setrecursionlimit()`.

Tenga en cuenta que las funciones (integradas y definidas por el usuario) se serializan con `pickle` por referencia de nombre «totalmente calificado», no por valor.² Esto significa que solo se serializa con `pickle` el nombre de la función, junto con el nombre del módulo en el que está definida la función. Ni el código de la función, ni ninguno de sus atributos de función se serializa. Por lo tanto, el módulo de definición debe ser importable en el entorno donde se hará el *unpickling*, y el módulo debe contener el objeto nombrado; de lo contrario, se lanzará una excepción.³

De manera similar, las clases se serializan con `pickle` por referencia con nombre, por lo que se aplican las mismas restricciones en el entorno donde se hará el *unpickling*. Tenga en cuenta que ninguno de los datos o el código de la clase son serializados con `pickle`, por lo que en el siguiente ejemplo el atributo de clase `attr` no se restaura en el entorno donde se hará el *unpickling*:

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

Estas restricciones son la razón por la que las funciones y clases serializables con `pickle` deben definirse en el nivel superior de un módulo.

De manera similar, cuando las instancias de clases son serializadas con `pickle`, el código y los datos de la clase no son serializados junto con ella. Solo los datos de la instancia son serializados con `pickle` (*pickled*). Esto se hace a propósito, por lo que puede corregir errores en una clase o agregar métodos a la clase y aún cargar objetos que fueron creados

² Esta es la razón por la que las funciones `lambda` no se pueden serializar con `pickle`: todas las funciones `lambda` comparten el mismo nombre: `<lambda>`.

³ La excepción generada probablemente será un `ImportError` o un `AttributeError` pero podría ser otra cosa.

con una versión anterior de la clase. Si planea tener objetos de larga duración que verán muchas versiones de una clase, puede valer la pena poner un número de versión en los objetos para que las conversiones adecuadas se puedan realizar mediante el método `__setstate__()`.

12.1.5 *Pickling* de Instancias de clases

En esta sección, describimos los mecanismos generales disponibles para que usted defina, personalice y controle cómo se serializan y deserializan con *Pickle* las instancias de clase.

En la mayoría de los casos, no se necesita código adicional para hacer que las instancias sean *picklable* (serializables con *pickle*). Por defecto, *pickle* recuperará la clase y los atributos de una instancia a través de la introspección. Cuando una instancia de clase es deserializada con *pickle* (*unpickled*), su método `__init__()` generalmente *no* se invoca. El comportamiento predeterminado es que primero crea una instancia no inicializada y luego restaura los atributos guardados. El siguiente código muestra una implementación de este comportamiento:

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def load(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

Las clases pueden alterar el comportamiento predeterminado proporcionando uno o varios métodos especiales:

`object.__getnewargs_ex__()`

En los protocolos 2 y más recientes, las clases que implementan el método `__getnewargs_ex__()` pueden dictar los valores pasados al método `__new__()` al hacer *unpickling*. El método debe retornar un par `(args, kwargs)` donde `args` es una tupla de argumentos posicionales y `kwargs` un diccionario de argumentos con nombre para construir el objeto. Estos se pasarán al método `__new__()` al hacer *unpickling*.

Debes implementar este método si el método `__new__()` de tu clase requiere argumentos de solo palabras clave. De lo contrario, se recomienda para la compatibilidad implementar `__getnewargs__()`.

Distinto en la versión 3.6: `__getnewargs_ex__()` ahora se usa en los protocolos 2 y 3.

`object.__getnewargs__()`

Este método tiene un propósito similar a `__getnewargs_ex__()`, pero solo admite argumentos posicionales. Debe retornar una tupla de argumentos `args` que se pasarán al método `__new__()` al hacer *unpickling*.

`__getnewargs__()` no se llamará si `__getnewargs_ex__()` está definido.

Distinto en la versión 3.6: Antes de Python 3.6, se llamaba a, `__getnewargs__()` en lugar de `__getnewargs_ex__()` en los protocolos 2 y 3.

`object.__getstate__()`

Las clases pueden influir aún más en cómo sus instancias se serializan con *pickle*; si la clase define el método `__getstate__()`, este es llamado y el objeto retornado se selecciona como contenido de la instancia, en lugar del contenido del diccionario de la instancia. Si el método `__getstate__()` está ausente, el `__dict__` de la instancia se conserva como de costumbre.

`object.__setstate__(state)`

Al hacer *unpickling*, si la clase define `__setstate__()`, este es llamado con el estado *unpickled* (no serializado con *pickle*). En ese caso, no es necesario que el objeto de estado sea un diccionario. De lo contrario, el estado *pickled* (*pickled state*) debe ser un diccionario y sus elementos se asignan al diccionario de la nueva instancia.

Nota: Si `__getstate__()` retorna un valor falso, el método `__setstate__()` no se llamará al hacer *unpickling*.

Consulte la sección *Manejo de objetos con estado* para obtener más información sobre cómo utilizar los métodos `__getstate__()` y `__setstate__()`.

Nota: Al momento de hacer *unpickling*, algunos métodos como `__getattr__()`, `__getattribute__()`, o `__setattr__()` pueden invocarse sobre la instancia. En caso de que esos métodos dependan de que algún invariante interno sea verdadero, el tipo debería implementar `__new__()` para establecer tal invariante, ya que `__init__()` no se llama cuando se hace *unpickling* de una instancia.

Como veremos, *pickle* no utiliza directamente los métodos descritos anteriormente. De hecho, estos métodos son parte del protocolo de copia que implementa el método especial `__reduce__()`. El protocolo de copia proporciona una interfaz unificada para recuperar los datos necesarios para hacer el *pickling* y la copia de objetos.⁴

Aunque es poderoso, implementar `__reduce__()` directamente en sus clases es propenso a errores. Por esta razón, los diseñadores de clases deben usar la interfaz de alto nivel (es decir, `__getnewargs_ex__()`, `__getstate__()` y `__setstate__()`) siempre que sea posible. Sin embargo, mostraremos casos en los que usar `__reduce__()` es la única opción o conduce a un *pickling* más eficiente o ambos.

`object.__reduce__()`

La interfaz se define actualmente de la siguiente manera. El método `__reduce__()` no toma ningún argumento y retornará una cadena o preferiblemente una tupla (el objeto retornado a menudo se denomina «valor reducido»).

Si se retorna una cadena, la cadena debe interpretarse como el nombre de una variable global. Debe ser el nombre local del objeto relativo a su módulo; el módulo *pickle* busca en el espacio de nombres del módulo para determinar el módulo del objeto. Este comportamiento suele ser útil para singletons.

Cuando se retorna una tupla, debe tener entre dos y seis elementos. Los elementos opcionales se pueden omitir o se puede proporcionar `None` como su valor. La semántica de cada elemento está en orden:

- Un objeto invocable que se llamará para crear la versión inicial del objeto.
- Una tupla de argumentos para el objeto invocable. Se debe proporcionar una tupla vacía si el invocable no acepta ningún argumento.
- Opcionalmente, el estado del objeto, que se pasará al método `__setstate__()` del objeto como se describió anteriormente. Si el objeto no tiene dicho método, el valor debe ser un diccionario y se agregará al atributo `__dict__` del objeto.
- Opcionalmente, un iterador (y no una secuencia) produce elementos sucesivos. Estos elementos se agregarán al objeto usando `obj.append(item)` o, por lotes, usando `obj.extend(list_of_items)`. Esto se usa principalmente para subclasses de lista, pero puede ser usado por otras clases siempre que tengan los métodos `append()` y `extend()` con la firma apropiada. (El uso de `append()` o `extend()` depende de la versión del protocolo *pickle* que se use, así como de la cantidad de elementos que se agregarán, por lo que ambos deben ser soportados.)
- Opcionalmente, un iterador (no una secuencia) que produce pares clave-valor sucesivos. Estos elementos se almacenarán en el objeto usando `obj[key] = value`. Esto se usa principalmente para subclasses de diccionario, pero otras clases pueden usarlo siempre que implementen `__setitem__()`.
- Opcionalmente, un invocable con una firma `(obj, state)`. Este invocable permite al usuario controlar programáticamente el comportamiento de actualización de estado de un objeto específico, en lugar de usar el método estático de `obj.__setstate__()`. Si no es `None`, este invocable tendrá prioridad sobre `obj's __setstate__()`.

Nuevo en la versión 3.8: Se agregó el sexto elemento opcional de tupla `(obj, state)`.

`object.__reduce_ex__(protocol)`

Alternativamente, se puede definir un método `__reduce_ex__()`. La única diferencia es que este método debe tomar un único argumento entero, la versión del protocolo. Cuando esté definido, *pickle* lo preferirá en lugar del método `__reduce__()`. Además, `__reduce__()` se convierte automáticamente en sinónimo de la versión extendida. El uso principal de este método es proporcionar valores reducidos compatibles con versiones anteriores para versiones anteriores de Python.

⁴ El módulo *copy* utiliza este protocolo para operaciones de copia superficial y profunda.

Persistencia de objetos externos

Para el beneficio de la persistencia del objeto, el módulo `pickle` admite la noción de una referencia a un objeto fuera del flujo de datos serializados con `pickle`. Dichos objetos son referenciados por un ID persistente, que debe ser una cadena de caracteres alfanuméricos (para el protocolo 0)⁵ o simplemente un objeto arbitrario (para cualquier protocolo más nuevo).

La resolución de tales ID persistentes no está definida por el módulo `pickle`; delegará esta resolución a los métodos definidos por el usuario en el `pickler` y el `unpickler`, `persistent_id()` y `persistent_load()` respectivamente.

Para seleccionar objetos que tienen una ID persistente externo, el `pickler` debe tener un método personalizado `persistent_id()` que toma un objeto como argumento y retorna `None` o el ID persistente para ese objeto. Cuando se retorna `None`, el `pickler` simplemente serializará el objeto de forma normal. Cuando se retorna una cadena de identificación persistente, el `pickler` serializará ese objeto, junto con un marcador para que el `unpickler` lo reconozca como una identificación persistente.

Para hacer el `unpickling` objetos externos, el `unpickler` debe tener un método personalizado `persistent_load()` que toma un objeto de identificación persistente y retorna el objeto referenciado.

Aquí hay un ejemplo completo que presenta cómo se puede usar la identificación persistente para hacer el `pickling` objetos externos por referencia.

```
# Simple example presenting how persistent ID can be used to pickle
# external objects by reference.

import pickle
import sqlite3
from collections import namedtuple

# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class instance, we emit a
        # persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple, containing a tag and a
            # key, which refers to a specific record in the database.
            return ("MemoRecord", obj.key)
        else:
            # If obj does not have a persistent ID, return None. This means obj
            # needs to be pickled as usual.
            return None

class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is encountered.
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
```

(continué en la próxima página)

⁵ La limitación de caracteres alfanuméricos se debe al hecho de que los ID persistentes, en el protocolo 0, están delimitados por el carácter de nueva línea. Por lo tanto, si se produce algún tipo de carácter de nueva línea en los ID persistentes, el serializado con `pickle` resultante se volverá ilegible.

(proviene de la página anterior)

```

    if type_tag == "MemoRecord":
        # Fetch the referenced record from the database and return it.
        cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
        key, task = cursor.fetchone()
        return MemoRecord(key, task)
    else:
        # Always raises an error if you cannot return the correct object.
        # Otherwise, the unpickler will think None is the object referenced
        # by the persistent ID.
        raise pickle.UnpicklingError("unsupported persistent object")

def main():
    import io
    import pprint

    # Initialize and populate our database.
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
        'fight with a zebra',
    )
    for task in tasks:
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))

    # Fetch the records to be pickled.
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # Save the records using our custom DBPickler.
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # Update a record, just for good measure.
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

    # Load the records from the pickle data stream.
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("Unpickled records:")
    pprint.pprint(memos)

if __name__ == '__main__':
    main()

```

Tablas de despacho

Si se desea personalizar el *pickling* de algunas clases sin alterar ningún otro código que dependa del *pickling*, se puede crear un *pickler* con una tabla de despacho privada.

La tabla de despacho global administrada por el módulo `copyreg` está disponible como `copyreg.dispatch_table`. Por lo tanto, se puede optar por utilizar una copia modificada de `copyreg.dispatch_table` como tabla de envío privada.

Por ejemplo

```
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

crea una instancia de `pickle.Pickler` con una tabla de despacho privada que maneja la clase `AlgunaClase` especialmente. Alternativamente, el código

```
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

hace lo mismo, pero todas las instancias de `MiPickler` compartirán por defecto la misma tabla de despacho. El código equivalente que usa el módulo `copyreg` es

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

Manejo de objetos con estado

Aquí hay un ejemplo que muestra cómo modificar el comportamiento del *pickling* de una clase. La clase `TextReader` abre un archivo de texto y retorna el número de línea y el contenido de la línea cada vez que se llama a su método `readline()`. Si se selecciona una instancia de `TextReader` se guardan todos los atributos *excepto* el miembro del objeto de archivo. Cuando se hace el *unpickling* de la instancia, el archivo se vuelve a abrir y la lectura se reanuda desde la última ubicación. Los métodos `__setstate__()` y `__getstate__()` se utilizan para implementar este comportamiento.

```
class TextReader:
    """Print and number lines in a text file."""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # Copy the object's state from self.__dict__ which contains
```

(continué en la próxima página)

(proviene de la página anterior)

```

    # all our instance attributes. Always use the dict.copy()
    # method to avoid modifying the original state.
    state = self.__dict__.copy()
    # Remove the unpicklable entries.
    del state['file']
    return state

    def __setstate__(self, state):
        # Restore instance attributes (i.e., filename and lineno).
        self.__dict__.update(state)
        # Restore the previously opened file's state. To do so, we need to
        # reopen it and read from it until the line count is restored.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # Finally, save the file.
        self.file = file

```

Un ejemplo de uso podría ser algo como esto:

```

>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'

```

12.1.6 Reducción personalizada para tipos, funciones y otros objetos

Nuevo en la versión 3.8.

A veces, `dispatch_table` puede no ser lo suficientemente flexible. En particular, es posible que deseemos personalizar el *pickling* en función de otro criterio que no sea el tipo de objeto, o es posible que deseemos personalizar el *pickling* de funciones y clases.

Para esos casos, es posible crear una subclase de la clase `Pickler` e implementar el método `reducer_override()`. Este método puede retornar una tupla de reducción arbitraria (ver `__reduce__()`). Alternativamente, puede retornar `NotImplemented` para volver al comportamiento tradicional.

Si se definen tanto `dispatch_table` como `reducer_override()`, entonces `reducer_override()` tiene prioridad.

Nota: Por motivos de rendimiento, no se puede llamar a `reducer_override()` para los siguientes objetos: `None`, `True`, `False`, e instancias exactas de `int`, `float`, `bytes`, `str`, `dict`, `set`, `frozenset`, `list` y `tuple`.

Aquí hay un ejemplo simple donde permitimos el *pickling* y reconstruir una clase dada `class`:

```

import io
import pickle

class MyClass:
    my_attribute = 1

class MyPickler(pickle.Pickler):
    def reducer_override(self, obj):

```

(continúe en la próxima página)

(proviene de la página anterior)

```

"""Custom reducer for MyClass."""
if getattr(obj, "__name__", None) == "MyClass":
    return type, (obj.__name__, obj.__bases__,
                  {'my_attribute': obj.my_attribute})
else:
    # For any other object, fallback to usual reduction
    return NotImplemented

f = io.BytesIO()
p = MyPickler(f)
p.dump(MyClass)

del MyClass

unpickled_class = pickle.loads(f.getvalue())

assert isinstance(unpickled_class, type)
assert unpickled_class.__name__ == "MyClass"
assert unpickled_class.my_attribute == 1

```

12.1.7 Búferes fuera de banda

Nuevo en la versión 3.8.

En algunos contextos, el módulo *pickle* se usa para transferir cantidades masivas de datos. Por lo tanto, puede ser importante minimizar el número de copias de memoria para preservar el rendimiento y el consumo de recursos. Sin embargo, el funcionamiento normal del módulo *pickle*, ya que transforma una estructura gráfica de objetos en un flujo secuencial de bytes, implica intrínsecamente copiar datos hacia y desde el flujo *pickle*.

Esta restricción puede evitarse si tanto el *proveedor* (la implementación de los tipos de objeto a transferir) como el *consumidor* (a implementación del sistema de comunicaciones) admiten las facilidades de transferencia fuera de banda proporcionadas por el protocolo *pickle* 5 y mayor.

API de proveedor

Los objetos de datos grandes que se van a serializar con *pickle* deben implementar un método `__reduce_ex__()` especializado para el protocolo 5 y superior, que retorna una instancia de *PickleBuffer* (en lugar de, por ejemplo, un objeto *bytes* object) para cualquier datos.

Un objeto *PickleBuffer* indica que el búfer subyacente es elegible para la transferencia de datos fuera de banda. Estos objetos siguen siendo compatibles con el uso normal del módulo *pickle*. Sin embargo, los consumidores también pueden optar por decirle a *pickle* que manejarán esos búferes por sí mismos.

API de consumidor

Un sistema de comunicaciones puede permitir el manejo personalizado de los objetos *PickleBuffer* generados al serializar un gráfico de objetos.

En el lado del envío, necesita pasar un argumento *buffer_callback* a *Pickler* (o a las funciones *dump()* o *dumps()*), que se llamará con cada *PickleBuffer* generado al hacer *pickling* del gráfico del objeto. Los búferes acumulados por *buffer_callback* no verán sus datos copiados en el flujo de *pickle*, solo se insertará un marcador barato.

En el lado receptor, necesita pasar un argumento *buffers* a *Unpickler* (o a las funciones *load()* o *loads()*), que es un iterable de los búferes que fueron pasado a *buffer_callback*. Ese iterable debería producir búferes en el mismo orden en que se pasaron a *buffer_callback*. Esos búferes proporcionarán los datos esperados por los reconstructores de los objetos cuyo *pickling* produjo los objetos originales *PickleBuffer*.

Entre el lado de envío y el lado de recepción, el sistema de comunicaciones es libre de implementar su propio mecanismo de transferencia para memorias intermedias fuera de banda. Las posibles optimizaciones incluyen el uso de memoria compartida o compresión dependiente del tipo de datos.

Ejemplo

Aquí hay un ejemplo trivial donde implementamos una subclase `bytearray` capaz de participar en el *pickling* de un búfer fuera de banda:

```
class ZeroCopyByteArray(bytearray):

    def __reduce_ex__(self, protocol):
        if protocol >= 5:
            return type(self)._reconstruct, (PickleBuffer(self),), None
        else:
            # PickleBuffer is forbidden with pickle protocols <= 4.
            return type(self)._reconstruct, (bytearray(self),)

    @classmethod
    def _reconstruct(cls, obj):
        with memoryview(obj) as m:
            # Get a handle over the original buffer object
            obj = m.obj
            if type(obj) is cls:
                # Original buffer object is a ZeroCopyByteArray, return it
                # as-is.
                return obj
            else:
                return cls(obj)
```

El reconstructor (el método de clase `_reconstruct`) retorna el objeto que proporciona el búfer si tiene el tipo correcto. Esta es una manera fácil de simular el comportamiento de copia cero en este ejemplo de juguete.

En el lado del consumidor, podemos serializar con *pickle* esos objetos de la forma habitual, que cuando no se serializan nos dará una copia del objeto original:

```
b = ZeroCopyByteArray(b"abc")
data = pickle.dumps(b, protocol=5)
new_b = pickle.loads(data)
print(b == new_b) # True
print(b is new_b) # False: a copy was made
```

Pero si pasamos un *buffer_callback* y luego retornamos los búferes acumulados al anular la serialización, podemos recuperar el objeto original:

```
b = ZeroCopyByteArray(b"abc")
buffers = []
data = pickle.dumps(b, protocol=5, buffer_callback=buffers.append)
new_b = pickle.loads(data, buffers=buffers)
print(b == new_b) # True
print(b is new_b) # True: no copy was made
```

Este ejemplo está limitado por el hecho de que `bytearray` asigna su propia memoria: no puedes crear una instancia de `bytearray` que esté respaldada por la memoria de otro objeto. Sin embargo, los tipos de datos de terceros, como las matrices NumPy no tienen esta limitación y permiten el uso de *pickling* de copia cero (o realizar la menor cantidad de copias posible) cuando se transfieren entre procesos o sistemas distintos.

Ver también:

PEP 574 – Protocolo Pickle 5 con datos fuera de banda

12.1.8 Restricción de Globals

De forma predeterminada, el *unpickling* importará cualquier clase o función que encuentre en los datos de *pickle*. Para muchas aplicaciones, este comportamiento es inaceptable, ya que permite al *unpickler* importar e invocar código arbitrario. Solo considere lo que hace este flujo de datos de *pickle* hechos a mano cuando se carga:

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
hello world
0
```

En este ejemplo, el *unpickler* importa la función `os.system()` y luego aplica el argumento de cadena «echo hello world». Aunque este ejemplo es inofensivo, no es difícil imaginar uno que pueda dañar su sistema.

Por esta razón, es posible que desee controlar lo que se deserializa con *pickle* personalizando *Unpickler.find_class()*. A diferencia de lo que sugiere su nombre, *Unpickler.find_class()* se llama siempre que se solicita un global (es decir, una clase o una función). Por lo tanto, es posible prohibir completamente los globales o restringirlos a un subconjunto seguro.

Aquí hay un ejemplo de un *unpickler* que permite cargar solo unas pocas clases seguras del módulo *builtins*:

```
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                      (module, name))

def restricted_loads(s):
    """Helper function analogous to pickle.loads()."""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

Un uso de muestra de nuestro *unpickler* trabajando tiene la intención de:

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
Traceback (most recent call last):
...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'getattr(__import__("os"), "system")\'
...                  b'("echo hello world")\'\'ntR.')
Traceback (most recent call last):
...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

Como muestran nuestros ejemplos, debes tener cuidado con lo que permites que se deserialice con *pickle*. Por lo tanto,

si la seguridad es un problema, puede considerar alternativas como la API de *marshalling* en `xmlrpc.client` o soluciones de terceros.

12.1.9 Performance

Las versiones recientes del protocolo *pickle* (desde el protocolo 2 en adelante) cuentan con codificaciones binarias eficientes para varias características comunes y tipos integrados. Además, el módulo *pickle* tiene un optimizador transparente escrito en C.

12.1.10 Ejemplos

Para obtener el código más simple, use las funciones `dump()` y `load()`.

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

El siguiente ejemplo lee los datos serializados con *pickle* resultantes.

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

Ver también:

Módulo *copyreg* Registro de constructor de interfaz *Pickle* para tipos de extensión.

Módulo *pickletools* Herramientas para trabajar y analizar datos serializados con *pickle*.

Módulo *shelve* Bases de datos indexadas de objetos; usa *pickle*.

Module *copy* Copia de objetos superficial y profunda.

Módulo *marshal* Serialización de alto rendimiento de tipos integrados.

Notas al pie

12.2 copyreg — Registrar funciones de soporte de pickle

Código fuente: [Lib/copyreg.py](#)

El módulo *copyreg* ofrece una manera de definir funciones usada cuando se serializan (*pickling*) objetos específicos. Los módulos *pickle* y *copy* utilizan estas funciones cuando se realizan acciones de serializado/copiado en esos objetos. El módulo provee información de configuración acerca de los objetos constructores, los cuales no son clases. Estos objetos constructores pueden ser funciones de fábrica o instancias de clase.

`copyreg.constructor` (*object*)

Declara que el *object* debe ser un constructor válido. Si el *object* no es invocable (y por lo tanto, no es válido como constructor), lanza una excepción `TypeError`.

`copyreg.pickle` (*type, function, constructor=None*)

Declara que la *function* deber ser usada como una función de «reducción» para objetos de tipo *type*. La *function* debe retornar ya sea una cadena de caracteres o una tupla que contenga dos o tres elementos.

El parámetro opcional *constructor*, si se proporciona, es un objeto invocable el cual, que puede ser usado para reconstruir el objeto cuando se llama con la tupla de argumentos retornados por la *function* en el momento de pickling. La excepción `TypeError` se lanzará si el *objeto* es una clase o si el *constructor* no es invocable.

Consulte el módulo `pickle` para más detalles sobre la interfaz esperada de *function* y *constructor*. Note que el atributo `dispatch_table` de un objeto pickler o subclase de `pickle.Pickler` puede también ser utilizado para declarar funciones de reducción.

12.2.1 Ejemplo

El siguiente ejemplo pretende mostrar cómo registrar una función pickle y cómo se utilizará:

```
>>> import copyreg, 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,)
...
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

12.3 `shelve` — Persistencia de objetos de Python

Source code: `Lib/shelve.py`

Un «estante» o *shelve*, es un objeto persistente similar a un diccionario. La diferencia con las bases de datos «dbm» es que los valores (¡no las claves!) en un estante pueden ser esencialmente objetos Python arbitrarios — cualquier cosa que el módulo `pickle` pueda manejar. Esto incluye la mayoría de las instancias de clase, tipos de datos recursivos y objetos que contienen muchos subobjetos compartidos. Las claves son cadenas ordinarias.

`shelve.open` (*filename, flag='c', protocol=None, writeback=False*)

Abre un diccionario persistente. El nombre de archivo especificado es el nombre de archivo base para la base de datos subyacente. Como efecto secundario, se puede agregar una extensión al nombre de archivo y se puede crear más de un archivo. De forma predeterminada, el archivo de base de datos subyacente se abre para leer y escribir. El parámetro opcional *flag* tiene la misma interpretación que el parámetro *flag* de `dbm.open()`.

De forma predeterminada, los *pickles* de la versión 3 se utilizan para serializar valores. La versión del protocolo *pickle* se puede especificar con el parámetro *protocol*.

Debido a la semántica de Python, un estante no puede saber cuándo se modifica una entrada de diccionario persistente mutable. De forma predeterminada, los objetos modificados se escriben *sólo* cuando se asignan al estante (consulte [Ejemplo](#)). Si el parámetro opcional *writeback* se establece en `True`, todas las entradas a las que se accede también se almacenan en caché en la memoria y se vuelven a escribir en `sync()` y `close()`;

esto puede hacer que sea más práctico mutar entradas mutables en el diccionario persistente, pero, si se accede a muchas entradas, puede consumir grandes cantidades de memoria para la memoria caché, y puede hacer que la operación de cierre sea muy lenta ya que todas las entradas a las que se accede se vuelven a escribir (no hay manera de determinar qué entradas a las que se accede son mutables, ni cuáles se mutaron realmente).

Nota: No confíe en que el estante se cerrará automáticamente; siempre llame a `close()` explícitamente cuando ya no lo necesite, o use `shelve.open()` como administrador de contexto:

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

Advertencia: Debido a que el módulo `shelve` está respaldado por `pickle`, es inseguro cargar un estante desde una fuente que no es de confianza. Al igual que con el `pickle`, cargar un estante puede ejecutar código arbitrario.

Los objetos de estante admiten todos los métodos admitidos por los diccionarios. Esto facilita la transición de scripts basados en diccionarios a aquellos que requieren almacenamiento persistente.

Se admiten dos métodos adicionales:

`Shelf.sync()`

Escriba todas las entradas en la caché si el estante se abrió con `writeback` establecido en `True`. También vacíe la caché y sincronice el diccionario persistente en el disco, si es posible. Esto se llama automáticamente cuando el estante se cierra con `close()`.

`Shelf.close()`

Sincronice y cierre el objeto persistente `dict`. Las operaciones en un estante cerrado fallarán con un `ValueError`.

Ver también:

Receta de diccionario persistente <<https://code.activestate.com/recipes/576642/>> _ con formatos de almacenamiento ampliamente compatibles y con la velocidad de los diccionarios nativos.

12.3.1 Restricciones

- La elección de qué paquete de base de datos se utilizará (como `dbm.ndbm` o `dbm.gnu`) depende de la interfaz disponible. Por lo tanto, no es seguro abrir la base de datos directamente usando `dbm`. La base de datos también está (desafortunadamente) sujeta a las limitaciones de `dbm`, si se usa — esto significa que (la representación `pickle` de) los objetos almacenados en la base de datos debe ser bastante pequeña, y en casos raros las colisiones de claves pueden hacer que la base de datos rechace las actualizaciones.
- El módulo `shelve` no admite el acceso *concurrent* de lectura/escritura a los objetos almacenados. (Los accesos de lectura múltiples simultáneos son seguros). Cuando un programa tiene un estante abierto para escritura, ningún otro programa debe tenerlo abierto para lectura o escritura. El bloqueo de archivos Unix se puede usar para resolver esto, pero esto difiere entre las versiones de Unix y requiere conocimiento sobre la implementación de la base de datos utilizada.

class `shelve.Shelf` (`dict`, `protocol=None`, `writeback=False`, `keyencoding='utf-8'`)

Una subclase de `collections.abc.MutableMapping` que almacena valores `pickle` en el objeto `dict`.

De forma predeterminada, los `pickles` de la versión 3 se utilizan para serializar valores. La versión del protocolo `pickle` se puede especificar con el parámetro `protocol`. Vea la documentación de `pickle` para una discusión de los protocolos de `pickle`.

Si el parámetro `writeback` es `True`, el objeto mantendrá un caché de todas las entradas a las que se accedió y las volverá a escribir en el `dict` en las horas de sincronización y cierre. Esto permite operaciones naturales en

entradas mutables, pero puede consumir mucha más memoria y hacer que la sincronización y el cierre tomen mucho tiempo.

El parámetro *keyencoding* es la codificación utilizada para codificar las claves antes de que se utilicen con el *dict* subyacente.

El objeto *Shelf* también se puede utilizar como administrador de contexto, en cuyo caso se cerrará automáticamente cuando finalice el bloque *with*.

Distinto en la versión 3.2: Se agregó el parámetro *keyencoding*; anteriormente, las claves siempre estaban codificadas en UTF-8.

Distinto en la versión 3.4: Agregado soporte para administrador de contexto.

class `shelve.BsdDbShelf` (*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

A subclass of *Shelf* which exposes `first()`, `next()`, `previous()`, `last()` and `set_location()` which are available in the third-party `bsddb` module from `pybsddb` 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*, *writeback*, and *keyencoding* parameters have the same interpretation as for the *Shelf* class.

class `shelve.DbfilenameShelf` (*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

Una subclase de *Shelf* que acepta un *filename* en lugar de un objeto tipo diccionario (*dict*). El archivo subyacente se abrirá usando `dbm.open()`. De forma predeterminada, el archivo se creará y se abrirá tanto para lectura como para escritura. El parámetro opcional *flag* tiene la misma interpretación que para la función `open()`. Los parámetros opcionales *protocol* y *writeback* tienen la misma interpretación que para la clase *Shelf*.

12.3.2 Ejemplo

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 = key in d             # true if the key exists
klist = list(d.keys())      # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]         # this works as expected, but...
d['xx'].append(3)           # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# 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
```

Ver también:

Módulo `dbm` Interfaz genérica para bases de datos estilo dbm.

Módulo `pickle` Serialización de objetos utilizada por `shelve`.

12.4 `marshal` — Serialización interna de objetos Python

Este módulo contiene funciones que pueden leer y escribir valores de Python en un formato binario. El formato es específico de Python, pero independiente de los problemas de arquitectura de la máquina (por ejemplo, puede escribir un valor de Python en un archivo en un PC, transportar el archivo a un Sun y leerlo allí). Los detalles del formato están indocumentados a propósito; pueden cambiar entre las versiones de Python (aunque rara vez lo hacen).¹

Este no es un módulo general de «persistencia». Para la persistencia general y la transferencia de objetos Python a través de llamadas RPC, consulte los módulos `pickle` y `shelve`. El módulo `marshal` existe principalmente para admitir la lectura y escritura del código «pseudocompilado» para los módulos Python de archivos `.pyc`. Por lo tanto, los mantenedores de Python se reservan el derecho de modificar el formato de cálculo de referencias de manera incompatible hacia atrás en caso de necesidad. Si va a serializar y deserializar objetos Python, utilice el módulo `pickle` en su lugar: el rendimiento es comparable, la independencia de la versión está garantizada y `pickle` admite una gama sustancialmente más amplia de objetos que `marshal`.

Advertencia: El módulo `marshal` no está destinado a ser seguro contra datos erróneos o contruidos maliciosamente. Nunca deserializar con `marshal` los datos recibidos de una fuente no confiable o no autenticada.

No se admiten todos los tipos de objetos de Python; en general, este módulo solo puede escribir y leer objetos cuyo valor es independiente de una invocación concreta de Python. Se admiten los siguientes tipos: booleanos, enteros, números de punto flotante, números complejos, cadenas de caracteres, bytes, arrays de bytes (`bytearray`), tuplas, listas, conjuntos (`set` y `frozenset`), diccionarios y objetos de código, donde se debe entender que las tuplas, listas, conjuntos y diccionarios solo se admiten siempre que se admitan los valores contenidos en ellos mismos. Los singletons `None`, `Ellipsis` y `StopIteration` también pueden ser marshalled y unmarshalled. Para el formato *versión* inferior a 3, no se pueden escribir listas, conjuntos ni diccionarios recursivos (véase más adelante).

Hay funciones que leen/escriben archivos, así como funciones que operan en objetos similares a bytes.

El módulo define estas funciones:

`marshal.dump(value, file[, version])`

Escribe el valor en el archivo abierto. El valor debe ser un tipo admitido. El archivo debe ser un archivo *binary file* en el que se pueda escribir.

Si el valor tiene (o contiene un objeto que tiene) un tipo no admitido, se produce una excepción `ValueError` — pero los datos no utilizados también se escribirán en el archivo. El objeto no será leído correctamente por `load()`.

El argumento *version* indica el formato de datos que `dump` debe usar (véase más adelante).

`marshal.load(file)`

Lee un valor del archivo abierto y lo retorna. Si no se lee ningún valor válido (por ejemplo, porque los datos tienen un formato de cálculo de referencias incompatible de una versión de Python diferente), lanza una excepción `EOFError`, `ValueError` o `TypeError`. El archivo debe ser un *binary file* habilitado para lectura.

Nota: Si un objeto que contiene un tipo no admitido se calcula con `dump()`, `load()` sustituirá `None` por el tipo «unmarshallable».

¹ El nombre de este módulo proviene de algunos términos utilizados por los diseñadores de Modula-3 (entre otros), que utilizan el término «marshalling» para el envío de datos de forma auto-contenida. Estrictamente hablando «marshalling», significa convertir algunos datos internos en un formato externo (por ejemplo, en un búfer RPC) y «unmarshalling» es el proceso inverso.

`marshal.dumps` (*value* [, *version*])

Retorna el objeto bytes que se escribiría en un archivo mediante `dump` (*value*, *file*). El valor debe ser un tipo admitido. Lanza una excepción `ValueError` si *value* tiene (o contiene un objeto que tiene) un tipo no admitido.

El argumento *version* indica el formato de datos que `dumps` debe usar (véase más adelante).

`marshal.loads` (*bytes*)

Convierte el objeto *bytes-like object* en un valor. Si no se encuentra ningún valor válido, lanza una excepción `EOFError`, `ValueError` o `TypeError`. Se omiten los bytes adicionales de la entrada.

Además, se definen las siguientes constantes:

`marshal.version`

Indica el formato que utiliza el módulo. La versión 0 es el formato histórico, la versión 1 comparte cadenas internadas y la versión 2 utiliza un formato binario para los números de punto flotante. La versión 3 agrega compatibilidad con la creación de instancias de objetos y la recursividad. La versión actual es 4.

Notas al pie

12.5 dbm — Interfaces para «bases de datos» de Unix

Código fuente: `Lib/dbm/__init__.py`

`dbm` es una interfaz genérica para variantes de la base de datos DBM — `dbm.gnu` o `dbm.ndbm`. Si ninguno de estos módulos son instalados, se utilizará la implementación lenta pero sencilla en el módulo `dbm.dumb`. Existe una [interfaz de terceros](#) para la Oracle Berkeley DB.

exception `dbm.error`

Una tupla que contiene las excepciones que pueden ser lanzadas por cada uno de los módulos soportados, con una excepción única también denominada `dbm.error` como el primer elemento — el último se usa cuando se genera `dbm.error`.

`dbm.whichdb` (*filename*)

Esta función intenta adivinar cuál de los varios módulos de base de datos simples disponibles — `dbm.gnu`, `dbm.ndbm` o `dbm.dumb` — deberán usarse para abrir un archivo.

Retorna uno de los siguientes valores: `None` si el archivo no se puede abrir porque no se puede leer o no existe; la cadena de caracteres vacía (`' '`) si no se puede adivinar el formato del archivo; o una cadena de caracteres que contenga el nombre del módulo requerido, como `'dbm.ndbm'` o `'dbm.gnu'`.

`dbm.open` (*file*, *flag*='r', *mode*=0o666)

Abre el archivo *file* de la base de datos y retorna un objeto correspondiente.

Si el archivo de la base de datos existe, la función `whichdb()` es usada para determinar su tipo y el módulo apropiado se utiliza; sino existe, se utiliza el primer módulo listado anteriormente que se puede importar.

El argumento opcional *flag* puede ser:

Valor	Significado
'r'	Abre la base de datos existente solo para lectura (predeterminado)
'w'	Abre la base de datos existente para leer y escribir
'c'	Abre la base de datos para lectura y escritura, creándola si no existe
'n'	Siempre cree una base de datos nueva, vacía, abierta para lectura y escritura

El argumento opcional *mode* es el modo Unix del archivo, usado solamente cuando la base de datos tiene que ser creada. Su valor predeterminado es octal 0o666 (y será modificado por el `umask` vigente).

El objeto retornado por `open()` admite la misma funcionalidad básica que los diccionarios; las claves y sus valores correspondientes se pueden almacenar, recuperar y eliminar, y el operador `in` y el método `keys()` están disponibles, así como `get()` y `setdefault()`.

Distinto en la versión 3.2: `get()` y `setdefault()` ya están disponibles en todos los módulos de base de datos.

Distinto en la versión 3.8: Al eliminar una clave de una base de datos de solo lectura lanza un error específico del módulo de la base de datos en lugar de `KeyError`.

La clave y los valores siempre se almacenan como bytes. Esto significa que cuando se utilizan cadenas de caracteres, se convierten implícitamente a la codificación predeterminada antes de almacenarse.

Estos objetos también admiten el uso en una instrucción `with`, que los cerrará automáticamente cuando termine.

Distinto en la versión 3.4: Se agregó soporte nativo para el protocolo de administración de contexto a los objetos retornados por `open()`.

El siguiente ejemplo registra algunos nombres de host y un título correspondiente, y luego imprime el contenido de la base de datos:

```
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:

    # Record some values
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # Note that the keys are considered bytes now.
    assert db[b'www.python.org'] == b'Python Website'
    # Notice how the value is now in bytes.
    assert db['www.cnn.com'] == b'Cable News Network'

    # Often-used methods of the dict interface work too.
    print(db.get('python.org', b'not present'))

    # Storing a non-string key or value will raise an exception (most
    # likely a TypeError).
    db['www.yahoo.com'] = 4

# db is automatically closed when leaving the with statement.
```

Ver también:

Módulo `shelve` Módulo de persistencia que almacena datos que no son cadenas de caracteres.

Los submódulos individuales se describen en las siguientes secciones.

12.5.1 `dbm.gnu` — La reinterpretación de GNU de `dbm`

Código fuente: `Lib/dbm/gnu.py`

Este módulo es bastante similar al módulo `dbm`, pero usa la biblioteca GNU `gdbm` en su lugar para proporcionar alguna funcionalidad adicional. Tenga en cuenta que los formatos de archivo creados por `dbm.gnu` y `dbm.ndbm` son incompatibles.

El módulo `dbm.gnu` proporciona una interfaz a la biblioteca GNU DBM. Los objetos `dbm.gnu.gdbm` se comportan como asignaciones (diccionarios), excepto que las claves y los valores siempre se convierten a bytes antes de almacenarlos. La impresión de un objeto `gdbm` no imprime las llaves y los valores, y los métodos `items()` y `values()` no son compatibles.

exception `dbm.gnu.error`

Se lanza en errores específicos `dbm.gnu`, como errores de E/S. `KeyError` se genera para errores generales de asignación, como especificar una clave incorrecta.

`dbm.gnu.open(filename[, flag[, mode]])`

Abre una base de datos `gdbm` y retorna un objeto `gdbm`. El argumento *filename* es el nombre del archivo de la base de datos.

El argumento opcional *flag* puede ser:

Valor	Significado
'r'	Abre la base de datos existente solo para lectura (predeterminado)
'w'	Abre la base de datos existente para leer y escribir
'c'	Abre la base de datos para lectura y escritura, creándola si no existe
'n'	Siempre cree una base de datos nueva, vacía, abierta para lectura y escritura

Los siguientes caracteres adicionales se pueden agregar al *flag* para controlar cómo se abre la base de datos:

Va- lor	Significado
'f'	Abre la base de datos en modo rápido. Las escrituras en la base de datos no se sincronizarán.
's'	Modo sincronizado. Esto hará que los cambios en la base de datos se escriban inmediatamente en el archivo.
'u'	No bloquea la base de datos.

No todos los *flags* son válidas para todas las versiones de `gdbm`. La constante del módulo `open_flags` es una cadena de caracteres de *flags* soportadas. La excepción `error` se lanza si se especifica un *flag* no válido.

El argumento opcional *mode* es el modo Unix del archivo, usado solo cuando la base de datos tiene que ser creada. Su valor predeterminado es octal `0o666`.

Además de los métodos similares a los diccionarios, los objetos `gdbm` tienen los siguientes métodos:

`gdbm.firstkey()`

Es posible recorrer cada clave en la base de datos usando este método y el método `nextkey()`. El recorrido está ordenado por los valores hash internos de `gdbm` y no se ordenará por los valores clave. Este método retorna la clave de inicio.

`gdbm.nextkey(key)`

Retorna la clave que sigue a *key* en el recorrido. El siguiente código imprime todas las claves en la base de datos `db`, sin tener que crear una lista en la memoria que las contenga todas:

```
k = db.firstkey()
while k != None:
    print(k)
    k = db.nextkey(k)
```

`gdbm.reorganize()`

Si ha realizado muchas eliminaciones y desea reducir el espacio utilizado por el archivo `gdbm`, esta rutina reorganizará la base de datos. Los objetos `gdbm` no acortarán la longitud de un archivo de base de datos excepto al usar esta reorganización; de lo contrario, el espacio de archivo eliminado se conservará y reutilizará cuando se agreguen nuevos pares (clave, valor).

`gdbm.sync()`

Cuando la base de datos se ha abierto en modo rápido, este método obliga a que los datos no escritos se escriban en el disco.

`gdbm.close()`

Cierra la base de datos `gdbm`.

12.5.2 dbm.ndbm — Interfaz basada en ndbm

Código fuente: [Lib/dbm/ndbm.py](#)

El módulo `dbm.ndbm` proporciona una interfaz a la biblioteca «(n)dbm» de Unix. Los objetos DBM se comportan como asignaciones (diccionarios), excepto que las claves y los valores siempre se almacenan como bytes. La impresión de un objeto `dbm` no imprime las claves y los valores, y los métodos `items()` y `values()` no son compatibles.

Este módulo se puede utilizar con la interfaz `ndbm` «clásica» o la interfaz de compatibilidad GNU GDBM. En Unix, el código **configure** intentará localizar el archivo de encabezado apropiado para simplificar la construcción de este módulo.

exception `dbm.ndbm.error`

Se lanza en errores específicos `dbm.ndbm`, como errores de E/S. `KeyError` lanza para errores generales de asignación, como especificar una clave incorrecta.

`dbm.ndbm.library`

Nombre de la biblioteca de implementación `ndbm` utilizada.

`dbm.ndbm.open(filename[, flag[, mode]])`

Abre una base de datos `dbm` y retorna un objeto `ndbm`. El argumento **filename** es el nombre de la base de datos (sin las extensiones `.dir` y `.pag`).

El argumento *flag* opcional debe ser uno de estos valores:

Valor	Significado
'r'	Abre la base de datos existente solo para lectura (predeterminado)
'w'	Abre la base de datos existente para leer y escribir
'c'	Abre la base de datos para lectura y escritura, creándola si no existe
'n'	Siempre cree una base de datos nueva, vacía, abierta para lectura y escritura

El argumento opcional *mode* es el modo Unix del archivo, usado solamente cuando la base de datos tiene que ser creada. Su valor predeterminado es octal `0o666` (y será modificado por el `umask` vigente).

Además de los métodos similares a los de un diccionario, los objetos `ndbm` proporcionan el siguiente método:

`ndbm.close()`

Cierra la base de datos `ndbm`.

12.5.3 dbm.dumb — Implementación de DBM portátil

Código fuente: [Lib/dbm/dumb.py](#)

Nota: El módulo `dbm.dumb` está pensado como último recurso para el módulo `dbm` cuando no hay disponible un módulo más robusto. El módulo `dbm.dumb` no está escrito para velocidad y no se usa tanto como los otros módulos de base de datos.

El módulo `dbm.dumb` proporciona una interfaz persistente similar a un diccionario que está escrita completamente en Python. A diferencia de otros módulos como `dbm.gnu`, no se requiere una biblioteca externa. Al igual que con otras asignaciones persistentes, las claves y los valores siempre se almacenan como bytes.

El módulo define lo siguiente:

exception `dbm.dumb.error`

Se lanza en errores específicos `dbm.dumb`, como errores de E/S. `KeyError` se lanza para errores de mapeo generales como especificar una clave incorrecta.

`dbm.dumb.open(filename[, flag[, mode]])`

Abre una base de datos `dumbdbm` y retorna un objeto `dumbdbm`. El argumento del *filename* es el nombre base del archivo de la base de datos (sin extensiones específicas). Cuando una base de datos `dumbdbm` se crea, archivos con la extensión `.dat` y `.dir` se crean.

El argumento opcional *flag* puede ser:

Valor	Significado
'r'	Abre la base de datos existente solo para lectura (predeterminado)
'w'	Abre la base de datos existente para leer y escribir
'c'	Abre la base de datos para lectura y escritura, creándola si no existe
'n'	Siempre cree una base de datos nueva, vacía, abierta para lectura y escritura

El argumento opcional *mode* es el modo Unix del archivo, usado solamente cuando la base de datos tiene que ser creada. Su valor predeterminado es octal `0o666` (y será modificado por el `umask` vigente).

Advertencia: Es posible bloquear el intérprete de Python cuando se carga una base de datos con una entrada suficientemente grande/compleja debido a las limitaciones de profundidad de la pila en el compilador AST de Python.

Distinto en la versión 3.5: `open()` siempre crea una nueva base de datos cuando el flag tiene valor de `'n'`.

Distinto en la versión 3.8: Una base de datos abierta con flags `'r'` ahora es de solo lectura. Abrir con los flags `'r'` y `'w'` ya no crea una base de datos si no existe.

Además de los métodos proporcionados por la clase `collections.abc.MutableMapping`, los objetos `dumbdbm` proporcionan los siguientes métodos:

`dumbdbm.sync()`

Sincroniza el directorio en disco y los archivos de datos. Este método es llamado por el método `Shelve.sync()`.

`dumbdbm.close()`

Cierra la base de datos `dumbdbm`.

12.6 sqlite3 — DB-API 2.0 interfaz para bases de datos SQLite

Código fuente: [Lib/sqlite3/](#)

SQLite es una biblioteca de C que provee una base de datos ligera basada en disco que no requiere un proceso de servidor separado y permite acceder a la base de datos usando una variación no estándar del lenguaje de consulta SQL. Algunas aplicaciones pueden usar SQLite para almacenamiento interno. También es posible prototipar una aplicación usando SQLite y luego transferir el código a una base de datos más grande como PostgreSQL u Oracle.

El módulo `sqlite3` fue escrito por *Gerhard Häring*. Provee una interfaz SQL compatible con la especificación DB-API 2.0 descrita por [PEP 249](#).

Para usar el módulo, primero se debe crear un objeto `Connection` que representa la base de datos. Aquí los datos serán almacenados en el archivo `example.db`:

```
import sqlite3
con = sqlite3.connect('example.db')
```

También se puede agregar el nombre especial `:memory:` para crear una base de datos en memoria RAM.

Una vez se tenga una `Connection`, se puede crear un objeto `Cursor` y llamar su método `execute()` para ejecutar comandos SQL:

```

cur = con.cursor()

# Create table
cur.execute('''CREATE TABLE stocks
              (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
cur.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
con.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.
con.close()

```

Los datos guardados son persistidos y están disponibles en sesiones posteriores:

```

import sqlite3
con = sqlite3.connect('example.db')
cur = con.cursor()

```

Usualmente, las operaciones SQL necesitarán usar valores de variables de Python. No se debe ensamblar la consulta usando operaciones de cadena de Python porque hacerlo es inseguro; vuelve el programa vulnerable a ataques de inyección SQL (ver este divertido ejemplo de lo que puede salir mal: <https://xkcd.com/327/>)

En cambio, se usan los parámetros de sustitución DB-API. Colocar `?` como un marcador de posición en el lugar donde se usara un valor, y luego se provee una tupla de valores como segundo argumento del método del cursor `execute()` (otros módulos de bases de datos pueden usar un marcado de posición diferente, como `%s` o `:1`). Por ejemplo:

```

# Never do this -- insecure!
symbol = 'RHAT'
cur.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)

# Do this instead
t = ('RHAT',)
cur.execute('SELECT * FROM stocks WHERE symbol=?', t)
print(cur.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),
              ]
cur.executemany('INSERT INTO stocks VALUES (?, ?, ?, ?, ?)', purchases)

```

Para obtener los datos luego de ejecutar una sentencia `SELECT`, se puede tratar el cursor como un *iterator*, llamar el método del cursor `fetchone()` para obtener un solo registro, o llamar `fetchall()` para obtener una lista de todos los registros.

Este ejemplo usa la forma con el iterador:

```

>>> for row in cur.execute('SELECT * FROM stocks ORDER BY price'):
      print(row)

('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
('2006-04-06', 'SELL', 'IBM', 500, 53.0)
('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)

```

Ver también:

<https://www.sqlite.org> La página web SQLite; la documentación describe la sintaxis y los tipos de datos disponibles para el lenguaje SQL soportado.

<https://www.w3schools.com/sql/> Tutorial, referencia y ejemplos para aprender sintaxis SQL.

PEP 249 - Especificación de la API 2.0 de base de datos PEP escrito por Marc-André Lemburg.

12.6.1 Funciones y constantes del módulo

`sqlite3.version`

El número de versión de este módulo, como una cadena de caracteres. Este no es la versión de la librería SQLite.

`sqlite3.version_info`

El número de versión de este módulo, como una tupla de enteros. Este no es la versión de la librería SQLite.

`sqlite3.sqlite_version`

El número de versión de la librería SQLite en tiempo de ejecución, como una cadena de caracteres.

`sqlite3.sqlite_version_info`

El número de versión de la librería SQLite en tiempo de ejecución, como una tupla de enteros.

`sqlite3.PARSE_DECLTYPES`

Esta constante se usa con el parámetro *detect_types* de la función `connect()`.

Configurarla hace que el módulo `sqlite3` analice el tipo declarado para cada columna que retorna. Este convertirá la primera palabra del tipo declarado, i. e. para «*integer primary key*», será convertido a «*integer*», o para «*number(10)*» será convertido a «*number*». Entonces para esa columna, revisará el diccionario de conversiones y usará la función de conversión registrada para ese tipo.

`sqlite3.PARSE_COLNAMES`

Esta constante se usa con el parámetro *detect_types* de la función `connect()`.

Configurar esto hace que la interfaz de SQLite analice el nombre para cada columna que retorna, buscare una cadena de caracteres [mytype], y decidirá cual “mytype” es el tipo de la columna. Tratará de encontrar una entrada “mytype” en el diccionario de conversiones y luego usar la función de conversión encontrada allí y retornar el valor. El nombre de la columna encontrada en `Cursor.description` no incluye el tipo, en otras palabras, si se usa algo como 'as 'Expiration date [datetime]'' en el SQL, entonces analizará todo lo demás hasta el primer '[' para el nombre de la columna y eliminará el espacio anterior: el nombre de la columna sería: «Expiration date».

`sqlite3.connect(database[, timeout, detect_types, isolation_level, check_same_thread, factory, cached_statements, uri])`

Abre una conexión al archivo de base de datos SQLite *database*. Por defecto retorna un objeto `Connection`, a menos que se indique un *factory* personalizado.

database es un *path-like object* indicando el nombre de ruta (absoluta o relativa al directorio de trabajo actual) del archivo de base de datos abierto. Se puede usar `":memory:"` para abrir una conexión de base de datos a una base de datos que reside en memoria RAM en lugar que disco.

Cuando una base de datos es accedida por múltiples conexiones, y uno de los procesos modifica la base de datos, la base de datos SQLite se bloquea hasta que la transacción se confirme. El parámetro *timeout* especifica que tanto debe esperar la conexión para que el bloqueo desaparezca antes de lanzar una excepción. Por defecto el parámetro *timeout* es de 5.0 (cinco segundos).

Para el parámetro *isolation_level*, por favor ver la propiedad *isolation_level* del objeto `Connection`.

De forma nativa SQLite soporta solo los tipos `TEXT`, `INTEGER`, `*REAL*`, `*BLOB*` y `NULL`. Si se quiere usar otros tipos, debe soportarlos usted mismo. El parámetro *detect_types* y el uso de **converters** personalizados registrados con la función a nivel del módulo `register_converter()` permite hacerlo fácilmente.

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. Due to SQLite behaviour, types can't be detected for generated fields (for example `max(data)`), even when *detect_types* parameter is set. In such case, the returned type is `str`.

Por defecto, `check_same_thread` es `True` y únicamente el hilo creado puede utilizar la conexión. Si se configura `False`, la conexión retornada podrá ser compartida con múltiples hilos. Cuando se utilizan múltiples hilos con la misma conexión, las operaciones de escritura deberán ser serializadas por el usuario para evitar corrupción de datos.

Por defecto el módulo `sqlite3` utiliza su propia clase `Connection` para la llamada de conexión. Sin embargo se puede crear una subclase de `Connection` y hacer que `connect()` use su clase en lugar de proveer la suya en el parámetro `factory`.

Consulte la sección [SQLite y tipos de Python](#) de este manual para más detalles.

El módulo `sqlite3` internamente usa cache de declaraciones para evitar un análisis SQL costoso. Si se desea especificar el número de sentencias que estarán en memoria caché para la conexión, se puede configurar el parámetro `cached_statements`. Por defecto están configurado para 100 sentencias en memoria caché.

Si `uri` es verdadero, la `database` se interpreta como una `URI`. Esto permite especificar opciones. Por ejemplo, para abrir la base de datos en modo solo lectura puedes usar:

```
db = sqlite3.connect('file:path/to/database?mode=ro', uri=True)
```

Más información sobre esta característica, incluyendo una lista de opciones reconocidas, pueden encontrarse en [la documentación de SQLite URI](#).

Lanza un *evento de auditoría* `sqlite3.connect` con argumento `database`.

Distinto en la versión 3.4: Agregado el parámetro `uri`.

Distinto en la versión 3.7: `database` ahora también puede ser un *path-like object*, no solo una cadena de caracteres.

`sqlite3.register_converter` (*typename*, *callable*)

Registra un invocable para convertir un *bytestring* de la base de datos en un tipo Python personalizado. El invocable será invocado por todos los valores de la base de datos que son del tipo *typename*. Conceder el parámetro `detect_types` de la función `connect()` para el funcionamiento de la detección de tipo. Se debe notar que *typename* y el nombre del tipo en la consulta son comparados insensiblemente a mayúsculas y minúsculas.

`sqlite3.register_adapter` (*type*, *callable*)

Registra un invocable para convertir el tipo Python personalizado *type* a uno de los tipos soportados por SQLite's. El invocable *callable* acepta un único parámetro de valor Python, y debe retornar un valor de los siguientes tipos: `int`, `float`, `str` or `bytes`.

`sqlite3.complete_statement` (*sql*)

Retorna `True` si la cadena *sql* contiene una o más sentencias SQL completas terminadas con punto y coma. No se verifica que la sentencia SQL sea sintácticamente correcta, solo que no existan literales de cadenas no cerradas y que la sentencia termine por un punto y coma.

Esto puede ser usado para construir un *shell* para SQLite, como en el siguiente ejemplo:

```
# 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 = input()
    if line == "":
        break
```

(continué en la próxima página)

(proviene de la página anterior)

```

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)`

Por defecto no se obtendrá ningún *tracebacks* en funciones definidas por el usuario, agregaciones, *converters*, autorizador de *callbacks* etc. si se quiere depurarlas, se puede llamar esta función con *flag* configurado a `True`. Después se obtendrán *tracebacks* de los *callbacks* en `sys.stderr`. Usar `False` para deshabilitar la característica de nuevo.

12.6.2 Objetos de conexión

class `sqlite3.Connection`

Una conexión a base de datos SQLite tiene los siguientes atributos y métodos:

isolation_level

Obtener o configurar el actual nivel de insolación. `None` para modo *autocommit* o uno de «DEFERRED», «IMMEDIATE» o «EXCLUSIVO». Ver sección [Controlando Transacciones](#) para una explicación detallada.

in_transaction

`True` si una transacción está activa (existen cambios *uncommitted*), `False` en sentido contrario. Atributo de solo lectura.

Nuevo en la versión 3.2.

cursor (*factory=Cursor*)

El método `cursor` acepta un único parámetro opcional *factory*. Si es agregado, éste debe ser un invocable que retorna una instancia de `Cursor` o sus subclases.

commit()

Este método asigna la transacción actual. Si no se llama este método, cualquier cosa hecha desde la última llamada de `commit()` no es visible para otras conexiones de bases de datos. Si se pregunta el porqué no se ven los datos que escribiste, por favor verifica que no olvidaste llamar este método.

rollback()

Este método retrocede cualquier cambio en la base de datos desde la llamada del último `commit()`.

close()

Este método cierra la conexión a la base de datos. Nótese que éste no llama automáticamente `commit()`. Si se cierra la conexión a la base de datos sin llamar primero `commit()`, los cambios se perderán!

execute (*sql[, parameters]*)

Este es un atajo no estándar que crea un objeto cursor llamando el método `cursor()`, llama su método `execute()` con los *parameters* dados, y retorna el cursor.

executemany (*sql[, parameters]*)

Este es un atajo no estándar que crea un objeto cursor llamando el método `cursor()`, llama su método `executemany()` con los *parameters* dados, y retorna el cursor.

executescript (*sql_script*)

Este es un atajo no estándar que crea un objeto cursor llamando el método `cursor()`, llama su método `executescript()` con el *sql_script*, y retorna el cursor.

create_function (*name*, *num_params*, *func*, *, *deterministic=False*)

Crea una función definida de usuario que se puede usar después desde declaraciones SQL con el nombre de función *name*. *num_params* es el número de parámetros que la función acepta (si *num_params* es -1, la función puede tomar cualquier número de argumentos), y *func* es un invocable de Python que es llamado como la función SQL. Si *deterministic* es verdadero, la función creada es marcada como **deterministic**, lo cual permite a SQLite hacer optimizaciones adicionales. Esta marca es soportada por SQLite 3.8.3 o superior, será lanzado `NotSupportedError` si se usa con versiones antiguas.

La función puede retornar cualquier tipo soportado por SQLite: bytes, str, int, float y None.

Distinto en la versión 3.8: El parámetro *deterministic* fue agregado.

Ejemplo:

```
import sqlite3
import hashlib

def md5sum(t):
    return hashlib.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", (b"foo",))
print(cur.fetchone()[0])

con.close()
```

create_aggregate (*name*, *num_params*, *aggregate_class*)

Crea una función agregada definida por el usuario.

La clase agregada debe implementar un método `step`, el cual acepta el número de parámetros *num_params* (si *num_params* es -1, la función puede tomar cualquier número de argumentos), y un método `finalize` el cual retornará el resultado final del agregado.

El método `finalize` puede retornar cualquiera de los tipos soportados por SQLite: bytes, str, int, float and None.

Ejemplo:

```
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])
```

(continué en la próxima página)

(proviene de la página anterior)

```
con.close()
```

create_collation (*name*, *callable*)

Crea una collation con el *name* y *callable* especificado. El invocable será pasado con dos cadenas de texto como argumentos. Se retornará -1 si el primero esta ordenado menor que el segundo, 0 si están ordenados igual y 1 si el primero está ordenado mayor que el segundo. Nótese que esto controla la ordenación (ORDER BY en SQL) por lo tanto sus comparaciones no afectan otras comparaciones SQL.

Note que el invocable obtiene sus parámetros como Python bytestrings, lo cual normalmente será codificado en UTF-8.

El siguiente ejemplo muestra una *collation* personalizada que ordena «La forma incorrecta»:

```
import sqlite3

def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1

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()
```

Para remover una collation, llama `create_collation` con `None` como invocable:

```
con.create_collation("reverse", None)
```

interrupt ()

Se puede llamar este método desde un hilo diferente para abortar cualquier consulta que pueda estar ejecutándose en la conexión. La consulta será abortada y quien realiza la llamada obtendrá una excepción.

set_authorizer (*authorizer_callback*)

Esta rutina registra un callback. El callback es invocado para cada intento de acceso a una columna de una tabla en la base de datos. El callback deberá retornar `SQLITE_OK` si el acceso esta permitido, `SQLITE_DENY` si la completa declaración SQL deberá ser abortada con un error y `SQLITE_IGNORE` si la columna deberá ser tratada como un valor NULL. Estas constantes están disponibles en el módulo `sqlite3`.

El primer argumento del callback significa que tipo de operación será autorizada. El segundo y tercer argumento serán argumentos o `None` dependiendo del primer argumento. El cuarto argumento es el nombre de la base de datos («main», «temp», etc.) si aplica. El quinto argumento es el nombre del disparador más interno o vista que es responsable por los intentos de acceso o `None` si este intento de acceso es directo desde el código SQL de entrada.

Por favor consulte la documentación de SQLite sobre los posibles valores para el primer argumento y el significado del segundo y tercer argumento dependiendo del primero. Todas las constantes necesarias están disponibles en el módulo `sqlite3`.

set_progress_handler (*handler*, *n*)

Esta rutina registra un *callback*. El *callback* es invocado para cada *n* instrucciones de la máquina virtual

SQLite. Esto es útil si se quiere tener llamado a SQLite durante operaciones de larga duración, por ejemplo para actualizar una GUI.

Si se desea limpiar cualquier *progress handler* instalado previamente, llame el método con `None` para *handler*.

Retornando un valor diferente a 0 de la función gestora terminará la actual consulta en ejecución y causará lanzar una excepción `OperationalError`.

set_trace_callback (*trace_callback*)

Registra *trace_callback* para ser llamado por cada sentencia SQL que realmente se ejecute por el *backend* de SQLite.

El único argumento pasado al callback es la sentencia (como cadena de texto) que se está ejecutando. El valor retornado del *callback* es ignorado. Nótese que el backend no solo ejecuta la sentencia pasada a los métodos `Cursor.execute()`. Otras fuentes incluyen el gestión de la transacción del módulo de Python y la ejecución de los disparadores definidos en la actual base de datos.

Pasando `None` como *trace_callback* deshabilitara el *trace callback*.

Nuevo en la versión 3.3.

enable_load_extension (*enabled*)

Esta rutina habilita/deshabilita el motor de SQLite para cargar extensiones SQLite desde bibliotecas compartidas. Las extensiones SQLite pueden definir nuevas funciones, agregaciones o una completa nueva implementación de tablas virtuales. Una bien conocida extensión es *fulltext-search* distribuida con SQLite.

Las extensiones cargables están deshabilitadas por defecto. Ver¹.

Nuevo en la versión 3.2.

```
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');
    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'"):
```

(continué en la próxima página)

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 pass `--enable-loadable-sqlite-extensions` to configure.

(proviene de la página anterior)

```
print(row)

con.close()
```

load_extension (*path*)

Esta rutina carga una extensión SQLite de una biblioteca compartida. Se debe habilitar la carga de extensiones con `enable_load_extension()` antes de usar esta rutina.

Las extensiones cargables están deshabilitadas por defecto. Ver¹.

Nuevo en la versión 3.2.

row_factory

Se puede cambiar este atributo a un invocable que acepta el cursor y la fila original como una tupla y retornará la fila con el resultado real. De esta forma, se puede implementar más avanzadas formas de retornar resultados, tales como retornar un objeto que puede también acceder a las columnas por su nombre.

Ejemplo:

```
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"])

con.close()
```

Si retornado una tupla no es suficiente y se quiere acceder a las columnas basadas en nombre, se debe considerar configurar `row_factory` a la altamente optimizada tipo `sqlite3.Row`. `Row` provee ambos accesos a columnas basada en índice y tipado insensible con casi nada de sobrecoste de memoria. Será probablemente mejor que tú propio enfoque de basado en diccionario personalizado o incluso mejor que una solución basada en `db_row`.

text_factory

Usando este atributo se puede controlar que objetos son retornados por el tipo de dato TEXT. Por defecto, este atributo es configurado a `str` y el módulo `sqlite3` retornará objetos Unicode para TEXT. Si en cambio se quiere retornar `bytestrings`, se debe configurar a `bytes`.

También se puede configurar a cualquier otro *callable* que acepte un único parámetro *bytestring* y retorne el objeto resultante.

Ver el siguiente ejemplo de código para ilustración:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()

AUSTRIA = "\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
```

(continúe en la próxima página)

(proviene de la página anterior)

```

assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings ...
con.text_factory = bytes
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is bytes
# 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 appends "foo" to all strings
con.text_factory = lambda x: x.decode("utf-8") + "foo"
cur.execute("select ?", ("bar",))
row = cur.fetchone()
assert row[0] == "barfoo"

con.close()

```

total_changes

Regresa el número total de filas de la base de datos que han sido modificadas, insertadas o borradas desde que la conexión a la base de datos fue abierta.

iterdump()

Regresa un iterador para volcar la base de datos en un texto de formato SQL. Es útil cuando guardamos una base de datos en memoria para posterior restauración. Esta función provee las mismas capacidades que el comando `dump` en el *shell* **sqlite3**.

Ejemplo:

```

# Convert file existing_db.db to SQL dump file dump.sql
import sqlite3

con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
con.close()

```

backup(target, *, pages=-1, progress=None, name="main", sleep=0.250)

Este método crea un respaldo de una base de datos SQLite incluso mientras está siendo accedida por otros clientes, o concurrente por la misma conexión. La copia será escrita dentro del argumento obligatorio *target*, que deberá ser otra instancia de *Connection*.

Por defecto, o cuando *pages* es 0 o un entero negativo, la base de datos completa es copiada en un solo paso; de otra forma el método realiza un bucle copiando hasta el número de *pages* a la vez.

Si *progress* es especificado, deberá ser `None` o un objeto *callable* que será ejecutado en cada iteración con los tres argumentos enteros, respectivamente el estado *status* de la última iteración, el restante *remaining* numero de páginas presentes para ser copiadas y el número total *total* de páginas.

El argumento *name* especifica el nombre de la base de datos que será copiada; deberá ser una cadena de texto que contenga el por defecto `"main"`, que indica la base de datos principal, `"temp"` que indica la base de datos temporal o el nombre especificado después de la palabra clave `AS` en una sentencia `ATTACH DATABASE` para una base de datos adjunta.

El argumento *sleep* especifica el número de segundos a dormir entre sucesivos intentos de respaldar páginas restantes, puede ser especificado como un entero o un valor de punto flotante.

Ejemplo 1, copiar una base de datos existente en otra:

```
import sqlite3

def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total} pages...')

con = sqlite3.connect('existing_db.db')
bck = sqlite3.connect('backup.db')
with bck:
    con.backup(bck, pages=1, progress=progress)
bck.close()
con.close()
```

Ejemplo 2: copiar una base de datos existente en una copia transitoria:

```
import sqlite3

source = sqlite3.connect('existing_db.db')
dest = sqlite3.connect(':memory:')
source.backup(dest)
```

Disponibilidad: SQLite 3.6.11 o superior

Nuevo en la versión 3.7.

12.6.3 Objetos Cursor

class `sqlite3.Cursor`

Una instancia de *Cursor* tiene los siguientes atributos y métodos.

execute (*sql* [, *parameters*])

Ejecuta una sentencia SQL. La sentencia SQL puede estar parametrizada (es decir marcadores en lugar de literales SQL). El módulo *sqlite3* soporta dos tipos de marcadores: signos de interrogación (estilo qmark) y marcadores nombrados (estilo nombrado).

Acá esta un ejemplo con los dos estilos:

```
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())

con.close()
```

execute() solo ejecutará una única sentencia SQL. Si se trata de ejecutar más de una sentencia con el, lanzará un *Warning*. Usar *executescript()* si se quiere ejecutar múltiples sentencias SQL con una llamada.

executemany (*sql*, *seq_of_parameters*)

Ejecuta un comando SQL contra todas las secuencias de parámetros o mapeos encontrados en la secuen-

cia *seq_of_parameters*. El módulo *sqlite3* también permite usar un *iterator* produciendo parámetros en lugar de una secuencia.

```
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)

cur.execute("select c from characters")
print(cur.fetchall())

con.close()
```

Acá un corto ejemplo usando un *generator*:

```
import sqlite3
import string

def char_generator():
    for c in string.ascii_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())

con.close()
```

executescript (*sql_script*)

Este es un conveniente método no estándar para ejecutar múltiples sentencias SQL de una vez. Emite una sentencia COMMIT primero, luego ejecuta el script SQL obtenido como parámetro.

sql_script puede ser una instancia de *str*.

Ejemplo:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
```

(continué en la próxima página)

(proviene de la página anterior)

```
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
);
"""
con.close()
```

fetchone()

Obtiene la siguiente fila de un conjunto resultado, retorna una única secuencia, o *None* cuando no hay más datos disponibles.

fetchmany() (*size=cursor.arraysize*)

Obtiene el siguiente conjunto de filas del resultado de una consulta, retornando una lista. Una lista vacía es retornada cuando no hay más filas disponibles.

El número de filas a obtener por llamado es especificado por el parámetro *size*. Si no es suministrado, el *arraysize* del cursor determina el número de filas a obtener. El método debería intentar obtener tantas filas como las indicadas por el parámetro *size*. Si esto no es posible debido a que el número especificado de filas no está disponible, entonces menos filas deberán ser retornadas.

Nótese que hay consideraciones de desempeño involucradas con el parámetro *size*. Para un optimo desempeño, es usualmente mejor usar el atributo *arraysize*. Si el parámetro *size* es usado, entonces es mejor retener el mismo valor de una llamada *fetchmany()* a la siguiente.

fetchall()

Obtiene todas las filas (restantes) del resultado de una consulta. Nótese que el atributo *arraysize* del cursor puede afectar el desempeño de esta operación. Una lista vacía será retornada cuando no hay filas disponibles.

close()

Cierra el cursor ahora (en lugar que cuando `__del__` es llamado)

El cursor no será usable de este punto en adelante; una excepción *ProgrammingError* será lanzada si se intenta cualquier operación con el cursor.

rowcount

A pesar de que la clase *Cursor* del módulo *sqlite3* implementa este atributo, el propio soporte del motor de base de datos para la determinación de «filas afectadas»/«filas seleccionadas» es raro.

Para sentencias *executemany()*, el número de modificaciones se resumen en *rowcount*.

Cómo lo requiere la especificación Python DB API, el atributo *rowcount* «es -1 en caso de que *executeXX()* no haya sido ejecutada en el cursor o en el *rowcount* de la última operación no haya sido determinada por la interface». Esto incluye sentencias *SELECT* porque no podemos determinar el número de filas que una consulta produce hasta que todas las filas sean obtenidas.

Con versiones de SQLite anteriores a 3.6.5, *rowcount* es configurado a 0 si se hace un *DELETE FROM table* sin ninguna condición.

lastrowid

Este atributo de solo lectura provee el rowid de la última fila modificada. Solo se configura si se emite una sentencia INSERT o REPLACE usando el método `execute()`. Para otras operaciones diferentes a INSERT o REPLACE o cuando `executemany()` es llamado, `lastrowid` es configurado a `None`.

Si la sentencia INSERT o REPLACE no se pudo insertar, se retorna el anterior rowid exitoso.

Distinto en la versión 3.6: Se agregó soporte para sentencias REPLACE.

arraysize

Atributo de lectura/escritura que controla el número de filas retornadas por `fetchmany()`. El valor por defecto es 1, lo cual significa que una única fila será obtenida por llamada.

description

Este atributo de solo lectura provee el nombre de las columnas de la última consulta. Para ser compatible con Python DB API, retorna una 7-tupla para cada columna en donde los últimos seis ítems de cada tupla son `None`.

También es configurado para sentencias SELECT sin ninguna fila coincidente.

connection

Este atributo de solo lectura provee la `Connection` de la base de datos SQLite usada por el objeto `Cursor`. Un objeto `Cursor` creado por la llamada de `con.cursor()` tendrá un atributo `connection` que se refiere a `con`:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```

12.6.4 Objetos Fila (Row)

class `sqlite3.Row`

Una instancia `Row` sirve como una altamente optimizada `row_factory` para objetos `Connection`. Esta trata de imitar una tupla en su mayoría de características.

Soporta acceso mapeado por nombre de columna e índice, iteración, representación, pruebas de igualdad y `len()`.

Si dos objetos `Row` tienen exactamente las mismas columnas y sus miembros son iguales, entonces se comparan a igual.

keys()

Este método retorna una lista con los nombre de columnas. Inmediatamente después de una consulta, es el primer miembro de cada tupla en `Cursor.description`.

Distinto en la versión 3.5: Agrega soporte de segmentación.

Vamos a asumir que se inicializa una tabla como en el ejemplo dado:

```
con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute('''create table stocks
(date text, trans text, symbol text,
qty real, price real)''')
cur.execute("""insert into stocks
            values ('2006-01-05', 'BUY', 'RHAT', 100, 35.14) """)
con.commit()
cur.close()
```

Ahora conectamos `Row` en:

```
>>> con.row_factory = sqlite3.Row
>>> cur = con.cursor()
>>> cur.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = cur.fetchone()
>>> type(r)
<class 'sqlite3.Row'>
>>> tuple(r)
('2006-01-05', 'BUY', 'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
'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
```

12.6.5 Excepciones

exception `sqlite3.Warning`

Una subclase de *Exception*.

exception `sqlite3.Error`

La clase base de otras excepciones en este módulo. Es una subclase de *Exception*.

exception `sqlite3.DatabaseError`

Excepción lanzada para errores que están relacionados con la base de datos.

exception `sqlite3.IntegrityError`

Excepción lanzada cuando la integridad de la base de datos es afectada, por ejemplo la comprobación de una llave foránea falla. Es una subclase de *DatabaseError*.

exception `sqlite3.ProgrammingError`

Excepción lanzada por errores de programación, e.g. tabla no encontrada o ya existente, error de sintaxis en la sentencia SQL, número equivocado de parámetros especificados, etc. Es una subclase de *DatabaseError*.

exception `sqlite3.OperationalError`

Excepción lanzada por errores relacionados por la operación de la base de datos y no necesariamente bajo el control del programador, por ejemplo ocurre una desconexión inesperada, el nombre de la fuente de datos no es encontrado, una transacción no pudo ser procesada, etc. Es una subclase de *DatabaseError*.

exception `sqlite3.NotSupportedError`

Excepción lanzada en caso de que un método o API de base de datos fuera usada en una base de datos que no la soporta, e.g. llamando el método *rollback()* en una conexión que no soporta la transacción o tiene deshabilitada las transacciones. Es una subclase de *DatabaseError*.

12.6.6 SQLite y tipos de Python

Introducción

SQLite soporta de forma nativa los siguientes tipos: NULL, INTEGER, REAL, TEXT, BLOB.

Los siguientes tipos de Python se pueden enviar a SQLite sin problema alguno:

Tipo de Python	Tipo de SQLite
<i>None</i>	NULL
<i>int</i>	INTEGER
<i>float</i>	REAL
<i>str</i>	TEXT
<i>bytes</i>	BLOB

De esta forma es como los tipos de SQLite son convertidos a tipos de Python por defecto:

Tipo de SQLite	Tipo de Python
NULL	<i>None</i>
INTEGER	<i>int</i>
REAL	<i>float</i>
TEXT	depende de <i>text_factory</i> , <i>str</i> por defecto
BLOB	<i>bytes</i>

El sistema de tipos del módulo *sqlite3* es extensible en dos formas: se puede almacenar tipos de Python adicionales en una base de datos SQLite vía adaptación de objetos, y se puede permitir que el módulo *sqlite3* convierta tipos SQLite a diferentes tipos de Python vía convertidores.

Usando adaptadores para almacenar tipos adicionales de Python en bases de datos SQLite

Como se describió anteriormente, SQLite soporta solamente un conjunto limitado de tipos de forma nativa. Para usar otros tipos de Python con SQLite, se deben **adaptar** a uno de los tipos de datos soportados por el módulo *sqlite3* para SQLite: uno de NoneType, int, float, str, bytes.

Hay dos formas de habilitar el módulo *sqlite3* para adaptar un tipo personalizado de Python a alguno de los admitidos.

Permitiéndole al objeto auto adaptarse

Este es un buen enfoque si uno mismo escribe la clase. Vamos a suponer que se tiene una clase como esta:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

Ahora se quiere almacenar el punto en una columna SQLite. Primero se debe elegir un tipo de los soportados para representar el punto. Se va a usar cadena de texto separando las coordenadas usando un punto y coma. Luego se necesita proveer a la clase el método `__conform__(self, protocol)` el cual deberá retornar el valor convertido. El parámetro *protocol* será `PrepareProtocol`.

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

(continué en la próxima página)

(proviene de la página anterior)

```
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])

con.close()
```

Registrando un adaptador invocable

La otra posibilidad es crear una función que convierta el escrito a representación de cadena de texto y registrar la función con `register_adapter()`.

```
import sqlite3

class Point:
    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])

con.close()
```

El módulo `sqlite3` tiene dos adaptadores por defecto para las funciones integradas de Python `datetime.date` y tipos `datetime.datetime`. Ahora vamos a suponer que queremos almacenar objetos `datetime.datetime` no en representación ISO, sino como una marca de tiempo Unix.

```
import sqlite3
import datetime
import 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()
cur.execute("select ?", (now,))
print(cur.fetchone()[0])

con.close()
```

Convertir valores SQLite a tipos de Python personalizados

Escribir un adaptador permite enviar escritos personalizados de Python a SQLite. Pero para hacer esto realmente útil, tenemos que hacer el flujo Python a SQLite a Python.

Ingresa convertidores.

Regresemos a la clase `Point`. Se almacena las coordenadas `x` y `y` de forma separada por punto y coma como una cadena de texto en SQLite.

Primero, se define una función de conversión que acepta la cadena de texto como un parámetro y construye un objeto `Point` de ahí.

Nota: Las funciones de conversión **siempre** son llamadas con un objeto `bytes`, no importa bajo qué tipo de dato se envió el valor a SQLite.

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

Ahora se necesita hacer que el módulo `sqlite3` conozca que lo que tu seleccionaste de la base de datos es de hecho un punto. Hay dos formas de hacer esto:

- Implícitamente vía el tipo declarado
- Explícitamente vía el nombre de la columna

Ambas formas están descritas en la sección *Funciones y constantes del módulo*, en las entradas para las constantes `PARSE_DECLTYPES` y `PARSE_COLNAMES`.

El siguiente ejemplo ilustra ambos enfoques.

```
import sqlite3

class Point:
    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).encode('ascii')

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
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)")
```

(continué en la próxima página)

(proviene de la página anterior)

```

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()

```

Adaptadores y convertidores por defecto

Hay adaptadores por defecto para los tipos `date` y `datetime` en el módulo `datetime`. Éstos serán enviados como fechas/marcas de tiempo ISO a SQLite.

Los convertidores por defecto están registrados bajo el nombre «date» para `datetime.date` y bajo el mismo nombre para «timestamp» para `datetime.datetime`.

De esta forma, se puede usar `date/timestamps` para Python sin ajuste adicional en la mayoría de los casos. El formato de los adaptadores también es compatible con las funciones experimentales de SQLite `date/time`.

El siguiente ejemplo demuestra esto.

```

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]))

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]))

con.close()

```

Si un *timestamp* almacenado en SQLite tiene una parte fraccional mayor a 6 números, este valor será truncado a precisión de microsegundos por el convertidor de *timestamp*.

12.6.7 Controlando Transacciones

La librería subyacente `sqlite3` opera en modo `autocommit` por defecto, pero el módulo de Python `sqlite3` no.

El modo `autocommit` significa que la sentencias que modifican la base de datos toman efecto de forma inmediata. Una sentencia `BEGIN` o `SAVEPOINT` deshabilitan el modo `autocommit`, y un `COMMIT`, un `ROLLBACK`, o un `RELEASE` que terminan la transacción más externa, habilitan de nuevo el modo `autocommit`.

El módulo de Python `sqlite3` emite por defecto una sentencia `BEGIN` implícita antes de una sentencia tipo Lenguaje Manipulación de Datos (DML) (es decir `INSERT/UPDATE/DELETE/REPLACE`).

Se puede controlar en qué tipo de sentencias `BEGIN` `sqlite3` implícitamente ejecuta vía el parámetro `isolation_level` a la función de llamada `connect()`, o vía las propiedades de conexión `isolation_level`. Si no se especifica `isolation_level`, se usa un plano `BEGIN`, el cuál es equivalente a especificar `DEFERRED`. Otros posibles valores son `IMMEDIATE` and `EXCLUSIVE`.

Se puede deshabilitar la gestión implícita de transacciones del módulo `sqlite3` con la configuración `isolation_level` a `None`. Esto dejará la subyacente biblioteca operando en modo `autocommit`. Se puede controlar completamente el estado de la transacción emitiendo explícitamente sentencias `BEGIN`, `ROLLBACK`, `SAVEPOINT`, y `RELEASE` en el código.

Distinto en la versión 3.6: `sqlite3` solía realizar `commit` en transacciones implícitamente antes de sentencias `DDL`. Este ya no es el caso.

12.6.8 Usando `sqlite3` eficientemente

Usando métodos atajo

Usando los métodos no estándar `execute()`, `executemany()` y `executescript()` del objeto `Connection`, el código puede ser escrito más consistentemente porque no se tienen que crear explícitamente los (a menudo superfluos) objetos `Cursor`. En cambio, los objetos de `Cursor` son creados implícitamente y estos métodos atajo retornan los objetos `cursor`. De esta forma, se puede ejecutar una sentencia `SELECT` e iterar directamente sobre él, solamente usando una única llamada al objeto `Connection`.

```
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
for row in con.execute("select firstname, lastname from person"):
    print(row)

print("I just deleted", con.execute("delete from person").rowcount, "rows")

# close is not a shortcut method and it's not called automatically,
# so the connection object should be closed manually
con.close()
```

Accediendo a las columnas por el nombre en lugar del índice

Una característica útil del módulo `sqlite3` es la clase incluida `sqlite3.Row` diseñada para ser usada como una fábrica de filas.

Filas envueltas con esta clase pueden ser accedidas tanto por índice (al igual que tuplas) como por nombre insensible a mayúsculas o minúsculas:

```
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"]

con.close()
```

Usando la conexión como un administrador de contexto

Los objetos de conexión pueden ser usados como administradores de contexto que automáticamente transacciones commit o rollback. En el evento de una excepción, la transacción es retrocedida; de otra forma, la transacción es confirmada:

```
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")

# Connection object used as context manager only commits or rollbacks transactions,
# so the connection object should be closed manually
con.close()
```


Notas al pie

Compresión de datos y archivado

Los módulos descritos en este capítulo soportan compresión de datos con los algoritmos zlib, gzip, bzip2 e lzma, y la creación de archivos con formato ZIP y tar. Véase también *Operaciones de archivado* provistos por el módulo *shutil*.

13.1 zlib — Compresión compatible con gzip

Para las aplicaciones que requieren compresión de datos, las funciones de este módulo permiten la compresión y la descompresión, utilizando la biblioteca zlib. La biblioteca zlib tiene su propia página de inicio en <http://www.zlib.net>. Existen incompatibilidades conocidas entre el módulo Python y las versiones de la biblioteca zlib anteriores a 1.1.3; 1.1.3 tiene una vulnerabilidad de seguridad, por lo que recomendamos usar 1.1.4 o posterior.

Las funciones de zlib tienen muchas opciones y a menudo necesitan ser utilizadas en un orden particular. Esta documentación no intenta cubrir todas las permutaciones; consultar el manual de zlib en <http://www.zlib.net/manual.html> para obtener información autorizada.

Para leer y escribir archivos .gz consultar el módulo *gzip*.

La excepción y las funciones disponibles en este módulo son:

exception `zlib.error`

Excepción provocada en errores de compresión y descompresión.

zlib.adler32 (*data* [, *value*])

Calcula una suma de comprobación Adler-32 de *data*. (Una suma de comprobación Adler-32 es casi tan confiable como un CRC32, pero se puede calcular mucho más rápidamente). El resultado es un entero de 32 bits sin signo. Si *value* está presente, se utiliza como el valor inicial de la suma de comprobación; de lo contrario, se utiliza un valor predeterminado de 1. Pasar *value* permite calcular una suma de comprobación en ejecución durante la concatenación de varias entradas. El algoritmo no es criptográficamente fuerte y no se debe utilizar para la autenticación o las firmas digitales. Puesto que está diseñado como un algoritmo de suma de comprobación, no es adecuado su uso como un algoritmo *hash* general.

Distinto en la versión 3.0: Siempre retorna un valor sin signo. Para generar el mismo valor numérico en todas las versiones y plataformas de Python, utilice `adler32(data) & 0xffffffff`.

zlib.compress (*data*, *level*=-1)

Comprime los bytes de *data*, retornando un objeto bytes que contiene datos comprimidos. *level* es

un entero de 0 a 9 o -1 que controla el nivel de compresión; 1 (Z_BEST_SPEED) es más rápido y produce la menor compresión, 9 (Z_BEST_COMPRESSION) es más lento y produce mayor compresión. 0 (Z_NO_COMPRESSION) no es compresión. El valor predeterminado es -1 (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION representa un compromiso predeterminado entre velocidad y compresión (actualmente equivalente al nivel 6). Genera la excepción `error` si se produce algún error.

Distinto en la versión 3.6: *level* ahora se puede utilizar como parámetro de palabra clave.

```
zlib.compressobj (level=-1, method=DEFLATED, wbits=MAX_WBITS, memLevel=DEF_MEM_LEVEL, strategy=Z_DEFAULT_STRATEGY[, zdict ])
```

Retorna un objeto de compresión, para ser usado para comprimir flujos de datos que no caben en la memoria de una vez.

level es el nivel de compresión – es un entero de “0” a “9” o “-1”. Un valor de “1” (Z_BEST_SPEED) es el más rápido y produce la menor compresión, mientras que un valor de “9” (Z_BEST_COMPRESSION) es el más lento y produce la mayor cantidad. 0 (Z_NO_COMPRESSION) no es compresión. El valor predeterminado es -1 (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION representa un compromiso predeterminado entre velocidad y compresión (actualmente equivalente al nivel 6).

method es el algoritmo de compresión. Actualmente, el único valor admitido es DEFLATED.

El argumento *wbits* controla el tamaño del búfer histórico (o el «tamaño de ventana») utilizado al comprimir datos, y si se incluye un encabezado y un avance en la salida. Puede tomar varios rangos de valores, por defecto es 15 (MAX_WBITS):

- +9 a +15: el logaritmo en base dos del tamaño de la ventana, que por lo tanto oscila entre 512 y 32768. Los valores más grandes producen una mejor compresión a expensas de un mayor uso de memoria. Como resultado se incluirá un encabezado y un avance específicos de zlib.
- -9 a -15: utiliza el valor absoluto de *wbits* como el logaritmo del tamaño de la ventana, al tiempo que produce una secuencia de salida sin encabezado ni suma de verificación.
- +25 a +31 = 16 + (9 a 15): utiliza los 4 bits bajos del valor como el logaritmo del tamaño de la ventana, mientras que incluye un encabezado básico **gzip** y suma de verificación en la salida.

El argumento *memLevel* controla la cantidad de memoria utilizada para el estado de compresión interna. Los valores posibles varían de 1 a 9. Los valores más altos usan más memoria, pero son más rápidos y producen una salida más pequeña.

strategy se usa para ajustar el algoritmo de compresión. Los valores posibles son Z_DEFAULT_STRATEGY, Z_FILTERED, Z_HUFFMAN_ONLY, Z_RLE (zlib 1.2.0.1) y Z_FIXED (zlib 1.2.2.2).

zdict es un diccionario de compresión predefinido. Este es una secuencia de bytes (como un objeto *bytes*) que contiene subsecuencias que se espera que ocurran con frecuencia en los datos que se van a comprimir. Las subsecuencias que se espera que sean más comunes deben aparecer al final del diccionario.

Distinto en la versión 3.3: Se agregó el parámetro *zdict* y el soporte de argumentos de palabras clave.

```
zlib.crc32 (data[, value ])
```

Calcula un suma de comprobación CRC (comprobación de redundancia cíclica, por sus siglas en inglés *Cyclic Redundancy Check*) de *datos*. El resultado es un entero de 32 bits sin signo. Si *value* está presente, se usa como el valor inicial de la suma de verificación; de lo contrario, se usa el valor predeterminado 0. Pasar *value* permite calcular una suma de verificación en ejecución sobre la concatenación de varias entradas. El algoritmo no es criptográficamente fuerte y no debe usarse para autenticación o firmas digitales. Dado que está diseñado para usarse como un algoritmo de suma de verificación, no es adecuado su uso como algoritmo *hash* general.

Distinto en la versión 3.0: Siempre retorna un valor sin signo. Para generar el mismo valor numérico en todas las versiones y plataformas de Python, use `crc32 (data) & 0xffffffff`.

```
zlib.decompress (data, wbits=MAX_WBITS, bufsize=DEF_BUF_SIZE)
```

Descomprime los bytes en *data*, retornando un objeto de bytes que contiene los datos sin comprimir. El parámetro *wbits* depende del formato de *data*, y se trata más adelante. Si se da *bufsize*, se usa como el tamaño inicial del búfer de salida. Provoca la excepción `error` si se produce algún error.

El parámetro *wbits* controla el tamaño del búfer histórico (o el «tamaño de ventana») y qué formato de encabezado y cola se espera. Es similar al parámetro para `compressobj()`, pero acepta más rangos de valores:

- +8 a +15: el logaritmo en base dos del tamaño del búfer. La entrada debe incluir encabezado y cola zlib.
- 0: determina automáticamente el tamaño del búfer desde el encabezado zlib. Solo se admite desde zlib 1.2.3.5.
- -8 to -15: utiliza el valor absoluto de *wbits* como el logaritmo del tamaño del búfer. La entrada debe ser una transmisión sin formato, sin encabezado ni cola.
- +24 a +31 = 16 + (8 a 15): utiliza los 4 bits de bajo orden del valor como el logaritmo del tamaño del búfer. La entrada debe incluir encabezado y cola gzip.
- +40 a +47 = 32 + (8 a 15): utiliza los 4 bits de bajo orden del valor como el logaritmo del tamaño del búfer y acepta automáticamente el formato zlib o gzip.

Al descomprimir una secuencia, el tamaño del búfer no debe ser menor al tamaño utilizado originalmente para comprimir la secuencia; el uso de un valor demasiado pequeño puede generar una excepción `error`. El valor predeterminado *wbits* corresponde al tamaño más grande de búfer y requiere que se incluya encabezado y cola zlib.

bufsize es el tamaño inicial del búfer utilizado para contener datos descomprimidos. Si se requiere más espacio, el tamaño del búfer aumentará según sea necesario, por lo que no hay que tomar este valor como exactamente correcto; al ajustarlo solo se guardarán algunas llamadas en `malloc()`.

Distinto en la versión 3.6: *wbits* y *bufsize* se pueden usar como argumentos de palabra clave.

`zlib.decompressobj(wbits=MAX_WBITS[, zdict])`

Retorna un objeto de descompresión, que se utilizará para descomprimir flujos de datos que no caben en la memoria de una vez.

El parámetro *wbits* controla el tamaño del búfer histórico (o el «tamaño de ventana») y qué formato de encabezado y cola se espera. Tiene el mismo significado que *described for decompress()*.

El parámetro *zdict* especifica un diccionario de compresión predefinido. Si se proporciona, este debe ser el mismo diccionario utilizado por el compresor que produjo los datos que se van a descomprimir.

Nota: Si *zdict* es un objeto mutable (como `bytearray`), no se debe modificar su contenido entre la llamada `decompressobj()` y la primera llamada al método descompresor `decompress()`.

Distinto en la versión 3.3: Se agregó el parámetro *zdict*.

Los objetos de compresión admiten los siguientes métodos:

`Compress.compress(data)`

Comprime *data*, y retorna al menos algunos de los datos comprimidos como un objeto de bytes. Estos datos deben concatenarse a la salida producida por cualquier llamada anterior al método `compress()`. Algunas entradas pueden mantenerse en un búfer interno para su posterior procesamiento.

`Compress.flush([mode])`

Se procesan todas las entradas pendientes y se retorna un objeto de bytes que contiene la salida comprimida restante. El argumento *mode* acepta una de las siguientes constantes `Z_NO_FLUSH`, `Z_PARTIAL_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH`, `Z_BLOCK` (zlib 1.2.3.4), o `Z_FINISH`, por defecto `Z_FINISH`. Excepto `Z_FINISH`, todas las constantes permiten comprimir más cadenas de bytes, mientras que `Z_FINISH` finaliza el flujo comprimido y evita la compresión de más datos. Después de llamar a `flush()` con *mode* establecido en `Z_FINISH`, el método `compress()` no se puede volver a llamar. La única acción posible es eliminar el objeto.

`Compress.copy()`

Retorna una copia del objeto compresor. Esto se puede utilizar para comprimir eficientemente un conjunto de datos que comparten un prefijo inicial común.

Distinto en la versión 3.8: Añadido `copy.copy()` y `copy.deepcopy()` para el soporte de objetos de compresión.

Los objetos de descompresión admiten los siguientes métodos y atributos:

Decompress.unused_data

Un objeto de bytes que contiene todos los bytes restantes después de los datos comprimidos. Es decir, esto permanece b "" hasta que esté disponible el último byte que contiene datos de compresión. Si la cadena de bytes completa resultó contener datos comprimidos, es b "", un objeto de bytes vacío.

Decompress.unconsumed_tail

Un objeto de bytes que contiene datos no procesados por la última llamada al método `descompress()`, debido a un desbordamiento del límite del búfer de datos descomprimidos. Estos datos aún no han sido procesados por la biblioteca `zlib`, por lo que debe enviarlos (potencialmente concatenando los datos nuevamente) mediante una llamada al método `descompress()` para obtener la salida correcta.

Decompress.eof

Un valor booleano que indica si se ha alcanzado el final del flujo de datos comprimido.

Esto hace posible distinguir entre un flujo comprimido correctamente y un flujo incompleto.

Nuevo en la versión 3.3.

Decompress.decompress(data, max_length=0)

Descomprime *data*, retornando un objeto de bytes que contiene al menos parte de los datos descomprimidos en *string*. Estos datos deben concatenarse con la salida producida por cualquier llamada anterior al método `decompress()`. Algunos de los datos de entrada pueden conservarse en búfer internos para su posterior procesamiento.

Si el parámetro opcional *max_length* no es cero, el valor de retorno no será más largo que *max_length*. Esto puede significar que no toda la entrada comprimida puede procesarse; y los datos no consumidos se almacenarán en el atributo `unconsumed_tail`. Esta cadena de bytes debe pasarse a una llamada posterior a `decompress()` si la descompresión ha de continuar. Si *max_length* es cero, toda la entrada se descomprime y `unconsumed_tail` queda vacío.

Distinto en la versión 3.6: *max_length* puede ser utilizado como argumento de palabras clave.

Decompress.flush([length])

Se procesan todas las entradas pendientes y se retorna un objeto de bytes que contiene el resto de los datos que se descomprimirán. Después de llamar a `flush()`, no se puede volver a llamar al método `decompress()`. La única acción posible es eliminar el objeto.

El parámetro opcional *length* establece el tamaño inicial del búfer de salida.

Decompress.copy()

Retorna una copia del objeto de descompresión. Puede usarlo para guardar el estado de la descompresión actual, de modo que pueda regresar rápidamente a esta ubicación más tarde.

Distinto en la versión 3.8: Añadido `copy.copy()` y `copy.deepcopy()` para el soporte de objetos de compresión.

La información sobre la versión de la biblioteca `zlib` en uso está disponible a través de las siguientes constantes:

zlib.ZLIB_VERSION

Versión de la biblioteca `zlib` utilizada al compilar el módulo. Puede ser diferente de la biblioteca `zlib` utilizada actualmente por el sistema, que puede ver en `ZLIB_RUNTIME_VERSION`.

zlib.ZLIB_RUNTIME_VERSION

Cadena que contiene la versión de la biblioteca `zlib` utilizada actualmente por el intérprete.

Nuevo en la versión 3.3.

Ver también:

Módulo `gzip` Lectura y escritura de los archivos en formato `gzip`.

<http://www.zlib.net> Página oficial de la biblioteca `zlib`.

<http://www.zlib.net/manual.html> El manual de `zlib` explica la semántica y el uso de las numerosas funciones de la biblioteca.

13.2 gzip — Soporte para archivos gzip

Código fuente: [Lib/gzip.py](#)

Este módulo proporciona una interfaz simple para comprimir y descomprimir archivos como lo harían los programas GNU **gzip** y **gunzip**.

La compresión de datos la proporciona el módulo *zlib*.

El módulo *gzip* proporciona la clase *GzipFile*, así como las funciones de conveniencia *open()*, *compress()* y *decompress()*. La clase *GzipFile* lee y escribe archivos de formato **gzip**, comprimiendo o descomprimiendo automáticamente los datos para que parezcan un *file object* ordinario.

Tenga en cuenta que los formatos de archivo adicionales que pueden ser descomprimidos por los programas **gzip** y **gunzip**, como los producidos por **compress** y **pack**, no son compatibles con este módulo.

El módulo define los siguientes elementos:

`gzip.open(filename, mode='rb', compresslevel=9, encoding=None, errors=None, newline=None)`
Abre un archivo comprimido gzip en modo binario o de texto, devolviendo un *file object*.

El argumento *filename* puede ser un nombre de archivo real (un objeto *str* o *bytes*), o un objeto archivo existente para leer o escribir.

El argumento *mode* puede ser cualquiera de 'r', 'rb', 'a', 'ab', 'w', 'wb', 'x' or 'xb' para el modo binario, o 'rt', 'at', 'wt', or 'xt' para el modo texto. El valor predeterminado es 'rb'.

El argumento *compresslevel* es un número entero de 0 a 9, como para el constructor *GzipFile*.

Para el modo binario, esta función es equivalente al constructor *GzipFile*: *GzipFile(filename, mode, compresslevel)*. En este caso, no se deben proporcionar los argumentos *encoding*, *errors* y *newline*.

Para el modo texto, se crea un objeto *GzipFile* y se envuelve en una instancia *io.TextIOWrapper* con la codificación especificada, comportamiento de manejo de errores y terminación(es) de línea.

Distinto en la versión 3.3: Se agregó soporte para que *filename* sea un objeto de archivo, soporte para el modo de texto y los argumentos *encoding*, *errors* y *newline*.

Distinto en la versión 3.4: Se agregó soporte para los modos 'x', 'xb' y 'xt'.

Distinto en la versión 3.6: Acepta un *path-like object*.

exception `gzip.BadGzipFile`

Se lanzó una excepción para archivos gzip no válidos. Hereda *OSError*. *OSError* y *zlib.error* también se pueden lanzar para archivos gzip no válidos.

Nuevo en la versión 3.8.

class `gzip.GzipFile(filename=None, mode=None, compresslevel=9, fileobj=None, mtime=None)`

Constructor para la clase *GzipFile*, que simula la mayoría de los métodos de *file object*, con la excepción del método *truncate()*. Al menos uno de *fileobj* y *filename* debe recibir un valor no trivial.

La nueva instancia de clase se basa en *fileobj*, que puede ser un archivo normal, un objeto *io.BytesIO*, o cualquier otro objeto que simule un archivo. El valor predeterminado es *None*, en cuyo caso se abre *filename* para proporcionar un objeto de archivo.

Cuando *fileobj* no es *None*, el argumento *filename* solo se usa para ser incluido en el encabezado del archivo **gzip**, que puede incluir el nombre de archivo original del archivo sin comprimir. De forma predeterminada, el nombre de archivo es de *fileobj*, si es discernible; de lo contrario, el valor predeterminado es la cadena vacía, y en este caso el nombre del archivo original no se incluye en el encabezado.

El argumento *mode* puede ser cualquiera de 'r', 'rb', 'a', 'ab', 'w', 'wb', 'x', o 'xb', dependiendo de si el archivo se leerá o escribirá. El valor predeterminado es el modo de *fileobj* si se puede discernir; de lo contrario, el valor predeterminado es 'rb'.

Tenga en cuenta que el archivo siempre se abre en modo binario. Para abrir un archivo comprimido en modo de texto, utilice `open()` (o ajuste su `GzipFile` con un `io.TextIOWrapper`).

El argumento `compresslevel` es un entero de 0 a 9 que controla el nivel de compresión; 1 es más rápido y produce la menor compresión, y 9 es más lento y produce la mayor compresión. 0 no es una compresión. El valor predeterminado es 9.

El argumento `mtime` es una marca de tiempo numérica opcional que se escribirá en el campo de tiempo de última modificación de la secuencia al comprimir. Sólo debe proporcionarse en modo de compresión. Si se omite o `None`, se utiliza la hora actual. Consulte el atributo `mtime` para obtener más detalles.

Al llamar en un objeto `GzipFile` el método `close()` no cierra `fileobj`, ya que es posible que desee anexas más material después de los datos comprimidos. Esto también le permite pasar un objeto `io.BytesIO` que se abrió para escribir como `fileobj`, y recupera el búfer de memoria resultante usando el `io.BytesIO` método del objeto `getvalue()`.

`GzipFile` supports the `io.BufferedIOBase` interface, including iteration and the `with` statement. Only the `truncate()` method isn't implemented.

`GzipFile` también proporciona el siguiente método y atributo:

peek(*n*)

Lee *n* bytes no comprimidos sin avanzar en la posición del archivo. A lo sumo se realiza una sola lectura en la secuencia comprimida para satisfacer la llamada. El número de bytes retornados puede ser mayor o menor de lo solicitado.

Nota: Al llamar a `peek()` no cambia la posición del archivo `GzipFile`, puede cambiar la posición del objeto de archivo subyacente (por ejemplo, si el `GzipFile` se construyó con el parámetro `fileobj`).

Nuevo en la versión 3.2.

mtime

Al descomprimir, el valor de la última modificación del campo de tiempo en el encabezado de lectura más reciente se puede leer de este atributo, como un entero. El valor inicial antes de leer cualquier encabezado es `None`.

Todas las secuencias comprimidas **gzip** deben contener este campo de marca de tiempo. Algunos programas, como **gunzip**, hacen uso de la marca de tiempo. El formato es el mismo que el valor retornado de `time.time()` y el atributo `st_mtime` del objeto retornado por `os.stat()`.

Distinto en la versión 3.1: Se ha agregado soporte para la declaración `with`, junto con el argumento del constructor `mtime` y el atributo `mtime`.

Distinto en la versión 3.2: Se agregó soporte para archivos con relleno de ceros y que no se pueden buscar.

Distinto en la versión 3.3: El método `io.BufferedIOBase.read1()` ahora está implementado.

Distinto en la versión 3.4: Se agregó soporte para los modos `'x'` y `'xb'`.

Distinto en la versión 3.5: Se ha añadido soporte para escribir objetos arbitrarios *bytes-like objects*. El método `read()` ahora acepta un argumento de `None`.

Distinto en la versión 3.6: Acepta un *path-like object*.

`gzip.compress(data, compresslevel=9, *, mtime=None)`

Comprime los *data*, devolviendo un objeto *bytes* que contiene los datos comprimidos. `compresslevel` y `mtime` tienen el mismo significado que en el constructor `GzipFile` anterior.

Nuevo en la versión 3.2.

Distinto en la versión 3.8: Se agregó el parámetro `mtime` para una salida reproducible.

`gzip.decompress(data)`

Descomprime los *data*, devolviendo un objeto *bytes* que contiene los datos sin comprimir.

Nuevo en la versión 3.2.

13.2.1 Ejemplos de uso

Ejemplos de como leer un archivo comprimido:

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

Ejemplos de como crear un archivo comprimido GZIP:

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

Ejemplos de como GZIP comprime un archivo existente:

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

Ejemplo de como GZIP comprime una cadena binaria:

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

Ver también:

Módulo `zlib` El módulo básico de compresión de datos necesario para admitir el formato de archivo **gzip**.

13.2.2 Interfaz de Línea de Comandos

El módulo `gzip` proporciona una interfaz de línea de comandos simple para comprimir o descomprimir archivos.

Una vez ejecutado el módulo `gzip` conserva los archivos de entrada.

Distinto en la versión 3.8: Agrega una nueva interfaz de línea de comandos con un uso. De forma predeterminada, cuando ejecutará la CLI, el nivel de compresión predeterminado es 6.

Opciones de línea de comandos

file

Si no se especifica *file*, lee de `sys.stdin`.

--fast

Indica el método de compresión más rápido (menos compresión).

--best

Indica el método de compresión más lento (mejor compresión).

-d, --decompress

Descomprime el archivo dado.

-h, --help

Muestra el mensaje de ayuda.

13.3 bz2 — Soporte para compresión bzip2

Código fuente: [Lib/bz2.py](#)

Este módulo proporciona una interfaz completa para comprimir y descomprimir datos utilizando el algoritmo de compresión bzip2.

El módulo `bz2` contiene:

- La función `open()` y la clase `BZ2File` para leer y escribir archivos comprimidos.
- Las clases `BZ2Compressor` y `BZ2Decompressor` para la (de)compresión incremental.
- Las funciones `compress()` y `decompress()` para una (de)compresión en un solo paso.

Se puede acceder de forma segura a todas las clases de este módulo desde varios hilos.

13.3.1 (De)compresión de archivos

`bz2.open(filename, mode='r', compresslevel=9, encoding=None, errors=None, newline=None)`

Abre un archivo comprimido con bzip2 en modo binario o texto, retorna un *objeto archivo*.

Al igual que con el constructor para `BZ2File`, el argumento `filename` puede ser un nombre de archivo real (un objeto `str` o `bytes`), o un objeto de archivo existente para leer o escribirle.

El argumento `mode` puede ser cualquiera de `'r'`, `'rb'`, `'w'`, `'wb'`, `'x'`, `'xb'`, `'a'` o `'ab'` para modo binario, o `'rt'`, `'wt'`, `'xt'`, o `'a'` para el modo de texto. El valor predeterminado es `'rb'`.

El argumento `compresslevel` es un entero del 1 al 9, como para el constructor de `BZ2File`.

Para el modo binario, esta función es equivalente a `BZ2File` constructor: `BZ2File(filename, mode, compresslevel=compresslevel)`. En este caso, no se deben proporcionar los argumentos `encoding`, `errors` y `newline` (nueva línea).

Para el modo de texto, se crea un objeto `BZ2File`, y se envuelve en una instancia `io.TextIOWrapper` con la codificación especificada, el comportamiento de manejo de errores y los final(es) de línea.

Nuevo en la versión 3.3.

Distinto en la versión 3.4: El modo `'x'` (creación exclusiva) ha sido agregado.

Distinto en la versión 3.6: Acepta un objeto similar a una ruta (*path-like object*).

class `bz2.BZ2File(filename, mode='r', buffering=None, compresslevel=9)`

Abre un archivo comprimido con bzip2 en modo binario.

Si `filename` es un objeto tipo `str` o `bytes`, abre el archivo nombrado directamente. De lo contrario, `filename` debería ser un *file object*, que se utilizará para leer o escribir los datos comprimidos.

El argumento `mode` puede ser `'r'` para lectura (predeterminado), `'w'` para sobrescribir, `'x'` para creación exclusiva o `'a'` para anexas. Estos se pueden dar de manera equivalente como `'rb'`, `'wb'`, `'xb'` y `'ab'` respectivamente.

Si `filename` es un objeto de archivo (en lugar de un nombre de archivo real), el modo `'w'` no trunca el archivo, y es equivalente a `'a'`.

El argumento `buffering` se ignora. Su uso está en desuso desde Python 3.0.

Si `mode` es `'w'` o `'a'`, `compresslevel` puede ser un número entero entre 1 y 9 especificando el nivel de compresión: 1 produce la menor compresión y 9 (predeterminado) produce la mayor compresión.

Si `mode` es `'r'`, el archivo de entrada puede ser la concatenación de múltiples flujos (streams) comprimidos.

`BZ2File` proporciona todos los miembros especificados por `io.BufferedIOBase`, excepto `detach()` y `truncate()`. Se admite la iteración y la palabra clave `with`.

`BZ2File` también proporciona el siguiente método:

peek (*[n]*)

Retorna datos almacenados en el búfer sin avanzar la posición del archivo. Se retornará al menos un byte de datos (a menos que sea EOF). El número exacto de bytes retornados no está especificado.

Nota: Al invocar `peek()` no cambia la posición del archivo de `BZ2File`, puede cambiar la posición del objeto de archivo subyacente (por ejemplo, si `BZ2File` se construyó pasando un objeto de archivo a `filename`).

Nuevo en la versión 3.3.

Obsoleto desde la versión 3.0: El argumento de la palabra clave `buffering` quedó en desuso y ahora se ignora.

Distinto en la versión 3.1: Se agregó soporte para la declaración `with`.

Distinto en la versión 3.3: Se agregaron los métodos `fileno()`, `legible()`, `seekable()`, `writable()`, `read1()` y `readinto()`.

Distinto en la versión 3.3: Se agregó soporte para `filename` siendo un objeto de archivo (*file object*) en lugar de un nombre de archivo real.

Distinto en la versión 3.3: Se agregó el modo `'a'` (agregar), junto con el soporte para leer archivos de flujo múltiple (multi-stream).

Distinto en la versión 3.4: El modo `'x'` (creación exclusiva) ha sido agregado.

Distinto en la versión 3.5: El método `read()` ahora acepta el argumento `None`.

Distinto en la versión 3.6: Acepta un objeto similar a una ruta (*path-like object*).

13.3.2 (De)compresión incremental

class `bz2.BZ2Compressor` (*compresslevel=9*)

Crea un nuevo objeto compresor. Este objeto puede usarse para comprimir datos de forma incremental. Para comprimir en un solo paso, use la función `compress()` en su lugar.

compresslevel, si se proporciona, debe ser un número entero entre 1 y 9. El valor predeterminado es 9.

compress (*data*)

Provee datos al objeto del compresor. Retorna un fragmento de datos comprimidos si es posible, o una cadena de bytes vacía de lo contrario.

Cuando haya terminado de proporcionar datos al compresor, llame al método `flush()` para finalizar el proceso de compresión.

flush ()

Termina el proceso de compresión. Retorna los datos comprimidos que quedan en los búferes internos.

El objeto compresor no puede usarse después de que se haya llamado a este método.

class `bz2.BZ2Decompressor`

Crea un nuevo objeto descompresor. Este objeto puede usarse para descomprimir datos de forma incremental. Para descomprimir en un solo paso, use la función `decompress()` en su lugar.

Nota: Esta clase no maneja de forma transparente las entradas que contienen múltiples flujos comprimidos, a diferencia de `decompress()` y `BZ2File`. Si necesitas descomprimir una entrada de flujo múltiple con `BZ2Decompressor`, debe usar un nuevo descompresor para cada flujo (stream).

decompress (*data*, *max_length=-1*)

Descomprime *datos* (un *bytes-like object*), retornando datos sin comprimir como bytes. Algunos de los *datos* pueden almacenarse internamente para su uso en llamadas posteriores a `decompress()`. Los datos retornados deben concatenarse con la salida de cualquier llamada anterior a `decompress()`.

Si `max_length` no es negativo, retorna como máximo `max_length` bytes de datos descomprimidos. Si se alcanza este límite y se pueden producir más resultados, el atributo `needs_input` se establecerá en `False`. En este caso, la siguiente llamada a `decompress()` puede proporcionar *datos* como `b""` para obtener más de la salida.

Si todos los datos de entrada se descomprimieron y retornaron (ya sea porque esto era menor que `max_length` bytes, o porque `max_length` era negativo), el atributo `needs_input` se establecerá en `True`.

Intentar descomprimir datos una vez que se alcanza el final de la transmisión genera un `EOFError`. Cualquier dato encontrado después del final del flujo se ignora y se guarda en el atributo `unused_data`.

Distinto en la versión 3.5: Añadido el parámetro `max_length`

eof

`True` si se ha alcanzado el marcador de fin de flujo.

Nuevo en la versión 3.3.

unused_data

Datos encontrados después del final del flujo comprimido.

Si se accede a este atributo antes de que se haya alcanzado el final del flujo, su valor será `b''`.

needs_input

`False` si el método `decompress()` puede proporcionar más datos descomprimidos antes de requerir una nueva entrada sin comprimir.

Nuevo en la versión 3.5.

13.3.3 (Des)comprimir en un solo paso

`bz2.compress(data, compresslevel=9)`

Comprime *datos*, un *objetos tipo binarios*.

`compresslevel`, si se proporciona, debe ser un número entero entre 1 y 9. El valor predeterminado es 9.

Para compresión incremental, use `BZ2Compressor` en su lugar.

`bz2.decompress(data)`

Descomprime *datos*, un *objetos tipo binarios*.

Si *data* es la concatenación de múltiples flujos comprimidos, descomprime todos los flujos.

Para la descompresión incremental, use `BZ2Decompressor` en su lugar.

Distinto en la versión 3.3: Se agregó soporte para entradas de flujo múltiple.

13.3.4 Ejemplos de uso

Aquí hay algunos ejemplos del uso típico del módulo `bz2`.

Usando `compress()` y `decompress()` para demostrar una compresión de ida y vuelta (*round-trip*):

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> c = bz2.compress(data)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> len(data) / len(c)  # Data compression ratio
1.513595166163142
>>> d = bz2.decompress(c)
>>> data == d  # Check equality to original object after round-trip
True
```

Usando *BZ2Compressor* para compresión incremental:

```
>>> import bz2
>>> def gen_data(chunks=10, chunksize=1000):
...     """Yield incremental blocks of chunksize bytes."""
...     for _ in range(chunks):
...         yield b"z" * chunksize
...
>>> comp = bz2.BZ2Compressor()
>>> out = b""
>>> for chunk in gen_data():
...     # Provide data to the compressor object
...     out = out + comp.compress(chunk)
...
>>> # Finish the compression process. Call this once you have
>>> # finished providing data to the compressor.
>>> out = out + comp.flush()
```

El ejemplo anterior utiliza un flujo de datos bastante «no aleatoria» (un flujo de fragmentos *b»z»*). Los datos aleatorios tienden a comprimirse mal, mientras que los datos ordenados y repetitivos generalmente producen una alta relación de compresión.

Escribiendo y leyendo un archivo comprimido con *bzip2* en modo binario:

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> with bz2.open("myfile.bz2", "wb") as f:
...     # Write compressed data to file
...     unused = f.write(data)
>>> with bz2.open("myfile.bz2", "rb") as f:
...     # Decompress data from file
...     content = f.read()
>>> content == data  # Check equality to original object after round-trip
True
```

13.4 lzma — Compresión utilizando el algoritmo LZMA

Nuevo en la versión 3.3.

Código fuente: `Lib/lzma.py`

Este módulo provee clases y funciones de conveniencia para comprimir y descomprimir datos utilizando el algoritmo de compresión LZMA. También se incluye una interfaz de fichero soportando el `.xz` y formatos de fichero `.lzma` heredados utilizados por la utilidad `xz`, así como flujos comprimidos sin procesar.

La interfaz que provee este módulo es muy similar al del módulo `bz2`. Sin embargo, note que `LZMAFile` no es seguro en hilos, a diferencia de `bz2.BZ2File`, así que si necesita utilizar una única instancia `LZMAFile` desde múltiples hilos, es necesario protegerse con un bloqueo (*lock*).

exception `lzma.LZMAError`

Esta excepción es generada cuando un error ocurre durante la compresión o descompresión, o mientras se inicializa el estado de compresor/descompresor.

13.4.1 Leyendo y escribiendo ficheros comprimidos

`lzma.open(filename, mode="rb", *, format=None, check=-1, preset=None, filters=None, encoding=None, errors=None, newline=None)`

Abre un fichero comprimido LZMA in binario o modo texto, retornando un *file object*.

El argumento *filename* puede ser un nombre de fichero real (dado como un objeto *str*, *bytes* o *path-like*), en cuyo caso el fichero nombrado es abierto, o puede ser un objeto de fichero existente para leer o escribir.

El argumento *mode* puede ser cualquiera de "r", "rb", "w", "wb", "x", "xb", "a" o "a" para modo binario, o "rt", "wt", "xt", o "at" para modo texto. Por defecto es "rb".

Al abrir un fichero para lectura, los argumentos *format* y *filters* tienen el mismo significado que para `LZMADecompressor`. En este caso, los argumentos *check* y *preset* no deberían ser utilizados.

Al abrir un fichero para lectura, los argumentos *format*, *check*, *preset* y *filters* tienen el mismo significado que para `LZMACompressor`.

Para modo binario, esta función es equivalente al constructor `LZMAFile(filename, mode, ...)`. En este caso, los argumentos *encoding*, *errors* y *newline* no deben ser proveídos.

Para modo texto, un objeto `LZMAFile` es creado, y envuelto en una instancia `io.TextIOWrapper` con codificación específica, comportamiento de manejo de errores, y codificación(es) de línea.

Distinto en la versión 3.4: Agregado soporte para los modos "x", "xb" y "xt".

Distinto en la versión 3.6: Acepta un *path-like object*.

class `lzma.LZMAFile(filename=None, mode="r", *, format=None, check=-1, preset=None, filters=None)`

Abre un fichero comprimido LZMA en modo binario.

Un `LZMAFile` puede envolver un ya abierto *file object*, u operar directamente en un fichero nombrado. El argumento *filename* especifica el objeto de fichero a envolver, o el nombre del fichero para abrir (como un *str*, *bytes* o un objeto *path-like*). Al envolver un objeto de fichero existente, el fichero envuelto no será cerrado cuando el `LZMAFile` es cerrado.

El argumento *mode* puede ser "r" para lectura (por defecto), "w" para sobrescritura, "x" para creación exclusiva, o "a" para agregado. Estos pueden ser equivalentemente dados como "rb", "wb", "xb" y "ab" respectivamente.

Si *filename* es un objeto de fichero (en lugar de un nombre de fichero actual), un modo de "w" no trunca el fichero, y en cambio es equivalente a "a".

Al abrir un fichero para escritura, el fichero de entrada puede ser la concatenación para múltiples flujos comprimidos separados. Estos son transparentemente decodificados como un único flujo lógico.

Al abrir un fichero para lectura, los argumentos *format* y *filters* tienen el mismo significado que para `LZMADecompressor`. En este caso, los argumentos *check* y *preset* no deberían ser utilizados.

Al abrir un fichero para lectura, los argumentos *format*, *check*, *preset* y *filters* tienen el mismo significado que para `LZMACompressor`.

`LZMAFile` soporta todos los miembros especificados por `io.BufferedIOBase`, excepto por `detach()` y `truncate()`. La declaración de iteración y `with` son soportados.

El siguiente método también es proveído:

peek (*size=-1*)

Retorna datos almacenados en búfer sin avanzar la posición del fichero. Al menos un byte de datos será retornado, a menos que EOF es alcanzado. El número exacto de bytes retornado no está especificado (el argumento *size* es ignorado).

Nota: Mientras que llamar `peek()` no cambia la posición de fichero del `LZMAFile`, puede cambiar la posición del objeto de fichero subyacente (por ejemplo si el `LZMAFile` fue construido pasando un objeto de fichero por *filename*).

Distinto en la versión 3.4: Agregado soporte para los modos "x" y "xb".

Distinto en la versión 3.5: El método `read()` acepta ahora un argumento de `None`.

Distinto en la versión 3.6: Acepta un *path-like object*.

13.4.2 Comprimiendo y descomprimiendo datos en memoria

class `lzma.LZMACompressor` (*format=FORMAT_XZ, check=-1, preset=None, filters=None*)

Crea un objeto compresor, el cual puede ser utilizado para comprimir datos incrementalmente.

Para una forma más conveniente de comprimir un único fragmento de datos, vea `compress()`.

El argumento *format* especifica qué formato de contenedor debería ser utilizado. Posibles valores son:

- **FORMAT_XZ:** El formato de contenedor `.xz`. Este es el formato por defecto.
- **FORMAT_ALONE:** El formato de contenedor `.lzma` heredado. Este formato es más limitado que `.xz` – no soporta chequeos de integridad o múltiples filtros.
- **FORMAT_RAW:** Un flujo de datos sin procesar, sin utilizar ningún formato de contenedor. Este especificador de formato no soporta chequeos de integridad, y requiere que siempre especifiques una cadena de filtro personalizada (para compresión y descompresión). Adicionalmente, los datos comprimidos de esta manera no pueden ser descomprimidos utilizando `FORMAT_AUTO` (vea `LZMADecompressor`).

El argumento *check* especifica el tipo de chequeo de integridad a incluir en los datos descomprimidos. Este chequeo es utilizado al descomprimir, para asegurarse que los datos no han sido corrompidos. Los posibles valores son:

- `CHECK_NONE`: Sin verificación de integridad. Este es el valor por defecto (y el único valor aceptable) para `FORMAT_ALONE` y `FORMAT_RAW`.
- `CHECK_CRC32`: Chequeo de Redundancia Cíclica de 32 bits.
- `CHECK_CRC64`: Chequeo de Redundancia Cíclica de 64 bits. Este es el valor por defecto para `FORMAT_XZ`.
- `CHECK_SHA256`: Algoritmo Hash Seguro de 256 bits.

Si el chequeo especificado no es soportado, un `LZMAError` es generado.

Las configuraciones de compresión pueden ser especificadas como un nivel de compresión predefinido (con el argumento *preset*), o en detalle como una cadena de filtro personalizada (con el argumento *filters*).

El argumento *preset* (si es proveído) debería ser un entero entre `0` y `9` (inclusive), opcionalmente OR con la constante `PRESET_EXTREME`. Si no se proporciona *preset* ni *filters*, el comportamiento por defecto es utilizar `PRESET_DEFAULT` (nivel preestablecido 6). Altos preestablecidos producen salidas más pequeñas, pero vuelve el proceso de compresión más lento.

Nota: En adición a ser más CPU-intensivo, la compresión con preestablecidos más altos también requiere mucha más memoria (y produce salida que necesita más memoria a descomprimir). Con un preestablecido de

9 por ejemplo, la sobrecarga para un objeto `LZMACompressor` puede ser tan alto como 800MiB. Por esta razón, es generalmente mejor quedarse con el preestablecido por defecto.

El argumento *filters* (si es proveído) debería ser un especificador de cadena de filtro. Vea *Especificando cadenas de filtro personalizadas* para detalles.

compress (*data*)

Comprime *data* (un objeto *bytes*), retornando un objeto *bytes* conteniendo información comprimida por al menos parte de la entrada. Algo de *data* puede ser almacenado internamente, para uso en llamadas posteriores a `compress()` y `flush()`. La información retornada debería ser concatenada con la salida en cualquier llamada previa a `compress()`.

flush ()

Finiquita el proceso de compresión, retornando un objeto *bytes* conteniendo cualquier información almacenada en los búferes internos del compresor.

El compresor no puede ser utilizado después de que este método es llamado.

class `lzma.LZMADecompressor` (*format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

Crea un objeto descompresor, el cual puede ser utilizado para descomprimir datos incrementalmente.

Para una forma más conveniente de descomprimir un flujo completo de compresión a la vez, vea `decompress()`.

El argumento *format* especifica el formato de contenedor que debería ser utilizado. El valor por defecto es `FORMAT_AUTO`, el cual puede descomprimir los ficheros `.xz` y `.lzma`. Otros posibles valores son `FORMAT_XZ`, `FORMAT_ALONE`, y `FORMAT_RAW`.

El argumento *memlimit* especifica un límite (en bytes) en la cantidad de memoria que el descompresor puede utilizar. Cuando este argumento es utilizado, la descompresión fallará con un `LZMAError` si no es posible descomprimir la entrada dentro del límite de memoria dado.

El argumento *filters* especifica la cadena de filtro que fue utilizado para crear el flujo que se descomprime. El argumento es requerido si *format* es `FORMAT_RAW`, pero no debería ser utilizado para otros formatos. Vea *Especificando cadenas de filtro personalizadas* para más información sobre cadenas de filtro.

Nota: Esta clase no maneja transparentemente entradas que contienen múltiples flujos comprimidos, a diferencia de `decompress()` y `LZMAFile`. Para descomprimir una entrada multi-flujo con `LZMADecompressor`, debería crear un nuevo descompresor para cada flujo.

decompress (*data*, *max_length=-1*)

Descomprime *data* (un *bytes-like object*), retornando información sin comprimir como bytes. Alguna *data* puede ser almacenada internamente, para uso en llamadas posteriores a `decompress()`. La información retornada debería ser concatenada con la salida de cualquier llamada anterior a `decompress()`.

Si *max_length* no es negativo, retorna al menos *max_length* bytes para descomprimir información. Si este límite es alcanzado y salidas adicionales pueden ser producidas, el atributo `needs_input` será establecido a `False`. En este caso, la siguiente llamada a `decompress()` podría proveer *data* como `"b "` para obtener más de la salida.

Si toda la información ingresada fue descomprimida y retornada (ya sea porque esto fue menos que *max_length* bytes, o porque *max_length* fue negativo), el atributo `needs_input` será establecido a `True`.

Intentar descomprimir la información descomprimida después de que el fin del flujo es alcanzado genera un `EOFError`. Cualquier información encontrada después de que el fin del flujo es ignorada y guardada en el atributo `unused_data`.

Distinto en la versión 3.5: Agregado el parámetro *max_length*.

check

El ID del chequeo de integridad utilizado por el flujo de entrada. Esto puede ser `CHECK_UNKNOWN` hasta que suficiente de la entrada ha sido decodificada para determinar qué chequeo de integridad utiliza.

eof

True si el marcador de fin-de-flujo ha sido alcanzado.

unused_data

Información encontrada después del fin del flujo comprimido.

Antes de que el fin del flujo es alcanzado, este será "b".

needs_input

False si el método `decompress()` puede proveer más información descomprimida antes de requerir nueva entrada descomprimida.

Nuevo en la versión 3.5.

`lzma.compress(data, format=FORMAT_XZ, check=-1, preset=None, filters=None)`

Comprime *data* (un objeto `bytes`), retornando la información comprimida como un objeto `bytes`.

Vea `LZMACompressor` arriba para una descripción de los argumentos *format*, *check*, *preset* y *filters*.

`lzma.decompress(data, format=FORMAT_AUTO, memlimit=None, filters=None)`

Descomprime *data* (un objeto `bytes`), retornando la información descomprimida como un objeto `bytes`.

Si *data* es la concatenación de múltiples flujos comprimidos distintos, descomprime todos esos flujos, y retorna la concatenación de los resultados.

Vea `LZMADecompressor` arriba para una descripción de los argumentos *format*, *memlimit* y *filters*.

13.4.3 Misceláneas

`lzma.is_check_supported(check)`

Retorna True si el chequeo de integridad dado es soportado en este sistema.

CHECK_NONE y CHECK_CRC32 son soportados siempre. CHECK_CRC64 y CHECK_SHA256 pueden no estar disponibles si está utilizando una versión de `liblzma` que fue compilada con un conjunto de funciones limitado.

13.4.4 Especificando cadenas de filtro personalizadas

Un especificador de cadena de filtro es una secuencia de diccionarios, donde cada diccionario contiene el ID y opciones para un único filtro. Cada diccionario debe contener la llave "id", y puede contener llaves adicionales para especificar opciones filtro-dependientes. Los ID de filtro válidos son como sigue:

- **Filtro de compresión:**

- FILTER_LZMA1 (para uso con FORMAT_ALONE)
- FILTER_LZMA2 (para uso con FORMAT_XZ y `bytesFORMAT_RAW`)

- **Filtro delta:**

- FILTER_DELTA

- **Filtros *Branch-Call-Jump (BCJ)*:**

- FILTER_X86
- FILTER_IA64
- FILTER_ARM
- FILTER_ARMTHUMB
- FILTER_POWERPC
- FILTER_SPARC

Una cadena de filtro puede consistir de hasta 4 filtros, y no puede estar vacía. El último filtro en la cadena debe ser un filtro de compresión, y cualquier otro filtro debe ser un filtro delta o BCJ.

Los filtros de compresión soportan las siguientes opciones (especificadas como entradas adicionales):

- `preset`: Un ajuste de compresión a utilizar como una fuente de valores por defecto para opciones que no están especificadas explícitamente.
- `dict_size`: Tamaño del diccionario en bytes. Esto debería estar entre 4 kiB y 1.5 GiB (inclusive).
- `lc`: Número de bits de contexto literal.
- `lp`: Número de bits de posición literal. La suma `lc + lp` debe ser al menos 4.
- `pb`: Número de bits de posición; debe ser al menos 4.
- `mode`: `MODE_FAST` o `MODE_NORMAL`.
- `nice_len`: Lo que debería ser considerado una «buena longitud» para una coincidencia. Esto debería ser 273 o menos.
- `mf`: Qué buscador de coincidencias utilizar – `MF_HC3`, `MF_HC4`, `MF_BT2`, `MF_BT3`, o `MF_BT4`.
- `depth`: Profundidad de búsqueda máxima utilizada por el buscador de coincidencias. 0 (por defecto) significa seleccionar automáticamente basado en otras opciones de filtro.

El filtro delta almacena las diferencias entre bytes, produciendo más entrada repetitiva para el compresor en ciertas circunstancias. Soporta una opción, `dist`. Esto indica la distancia entre bytes a ser sustraída. Por defecto es 1, por ejemplo toma las diferencias entre bytes adyacentes.

Los filtros BCJ están destinados a ser aplicados a código máquina. Convierten ramas, llamadas y saltos relativos en el código para utilizar el direccionamiento absoluto, con el objetivo de incrementar la redundancia que puede ser explotada por el compresor. Estos filtros soportan una opción, `start_offset`. Esto especifica la dirección que debería ser mapeada al comienzo de la entrada de información. Por defecto es 0.

13.4.5 Ejemplos

Leyendo un fichero comprimido:

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

Creando un fichero comprimido:

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

Comprimiendo información en memoria:

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

Compresión incremental:

```
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
```

(continué en la próxima página)

(proviene de la página anterior)

```
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

Escribiendo información comprimida en fichero ya abierto:

```
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

Creando un fichero comprimido utilizando una cadena de filtro personalizada:

```
import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME},
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
```

13.5 zipfile — Trabajar con archivos ZIP

Source code: [Lib/zipfile.py](#)

El formato de archivo ZIP es un estándar común de archivo y compresión. Este módulo proporciona herramientas para crear, leer, escribir, agregar y listar un archivo ZIP. Cualquier uso avanzado de este módulo requerirá una comprensión del formato, tal como se define en la [PKZIP Application Note](#).

Actualmente, este módulo no maneja archivos ZIP multi-disco. Puede manejar archivos ZIP que usan las extensiones ZIP64 (es decir, archivos ZIP que tienen más de 4 GB de tamaño). Admite el descifrado de archivos cifrados en archivos ZIP, pero actualmente no puede crear un archivo cifrado. El descifrado es extremadamente lento ya que se implementa en Python nativo en lugar de C.

El módulo define los siguientes elementos:

exception `zipfile.BadZipFile`

El error lanzado para archivos ZIP incorrectos.

Nuevo en la versión 3.2.

exception `zipfile.BadZipfile`

Alias de `BadZipFile`, para compatibilidad con versiones anteriores de Python.

Obsoleto desde la versión 3.2.

exception `zipfile.LargeZipFile`

El error lanzado cuando un archivo ZIP requiera la funcionalidad ZIP64 pero no ha sido habilitado.

class `zipfile.ZipFile`

La clase para leer y escribir archivos ZIP. Vea la sección *Objetos ZipFile* para detalles del constructor.

class `zipfile.Path`

Un contenedor compatible con pathlib para archivos zip. Vea la sección *Objetos de ruta* para más detalles.

Nuevo en la versión 3.8.

class `zipfile.PyZipFile`

Clase para crear archivos ZIP que contienen bibliotecas de Python.

class `zipfile.ZipInfo` (*filename*='NoName', *date_time*=(1980, 1, 1, 0, 0, 0))

Clase utilizada para representar información sobre un miembro de un archivo. Las instancias de esta clase son retornadas por los métodos `getinfo()` y `infolist()` de objetos `ZipFile`. La mayoría de los usuarios del módulo `zipfile` no necesitarán crearlos, sino que solo usarán aquellos creados por este módulo. *filename* debe ser el nombre completo del miembro del archivo, y *date_time* debe ser una tupla que contenga seis campos que describan la hora de la última modificación del archivo; los campos se describen en la sección *Objetos ZipInfo*.

`zipfile.is_zipfile` (*filename*)

Retorna `True` si *filename* es un archivo ZIP válido basado en su número mágico; de lo contrario, retorna `False`. *filename* también puede ser un archivo o un objeto similar a un archivo.

Distinto en la versión 3.1: Soporte para archivos y objetos similares a archivos.

`zipfile.ZIP_STORED`

La constante numérica para un miembro de archivo sin comprimir.

`zipfile.ZIP_DEFLATED`

La constante numérica para el método de compresión ZIP habitual. Esto requiere el módulo `zlib`.

`zipfile.ZIP_BZIP2`

La constante numérica para el método de compresión BZIP2. Esto requiere el módulo `bz2`.

Nuevo en la versión 3.3.

`zipfile.ZIP_LZMA`

La constante numérica para el método de compresión LZMA. Esto requiere el módulo `lzma`.

Nuevo en la versión 3.3.

Nota: La especificación del formato del archivo ZIP ha incluido soporte para la compresión bzip2 desde 2001 y para la compresión LZMA desde 2006. Sin embargo, algunas herramientas (incluidas las versiones anteriores de Python) no admiten estos métodos de compresión y pueden negarse a procesar el archivo ZIP por completo o no puede extraer archivos individuales.

Ver también:

PKZIP Application Note Documentación sobre el formato de archivo ZIP por Phil Katz, el creador del formato y los algoritmos utilizados.

Info-ZIP Home Page Información sobre los programas de archivo ZIP del proyecto Info-ZIP y las bibliotecas de desarrollo.

13.5.1 Objetos ZipFile

class `zipfile.ZipFile` (*file*, *mode*='r', *compression*=`ZIP_STORED`, *allowZip64*=`True`, *compresslevel*=`None`, *, *strict_timestamps*=`True`)

Abra un archivo ZIP, donde *file* puede ser una ruta a un archivo (una cadena), un objeto similar a un archivo o un *path-like object*.

El parámetro *mode* debe ser `'r'` para leer un archivo existente, `'w'` para truncar y escribir un nuevo archivo, `'a'` para agregarlo a un archivo existente, o `'x'` para crear y escribir exclusivamente un nuevo archivo. Si *mode* es `'x'` y *file* se refiere a un archivo existente, se generará a `FileExistsError`. Si *mode* es `'a'` y *file* se refiere a un archivo ZIP existente, entonces se le agregan archivos adicionales. Si *file* no se refiere a un archivo ZIP, se agrega un nuevo archivo ZIP al archivo. Esto está destinado a agregar un archivo ZIP a otro archivo (como: `archivo:python.exe`). Si *mode* es `'a'` y el archivo no existe en absoluto, se crea. Si *mode* es `'r'` o `'a'`, el archivo debe poder buscarse.

compression es el método de compresión ZIP que se utiliza al escribir el archivo, y debe ser `ZIP_STORED`, `ZIP_DEFLATED`, `ZIP_BZIP2` o `ZIP_LZMA`; los valores no reconocidos harán que se lance `NotImplementedError`. Si `ZIP_DEFLATED`, `ZIP_BZIP2` o `ZIP_LZMA` se especifica pero el módulo

correspondiente (*zlib*, *bz2* o *lzma*) no está disponible, *RuntimeError* es lanzado. El valor predeterminado es *ZIP_STORED*.

Si *allowZip64* es *True* (el valor predeterminado) *zipfile* creará archivos ZIP que usan las extensiones ZIP64 cuando el archivo zip es mayor que 4 GB. Si es *False* *zipfile* generará una excepción cuando el archivo ZIP requiera extensiones ZIP64.

El parámetro *compresslevel* controla el nivel de compresión que se utilizará al escribir archivos en el archivo. Cuando se utiliza *ZIP_STORED* o *ZIP_LZMA* no tiene ningún efecto. Cuando se usa *ZIP_DEFLATED* se aceptan los enteros 0 a 9 (ver *zlib* para más información). Cuando se utiliza *ZIP_BZIP2* se aceptan enteros 1 a 9 (consulte *bz2* para obtener más información).

El argumento *strictly_timestamps*, cuando se establece en *False*, permite comprimir archivos anteriores a 1980-01-01 a costa de establecer la marca de tiempo en 1980-01-01. Un comportamiento similar ocurre con archivos más nuevos que 2107-12-31, la marca de tiempo también se establece en el límite.

Si el archivo se crea con el modo 'w', 'x' o 'a' y luego *closed* sin agregar ningún archivo al archivo, Las estructuras ZIP apropiadas para un archivo vacío se escribirán en el archivo.

ZipFile también es un manejador de contexto y por lo tanto, admite la declaración *with*. En el ejemplo, *myzip* se cierra después que el conjunto de instrucciones *with* se termine—incluso si se produce una excepción:

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

Nuevo en la versión 3.2: Se agregó la capacidad de usar *ZipFile* como administrador de contexto.

Distinto en la versión 3.3: Soporte agregado para *bzip2* y compresión *lzma*.

Distinto en la versión 3.4: Las extensiones ZIP64 están habilitadas por defecto.

Distinto en la versión 3.5: Se agregó soporte para escribir en secuencias que no se pueden buscar. Se agregó soporte para el modo 'x'.

Distinto en la versión 3.6: Anteriormente, se generó un simple *RuntimeError* para valores de compresión no reconocidos.

Distinto en la versión 3.6.2: El parámetro *file* acepta un *path-like object*.

Distinto en la versión 3.7: Agregue el parámetro *compresslevel*.

Nuevo en la versión 3.8: El Argumento *strict_timestamps* solo palabra clave

ZipFile.close()

Cierra el archivo. Debe llamar a *close()* antes de salir de su programa o no se escribirán registros esenciales.

ZipFile.getinfo(name)

Retorna un objeto *ZipInfo* con información sobre el miembro del archivo *name*. Llamando a *getinfo()* para obtener un nombre que no figura actualmente en el archivo generará un *KeyError*.

ZipFile.infolist()

Retorna una lista que contiene un objeto *ZipInfo* para cada miembro del archivo. Los objetos están en el mismo orden que sus entradas en el archivo ZIP real en el disco si se abrió un archivo existente.

ZipFile.namelist()

Retorna una lista de miembros del archivo por nombre.

*ZipFile.open(name, mode='r', pwd=None, *, force_zip64=False)*

Acceda a un miembro del archivo como un objeto binario similar a un archivo. *name* puede ser el nombre de un archivo dentro del archivo o un objeto *ZipInfo*. El parámetro *mode*, si está incluido, debe ser 'r' (el valor predeterminado) o 'w'. *pwd* es la contraseña utilizada para descifrar archivos ZIP cifrados.

open() también es un administrador de contexto y por lo tanto, soporta *with statement*:

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

Con *mode* 'r', el objeto tipo archivo (`ZipExtFile`) es de solo lectura y proporciona los siguientes métodos: `read()`, `readline()`, `readlines()`, `seek()`, `tell()`, `__iter__()`, `__next__()`. Estos objetos pueden funcionar independientemente del archivo Zip.

Con `mode = 'w'`, se retorna un controlador de archivo escribible, que admite el método `write()`. Mientras está abierto un identificador de archivo escribible, intentar leer o escribir otros archivos en el archivo ZIP generará un `ValueError`.

Al escribir un archivo, si el tamaño del archivo no se conoce de antemano pero puede exceder los 2 GB, pase `force_zip64 = True` para asegurarse de que el formato del encabezado sea capaz de admitir archivos grandes. Si el tamaño del archivo se conoce de antemano, construya un objeto `ZipInfo` con `file_size` establecido, y úselo como parámetro `name`.

Nota: Los métodos `open()`, `read()` y `extract()` pueden tomar un nombre de archivo o un objeto `ZipInfo`. Apreciará esto cuando intente leer un archivo ZIP que contiene miembros con nombres duplicados.

Distinto en la versión 3.6: Se eliminó el soporte de `mode= 'U '`. Use `io.TextIOWrapper` para leer archivos de texto comprimido en modo *universal newlines*.

Distinto en la versión 3.6: `open()` ahora se puede usar para escribir archivos en el archivo con la opción `mode='w'`.

Distinto en la versión 3.6: Llamar a `open()` en un `ZipFile` cerrado generará un `ValueError`. Anteriormente, se planteó a `RuntimeError`.

`ZipFile.extract (member, path=None, pwd=None)`

Extraer un miembro del archivo al directorio de trabajo actual; *member* debe ser su nombre completo o un objeto *ZipInfo*. La información de su archivo se extrae con la mayor precisión posible. *path* especifica un directorio diferente para extraer. *member* puede ser un nombre de archivo o un objeto *ZipInfo*. *pwd* es la contraseña utilizada para archivos cifrados.

Retorna la ruta normalizada creada (un directorio o archivo nuevo).

Nota: Si el nombre de archivo de un miembro es una ruta absoluta, se eliminarán un punto compartido de drive/UNC y las barras diagonales (hacia atrás), ej: `///foo/bar` se convierte en `foo/bar` en Unix y `C:\foo\bar` se convierte en `foo\bar` en Windows. Y todos los componentes “«.»” en un nombre de archivo miembro se eliminarán, ej: `.././foo.././ba..r` se convierte en `foo../ba..r`. En Windows, los caracteres ilegales (: ` , ` < , > ` , ` | ` , ` " ` , ` ' , ` ¿ , ` Y *) se reemplazan por guion bajo (`_`).

Distinto en la versión 3.6: Llamando `extract()` en un `ZipFile` cerrado generará un `ValueError`. Anteriormente, se planteó a `RuntimeError`.

Distinto en la versión 3.6.2: El parámetro *path* acepta un *path-like object*.

`ZipFile.extractall (path=None, members=None, pwd=None)`

Extrae todos los miembros del archivo al directorio de trabajo actual. *path* especifica un directorio diferente para extraer. *members* es opcional y debe ser un subconjunto de la lista retornada por `namelist()`. *pwd* es la contraseña utilizada para archivos cifrados.

Advertencia: Nunca extraiga archivos de fuentes no confiables sin inspección previa. Es posible que los archivos se creen fuera de *path*, ej. miembros que tienen nombres de archivo absolutos que comienzan con "/" o nombres de archivo con dos puntos ". . .". Este módulo intenta evitar eso. Ver `extract()` nota.

Distinto en la versión 3.6: Llamar a `extractall()` en un `ZipFile` cerrado generará un `ValueError`. Anteriormente, se planteó a `RuntimeError`.

Distinto en la versión 3.6.2: El parámetro *path* acepta un *path-like object*.

`ZipFile.printdir()`

Imprime una tabla de contenido para el archivo en `sys.stdout`.

`ZipFile.setpassword(pwd)`

Establece `pwd` como contraseña predeterminada para extraer archivos cifrados.

`ZipFile.read(name, pwd=None)`

Retorna los bytes del archivo `name` en el archivo. `name` es el nombre del archivo en el archivo, o un objeto `ZipInfo`. El archivo debe estar abierto para leer o agregar. `pwd` es la contraseña utilizada para los archivos cifrados y, si se especifica, anulará la contraseña predeterminada establecida con `setpassword()`. Llamar a `read()` en un archivo Zip que utiliza un método de compresión que no sea `ZIP_STORED`, `ZIP_DEFLATED`, `ZIP_BZIP2` o `ZIP_LZMA` generará un `NotImplementedError`. También se generará un error si el módulo de compresión correspondiente no está disponible.

Distinto en la versión 3.6: Llamando `read()` en un `ZipFile` cerrado generará un `ValueError`. Anteriormente, se planteó a `RuntimeError`.

`ZipFile.testzip()`

Lee todos los archivos en el archivo y verifica sus CRC y encabezados de archivo. Retorna el nombre del primer archivo incorrecto o retorna `None`.

Distinto en la versión 3.6: Llamando a `testzip()` en un `ZipFile` cerrado generará un `ValueError`. Anteriormente, se planteó a `RuntimeError`.

`ZipFile.write(filename, arcname=None, compress_type=None, compresslevel=None)`

Escribe el archivo llamado `filename` en el archivo, dándole el nombre de archivo `arcname` (por defecto, será el mismo que `filename`, pero sin una letra de unidad y con los separadores de ruta principales eliminados). Si se proporciona, `compress_type` anula el valor dado para el parámetro `compression` al constructor para la nueva entrada. Del mismo modo, `compresslevel` anulará el constructor si se proporciona. El archivo debe estar abierto con el modo `'w'`, `'x'` o `'a'`.

Nota: Los nombres de archivo deben ser relativos a la raíz del archivo, es decir, no deben comenzar con un separador de ruta.

Nota: Si `arcname` (o `filename`, si `arcname` no se proporciona) contiene un byte nulo, el nombre del archivo en el archivo se truncará en el byte nulo.

Distinto en la versión 3.6: Llamando `write()` en un `ZipFile` creado con el modo `'r'` o un `ZipFile` cerrado generará un `ValueError`. Anteriormente, se planteó a `RuntimeError`.

`ZipFile.writestr(zinfo_or_arcname, data, compress_type=None, compresslevel=None)`

Escribe un registro en el archivo. El contenido es `data`, que puede ser una instancia de `str` o a `bytes`; si es una `str`, primero se codifica como UTF-8. `zinfo_or_arcname` es el nombre del archivo que se le dará en el archivo o una instancia de `ZipInfo`. Si se trata de una instancia, se debe proporcionar al menos el nombre de archivo, la fecha y la hora. Si es un nombre, la fecha y la hora se configuran en la fecha y hora actuales. El archivo debe abrirse con el modo `'w'`, `'x'` o `'a'`.

Si se proporciona, `compress_type` anula el valor dado para el parámetro `compression` al constructor para la nueva entrada, o en `zinfo_or_arcname` (si es una instancia de `ZipInfo`). Del mismo modo, `compresslevel` anulará el constructor si se proporciona.

Nota: Al pasar una instancia de `ZipInfo` como el parámetro `zinfo_or_arcname`, el método de compresión utilizado será el especificado en el miembro `compress_type` de la instancia dada `ZipInfo`. Por defecto, el constructor `ZipInfo` establece este miembro en `ZIP_STORED`.

Distinto en la versión 3.2: El argumento `compress_type`.

Distinto en la versión 3.6: Llamando `writestr()` en un `ZipFile` creado con el modo `'r'` o un `ZipFile` cerrado generará un `ValueError`. Anteriormente, se planteó a `RuntimeError`.

Los siguientes atributos de datos también están disponibles:

`ZipFile.filename`

Nombre del archivo ZIP.

`ZipFile.debug`

El nivel de salida de depuración a usar. Esto se puede configurar de 0 (el valor predeterminado, sin salida) a 3 (la mayor cantidad de salida). La información de depuración se escribe en `sys.stdout`.

`ZipFile.comment`

El comentario asociado con el archivo ZIP como un objeto *bytes*. Si se asigna un comentario a una instancia de *ZipFile* creada con el modo 'w', 'x' o 'a', no debe tener más de 65535 bytes. Los comentarios más largos que esto se truncarán.

13.5.2 Objetos de ruta

class `zipfile.Path` (*root*, *at=""*)

Construye un objeto *Path* a partir de un archivo zip *root* (que puede ser una instancia *ZipFile* o *file* adecuado para pasar al constructor *ZipFile*).

at especifica la ubicación de esta ruta dentro del archivo zip, ej. "dir/file.txt", "dir/" o "". El valor predeterminado es la cadena vacía, que indica la raíz.

Los objetos de ruta exponen las siguientes características de objetos *pathlib.Path*:

Los objetos de ruta se pueden atravesar utilizando el operador `/`.

`Path.name`

El componente final de la ruta.

`Path.open` (*, **)

Invoca *ZipFile.open*() en la ruta actual. Acepta los mismos argumentos que *ZipFile.open*() .

Prudencia: La firma en esta función cambia de manera incompatible en Python 3.9. Para una versión compatible con el futuro, considere usar el paquete de terceros *zippath* (3.0 o posterior).

`Path.iterdir`()

Enumera los hijos del directorio actual.

`Path.is_dir`()

Retorna *True* si el contexto actual hace referencia a un directorio.

`Path.is_file`()

Retorna *True* si el contexto actual hace referencia a un archivo.

`Path.exists`()

Retorna *True* si el contexto actual hace referencia a un archivo o directorio en el archivo zip.

`Path.read_text` (*, **)

Lee el archivo actual como texto unicode. Los argumentos posicionales y de palabras clave se pasan a *io.TextIOWrapper* (excepto *buffer*, que está implícito en el contexto).

`Path.read_bytes`()

Lee el archivo actual como bytes.

13.5.3 Objetos PyZipFile

El constructor `PyZipFile` toma los mismos parámetros que el constructor `ZipFile`, y un parámetro adicional, `optimize`.

class `zipfile.PyZipFile` (*file*, *mode*='r', *compression*=ZIP_STORED, *allowZip64*=True, *optimize*=1)

Nuevo en la versión 3.2: El parámetro `optimize`.

Distinto en la versión 3.4: Las extensiones ZIP64 están habilitadas por defecto.

Las instancias tienen un método además de los objetos `ZipFile`:

writepy (*pathname*, *basename*="", *filterfunc*=None)

Busca archivos `*.py` y agrega el archivo correspondiente al archivo.

Si no se proporcionó el parámetro `optimize` a `PyZipFile` o `-1`, el archivo correspondiente es un archivo `*.pyc`, compilando si es necesario.

Si el parámetro `optimize` a `PyZipFile` era 0, 1 or 2, solo se agregarán a ese archivo los archivos con ese nivel de optimización (ver `compile()`) el archivo, compilando si es necesario.

Si *pathname* es un archivo, el nombre del archivo debe terminar con `.py`, y solo el archivo (correspondiente `*.pyc`) se agrega en el nivel superior (sin información de ruta). Si *pathname* es un archivo que no termina con `.py`, se generará `RuntimeError`. Si es un directorio, y el directorio no es un directorio de paquetes, entonces todos los archivos `*.pyc` se agregan en el nivel superior. Si el directorio es un directorio de paquetes, todos `*.pyc` se agregan bajo el nombre del paquete como una ruta de archivo, y si alguno de los subdirectorios son directorios de paquetes, todos estos se agregan recursivamente en orden ordenado.

basename está destinado solo para uso interno.

filterfunc, si se proporciona, debe ser una función que tome un único argumento de cadena. Se le pasará cada ruta (incluida cada ruta de archivo completa individual) antes de que se agregue al archivo. Si *filterfunc* retorna un valor falso, la ruta no se agregará y si se trata de un directorio se ignorará su contenido. Por ejemplo, si nuestros archivos de prueba están todos en directorios de `test` o comienzan con la cadena `test_`, podemos usar un *filterfunc* para excluirlos

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or fn.startswith('test_')))
>>> zf.writepy('myprog', filterfunc=notests)
```

El método `writepy()` crea archivos con nombres de archivo como este

<code>string.pyc</code>	<code># Top level name</code>
<code>test/__init__.pyc</code>	<code># Package directory</code>
<code>test/testall.pyc</code>	<code># Module test.testall</code>
<code>test/bogus/__init__.pyc</code>	<code># Subpackage directory</code>
<code>test/bogus/myfile.pyc</code>	<code># Submodule test.bogus.myfile</code>

Nuevo en la versión 3.4: El parámetro `filterfunc`.

Distinto en la versión 3.6.2: El parámetro `pathname` acepta un *path-like object*.

Distinto en la versión 3.7: La recursividad ordena las entradas del directorio.

13.5.4 Objetos ZipInfo

Las instancias de la clase `ZipInfo` son retornadas por los métodos `getinfo()` y `infolist()` de `ZipFile`. Cada objeto almacena información sobre un solo miembro del archivo ZIP.

Hay un método de clase para hacer una instancia de `ZipInfo` para un archivo de sistema de archivos:

classmethod `ZipInfo.from_file(filename, arcname=None, *, strict_timestamps=True)`

Construye una instancia de `ZipInfo` para un archivo en el sistema de archivos, en preparación para agregarlo a un archivo zip.

filename debe ser la ruta a un archivo o directorio en el sistema de archivos.

Si se especifica *arcname*, este es usado como el nombre dentro del archivo. Si no se especifica *arcname*, el nombre será el mismo que *filename*, pero con cualquier letra de unidad y separadores de ruta principales eliminados.

El argumento *strictly_timestamps*, cuando se establece en `False`, permite comprimir archivos anteriores a 1980-01-01 a costa de establecer la marca de tiempo en 1980-01-01. Un comportamiento similar ocurre con archivos más nuevos que 2107-12-31, la marca de tiempo también se establece en el límite.

Nuevo en la versión 3.6.

Distinto en la versión 3.6.2: El parámetro *filename* acepta un *path-like object*.

Nuevo en la versión 3.8: El Argumento *strict_timestamps* solo palabra clave

Las instancias tienen los siguientes métodos y atributos:

`ZipInfo.is_dir()`

Retorna `True` si este miembro del archivo es un directorio.

Utiliza el nombre de la entrada: los directorios siempre deben terminar con `/`.

Nuevo en la versión 3.6.

`ZipInfo.filename`

Nombre del archivo en el archivo.

`ZipInfo.date_time`

La hora y fecha de la última modificación al miembro del archivo. Esta es una tupla de seis valores:

Índice	Valor
0	Año (≥ 1980)
1	Mes (basado en uno)
2	Día del mes (basado en uno)
3	Horas (basados en cero)
4	Minutos (basados en cero)
5	Segundos (basado en cero)

Nota: El formato de archivo ZIP no admite marcas de tiempo anteriores a 1980.

`ZipInfo.compress_type`

Tipo de compresión para la miembro del archivo.

`ZipInfo.comment`

Comenta para el miembro de archivo individual como un objeto *bytes*.

`ZipInfo.extra`

Datos de campo de expansión. La [PKZIP Application Note](#) contiene algunos comentarios sobre la estructura interna de los datos contenidos en este objeto *bytes*.

`ZipInfo.create_system`

Sistema que creó el archivo ZIP.

`ZipInfo.create_version`

Versión PKZIP que creó el archivo ZIP.

`ZipInfo.extract_version`

Se necesita la versión PKZIP para extraer el archivo.

`ZipInfo.reserved`

Debe ser cero.

`ZipInfo.flag_bits`

Bits de bandera ZIP.

`ZipInfo.volume`

Número de volumen del encabezado del archivo.

`ZipInfo.internal_attr`

Atributos internos.

`ZipInfo.external_attr`

Atributos de archivo externo.

`ZipInfo.header_offset`

Byte desplazado al encabezado del archivo.

`ZipInfo.CRC`

CRC-32 del archivo sin comprimir.

`ZipInfo.compress_size`

Tamaño de los datos comprimidos.

`ZipInfo.file_size`

Tamaño del archivo sin comprimir.

13.5.5 Interfaz de línea de comandos

El módulo `zipfile` proporciona una interfaz de línea de comandos simple para interactuar con archivos ZIP.

Si desea crear un nuevo archivo ZIP, especifique su nombre después de la opción `-c` y luego enumere los nombres de archivo que deben incluirse:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

Pasar un directorio también es aceptable:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

Si desea extraer un archivo ZIP en el directorio especificado, use la opción `-e`:

```
$ python -m zipfile -e monty.zip target-dir/
```

Para obtener una lista de los archivos en un archivo ZIP, use la opción `-l`:

```
$ python -m zipfile -l monty.zip
```

Opciones de línea de comando

```
-l <zipfile>
--list <zipfile>
    Lista de archivos en un archivo zip.

-c <zipfile> <source1> ... <sourceN>
--create <zipfile> <source1> ... <sourceN>
    Crea el archivo zip a partir de archivos fuente.

-e <zipfile> <output_dir>
--extract <zipfile> <output_dir>
    Extrae el archivo zip en el directorio de destino.

-t <zipfile>
--test <zipfile>
    Prueba si el archivo zip es válido o no.
```

13.5.6 Problemas de descompresión

La extracción en el módulo zipfile puede fallar debido a algunos problemas que se enumeran a continuación.

Del archivo mismo

La descompresión puede fallar debido a una contraseña incorrecta / suma de verificación CRC / formato ZIP o método / descifrado de compresión no compatible.

Limitaciones del sistema de archivos

Exceder las limitaciones en diferentes sistemas de archivos puede causar que la descompresión falle. Como los caracteres permitidos en las entradas del directorio, la longitud del nombre del archivo, la longitud de la ruta, el tamaño de un solo archivo y la cantidad de archivos, etc.

Limitaciones de recursos

La falta de memoria o volumen de disco conduciría a la descompresión fallida. Por ejemplo, las bombas de descompresión (también conocido como [ZIP bomb](#)) se aplican a la biblioteca de archivos zip que pueden causar el agotamiento del volumen del disco.

Interrupción

La interrupción durante la descompresión, como presionar control-C o matar el proceso de descompresión, puede dar como resultado una descompresión incompleta del archivo.

Comportamientos predeterminados de extracción

No conocer los comportamientos de extracción predeterminados puede causar resultados de descompresión inesperados. Por ejemplo, al extraer el mismo archivo dos veces, sobrescribe los archivos sin preguntar.

13.6 tarfile — Leer y escribir archivos tar

Código fuente: [Lib/tarfile.py](#)

El módulo `tarfile` hace posible escribir y leer archivos tar, incluyendo aquellos que hacen uso de compresión gzip, bz2 y lzma. Utiliza el módulo `zipfile` para leer o escribir archivos `.zip`, o las funciones de nivel superior en `shutil`.

Algunos hechos y cifras:

- lee y escribe archivos comprimidos `gzip`, `bz2` y `lzma` si los respectivos módulos están disponibles.
- soporte de lectura/escritura para el formato POSIX.1-1988 (ustar).
- soporte de lectura/escritura para el formato GNU tar incluyendo extensiones `longname` y `longlink`, soporte de solo escritura para todas las variantes de extensiones de `sparse` (archivo disperso) incluyendo restablecimiento de archivos dispersos.
- soporte de lectura/escritura para el formato POSIX.1-2001 (pax).
- gestiona directorios, archivos regulares, enlaces duros, enlaces simbólicos, fifos, dispositivos de carácter, dispositivos de bloque y puede adquirir y restaurar información de archivo como marca de tiempo, permisos de acceso y dueño.

Distinto en la versión 3.3: Añadido soporte para compresión `lzma`.

`tarfile.open` (*name=None, mode='r', fileobj=None, bufsize=10240, **kwargs*)

Retorna un objeto `TarFile` para el nombre de ruta *name*. Para información detallada en los objetos de la clase `TarFile` y los argumentos por palabra clave que están permitidos, dirígete a [Objetos TarFile](#).

mode tiene que ser una cadena de texto en el formato `'filemode[:compression]'`, adquiere el valor de `'r'` de forma predeterminada. Aquí hay una lista completa de combinaciones de modo:

modo	acción
'r' o 'r:*	Abrir para leer con compresión transparente (recomendado).
'r:'	Abrir para leer exclusivamente sin compresión.
'r:gz'	Abrir para leer con compresión gzip.
'r:bz2'	Abrir para leer con compresión bzip2.
'r:xz'	Abrir para leer con compresión lzma.
'x' o 'x:'	Crear un archivo <i>tarfile</i> exclusivamente sin compresión. Lanza una excepción <code>FileExistsError</code> si el archivo ya existe.
'x:gz'	Crear un archivo tar con compresión gzip. Lanza una excepción <code>FileExistsError</code> si ya existe.
'x:bz2'	Crear un archivo tar con compresión bzip2. Lanza una excepción <code>FileExistsError</code> si ya existe.
'x:xz'	Crear un archivo tar con compresión lzma. Lanza una excepción <code>FileExistsError</code> si ya existe.
'a' or 'a:'	Abrir para anexar sin compresión. El archivo se crea si no existe.
'w' o 'w:'	Abrir para escritura con compresión lzma.
'w:gz'	Abrir para escritura con compresión gzip.
'w:bz2'	Abrir para escritura con compresión bzip2.
'w:xz'	Abrir para escritura con compresión lzma.

Ten en cuenta que `'a:gz'`, `'a:bz2'` or `'a:xz'` no son posibles. Si *mode* no es apropiado para abrir ciertos archivos (comprimidos) para lectura, se lanza una excepción de tipo `ReadError`. Usa el modo `'r'` para evitar esto. Si el método de compresión no es admitido, se lanza una excepción `CompressionError`.

Si *fileobj* es especificado, se utiliza como alternativa a *file object* abierto en modo binario para *name*. Debe estar en la posición 0.

Para los modos 'w:gz', 'r:gz', 'w:bz2', 'r:bz2', 'x:gz', 'x:bz2', *tarfile.open()* acepta el argumento por palabra clave *compresslevel* (por defecto 9) para especificar el nivel de compresión del archivo.

Para propósitos especiales, hay un segundo formato para *modo*: 'filemode|[compression]'. *tarfile.open()* retornará un objeto *TarFile* que procesa sus datos como un *stream* de bloques. No se realizará una búsqueda aleatoria en el archivo. Si se proporciona, *fileobj* puede ser un objeto con los métodos *read()* o *write()* (dependiendo del *modo*). *bufsize* especifica el *blocksize* y por default tiene un valor de 20 * 512 bytes. Utiliza esta variante en combinación con por ejemplo *sys.stdin*, un socket *file object* o un *tape device*. Sin embargo, dicho objeto *TarFile* está limitado en el aspecto de que no permite acceso aleatorio. Consulta *Ejemplos*. Los modos posibles:

Modo	Acción
'r *'	Abre un <i>stream</i> de bloques tar para leer con compresión transparente.
'r '	Abre un <i>stream</i> de bloques tar sin comprimir para lectura.
'r gz'	Abre un <i>stream</i> comprimido con gzip para lectura.
'r bz2'	Abre un <i>stream</i> bzip2 comprimido para lectura.
'r xz'	Abre un <i>stream</i> lzma comprimido para lectura.
'w '	Abre un <i>stream</i> sin comprimir para escritura.
'w gz'	Abre un <i>stream</i> gzip comprimido para escritura.
'w bz2'	Abre un <i>stream</i> bzip2 comprimido para escritura.
'w xz'	Abre un <i>stream</i> lzma comprimido para escritura.

Distinto en la versión 3.5: El modo 'x' (creación exclusiva) fue añadido.

Distinto en la versión 3.6: El parámetro *name* acepta un objeto *path-like object*.

class tarfile.TarFile

Clase para leer y escribir archivos tar. No utilices esta clase directamente; usa *tarfile.open()* en su lugar. Consulta *Objetos TarFile*.

tarfile.is_tarfile(name)

Retorna *True* si *name* es un archivo tar, que el módulo *tarfile* puede leer.

El módulo *tarfile* define las siguientes excepciones:

exception tarfile.TarError

Clase base para todas las excepciones del módulo *tarfile*.

exception tarfile.ReadError

Se lanza cuando un archivo tar se abre, que no puede ser manejado por el módulo *tarfile* o de alguna manera no es válido.

exception tarfile.CompressionError

Se lanza cuando un método de compresión no tiene soporte o cuando la información no puede ser decodificada de manera apropiada.

exception tarfile.StreamError

Se lanza para limitaciones que son comunes para objetos *stream-like TarFile*.

exception tarfile.ExtractError

Se lanza para errores no fatales cuando se utiliza *TarFile.extract()*, pero solo si *TarFile.errorlevel*== 2.

exception tarfile.HeaderError

Se lanza por *TarInfo.frombuf()* si el buffer que obtiene es invalido.

exception tarfile.FilterError

Base class for members *refused* by filters.

tarinfo

Information about the member that the filter refused to extract, as [TarInfo](#).

exception tarfile.AbsolutePathError

Raised to refuse extracting a member with an absolute path.

exception tarfile.OutsideDestinationError

Raised to refuse extracting a member outside the destination directory.

exception tarfile.SpecialFileError

Raised to refuse extracting a special file (e.g. a device or pipe).

exception tarfile.AbsoluteLinkError

Raised to refuse extracting a symbolic link with an absolute path.

exception tarfile.LinkOutsideDestinationError

Raised to refuse extracting a symbolic link pointing outside the destination directory.

Las siguientes constantes están disponibles a nivel de módulo:

tarfile.ENCODING

La codificación para caracteres predeterminada: 'utf-8' en Windows, de otra manera, el valor retornado por `sys.getfilesystemencoding()`.

Cada una de las siguientes constantes define un formato de archivo tar que el módulo `tarfile` puede crear. Ve la sección: [Formatos tar con soporte](#) para más detalles.

tarfile.USTAR_FORMAT

Formato POSIX.1-1988 (ustar).

tarfile.GNU_FORMAT

Formato GNU tar.

tarfile.PAX_FORMAT

Formato POSIX.1-2001 (pax).

tarfile.DEFAULT_FORMAT

El formato predeterminado para crear archivos. Es actualmente `PAX_FORMAT`.

Distinto en la versión 3.8: El formato predeterminado para nuevos archivos fue cambiado de `GNU_FORMAT` a `PAX_FORMAT`

Ver también:

Módulo `zipfile` Documentación del módulo estándar `zipfile`.

[Operaciones de archivado](#) Documentación para las facilidades de archivos de más alto nivel proporcionadas por el módulo estándar `shutil`.

[Manual GNU tar, formato básico tar](#) Documentación para archivos de tipo tar, incluyendo extensiones GNU tar.

13.6.1 Objetos `TarFile`

El objeto `TarFile` provee una interfaz a un archivo tar. Un archivo tar es una secuencia de bloques. Un miembro de archivos (un archivo almacenado) está hecho de un bloque de encabezado seguido de bloques de datos. Es posible almacenar un archivo dentro de un tar múltiples veces. Cada archivo miembro está representado por un objeto `TarInfo`, consulta [Objetos TarInfo](#) para más detalles.

Un objeto `TarFile` puede ser usado como un administrador de contexto en una declaración `with`. Será automáticamente cerrado cuando el bloque sea completado. Ten en cuenta que en el evento de una excepción un archivo abierto para escritura no será terminado; solo el objeto de archivo interno será cerrado. Consulta la sección [Ejemplos](#) para un caso de uso.

Nuevo en la versión 3.2: Añadido soporte para el protocolo de administración de contexto.

```
class tarfile.TarFile (name=None, mode='r', fileobj=None, format=DEFAULT_FORMAT, tarinfo=
TarInfo, dereference=False, ignore_zeros=False, encoding=ENCODING,
errors='surrogateescape', pax_headers=None, debug=0, errorlevel=1)
```

Los siguientes argumentos son opcionales y también se pueden acceder como atributos de instancia.

name es el nombre de ruta del archivo. *name* puede ser un objeto *path-like object*. Puede ser omitido si *fileobj* se proporciona. En este caso, el atributo *name* del objeto de archivo se usa si existe.

mode puede ser 'r' para leer desde un archivo existente, 'a' para adjuntar datos a un archivo existente, "w" para crear un nuevo archivo sobrescribiendo uno existente, o 'x' para crear un nuevo archivo únicamente si este no existe.

Si *fileobj* es proporcionado, se utiliza para escritura o lectura de datos. Si puede ser determinado, *mode* puede ser anulado por el modo de *fileobj*. *fileobj* será usado desde la posición 0.

Nota: *fileobj* no se cierra cuando *TarFile* se cierra.

format controla el formato de archivo para la escritura. Debe ser una de las constantes *USTAR_FORMAT*, *GNU_FORMAT* o *PAX_FORMAT* que se definen a nivel de módulo. Al leer, el formato se detectará automáticamente, incluso si hay diferentes formatos en un solo archivo.

El argumento *tarinfo* puede ser utilizado para reemplazar la clase predeterminada *TarInfo* con una diferente.

Si *dereference* tiene el valor de *False*, añade enlaces simbólicos y duros al archivo. Si tiene el valor de *True*, añade el contenido de archivos objetivo al archivo. Esto no tiene ningún efecto en los sistemas que no admiten enlaces simbólicos.

Si *ignore_zeros* tiene el valor de *False*, trata un bloque vacío como el final del archivo. Si es *True*, omite los bloques vacíos (y no válidos) e intenta obtener tantos miembros como sea posible. Esto solo es útil para leer archivos concatenados o dañados.

debug puede ser establecido con valores desde 0 (sin mensajes de depuración) hasta 3 (todos los mensajes de depuración). Los mensajes son escritos en `sys.stderr`.

errorlevel controls how extraction errors are handled, see the corresponding attribute.

Los argumentos *encoding* y *errors* definen la codificación de caracteres que se utilizará para lectura o escritura del archivo y como los errores de conversión van a ser manejados. La configuración predeterminada funcionará para la mayoría de los usuarios. Mira la sección *Problemas Unicode* para más información a detalle.

El argumento *pax_headers* es un diccionario opcional de cadenas las cuales serán añadidas como un encabezado pax global si el valor de *format* es *PAX_FORMAT*.

Distinto en la versión 3.2: Utiliza 'surrogateescape' como valor predeterminado del argumento *errors*.

Distinto en la versión 3.5: El modo 'x' (creación exclusiva) fue añadido.

Distinto en la versión 3.6: El parámetro *name* acepta un objeto *path-like object*.

```
classmethod TarFile.open (...)
```

Constructor alternativo. La función `tarfile.open()` es un acceso directo a este método de la clase

```
TarFile.getmember (name)
```

Retorna un objeto *TarInfo* para el miembro *name*. Si *name* no puede ser encontrado, entonces una excepción de tipo *KeyError* será lanzada.

Nota: Si un miembro aparece más de una vez en el archivo, se asume que su última aparición es la versión más actualizada.

```
TarFile.getmembers ()
```

Retorna los miembros del archivo como una lista de objetos *TarInfo*. La lista tiene el mismo orden que los miembros del archivo.

`TarFile.getnames()`

Retorna los miembros como una lista de sus nombres. Tiene el mismo orden que la lista retornada por `getmembers()`.

`TarFile.list(verbose=True, *, members=None)`

Imprime en `sys.stdout` una tabla de contenidos. Si `verbose` es `False`, solo los nombres de los miembros son impresos. Si es `True`, se produce una salida de impresión similar a la de `ls -l`. Si el parámetro `members` es proporcionado, debe ser un subconjunto de la lista retornada por `getmembers()`.

Distinto en la versión 3.5: Se agregó el parámetro `members`.

`TarFile.next()`

Retorna el siguiente miembro del archivo como un objeto `TarInfo` cuando `TarFile` se abre para lectura. Retorna `None` si no hay más miembros disponibles.

`TarFile.extractall(path=".", members=None, *, numeric_owner=False, filter=None)`

Extrae todos los miembros de un archivo al directorio de trabajo actual o al directorio de `path`. Si se proporciona el parámetro opcional `members`, debe ser un subconjunto de la lista retornada por el método `getmembers()`. Información de directorio como dueño, fecha de modificación y permisos son establecidos una vez que todos los miembros han sido extraídos. Esto se realiza para trabajar con dos problemáticas: La fecha de modificación de un directorio es modificada cada vez que un archivo es creado en el. Y si los permisos de un directorio no permiten escritura, extraer archivos en el no será posible.

Si `numeric_owner` tiene el valor de `True`, los números uid y gid del archivo tar se utilizan para establecer el dueño/grupo de los archivos extraídos. De lo contrario, se utilizan los valores nombrados del archivo tar.

The `filter` argument, which was added in Python 3.8.17, specifies how `members` are modified or rejected before extraction. See [Extraction filters](#) for details. It is recommended to set this explicitly depending on which `tar` features you need to support.

Advertencia: Nunca extraigas archivos de fuentes no confiables sin una inspección previa. Es posible que los archivos se creen fuera de `path`, Ej. miembros que tienen nombres de archivo absolutos que comienzan con `"/"` o nombres de archivo con dos puntos `".."`.

Set `filter='data'` to prevent the most dangerous security issues, and read the [Extraction filters](#) section for details.

Distinto en la versión 3.5: Se agregó el parámetro `numeric_owner`.

Distinto en la versión 3.6: El parámetro `path` acepta a `path-like object`.

Distinto en la versión 3.8.17: Se agregó el parámetro `filter`.

`TarFile.extract(member, path="", set_attrs=True, *, numeric_owner=False, filter=None)`

Extrae un miembro del archivo al directorio de trabajo actual utilizando su nombre completo. La información de su archivo se extrae con la mayor precisión posible. `member` puede ser un nombre de archivo o un objeto `TarInfo`. Puedes especificar un directorio diferente utilizando `path`. `path` puede ser un `path-like object`. Los atributos de archivo (dueño, fecha de modificación, modo) son establecidos a no ser que `set_attrs` sea falso.

The `numeric_owner` and `filter` arguments are the same as for `extractall()`.

Nota: El método `extract()` no cuida de varios problemas de extracción. En la mayoría de los casos deberías considerar utilizar el método `extractall()`.

Advertencia: Consulta la advertencia para `extractall()`.

Set `filter='data'` to prevent the most dangerous security issues, and read the [Extraction filters](#) section for details.

Distinto en la versión 3.2: Se agregó el parámetro `set_attrs`.

Distinto en la versión 3.5: Se agregó el parámetro *numeric_owner*.

Distinto en la versión 3.6: El parámetro *path* acepta a *path-like object*.

Distinto en la versión 3.8.17: Se agregó el parámetro *filter*.

`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 or a link, an *io.BufferedReader* object is returned. Otherwise, *None* is returned.

Distinto en la versión 3.3: Retorna un objeto *io.BufferedReader*.

`TarFile.errorlevel: int`

If *errorlevel* is 0, errors are ignored when using *TarFile.extract()* and *TarFile.extractall()*. Nevertheless, they appear as error messages in the debug output when *debug* is greater than 0. If 1 (the default), all *fatal* errors are raised as *OSError* or *FilterError* exceptions. If 2, all *non-fatal* errors are raised as *TarError* exceptions as well.

Some exceptions, e.g. ones caused by wrong argument types or data corruption, are always raised.

Custom *extraction filters* should raise *FilterError* for *fatal* errors and *ExtractError* for *non-fatal* ones.

Note that when an exception is raised, the archive may be partially extracted. It is the user's responsibility to clean up.

`TarFile.extraction_filter`

Nuevo en la versión 3.8.17.

The *extraction filter* used as a default for the *filter* argument of *extract()* and *extractall()*.

The attribute may be *None* or a callable. String names are not allowed for this attribute, unlike the *filter* argument to *extract()*.

If *extraction_filter* is *None* (the default), calling an extraction method without a *filter* argument will use the *fully_trusted* filter for compatibility with previous Python versions.

In Python 3.12+, leaving *extraction_filter=None* will emit a *DeprecationWarning*.

In Python 3.14+, leaving *extraction_filter=None* will cause extraction methods to use the *data* filter by default.

The attribute may be set on instances or overridden in subclasses. It also is possible to set it on the *TarFile* class itself to set a global default, although, since it affects all uses of *tarfile*, it is best practice to only do so in top-level applications or *site configuration*. To set a global default this way, a filter function needs to be wrapped in *staticmethod()* to prevent injection of a *self* argument.

`TarFile.add(name, arcname=None, recursive=True, *, filter=None)`

Añade el archivo *name* al archivo. *name* puede ser cualquier tipo de archivo (directorio, fifo, enlace simbólico, etc.). Si se proporciona, *arcname* especifica un nombre alternativo para el archivo en el archivo. Los directorios se agregan de forma recursiva de forma predeterminada. Esto se puede evitar estableciendo *recursive* con el valor de *False*. La recursividad agrega entradas en orden ordenado. Si se proporciona *filter*, debería ser una función que tome un argumento de objeto *TarInfo* y retorna el objeto *TarInfo* modificado. Si en cambio retorna *None*, el objeto *TarInfo* será excluido del archivo. Consulta *Ejemplos* para ver un ejemplo.

Distinto en la versión 3.2: Se agregó el parámetro *filter*.

Distinto en la versión 3.7: La recursividad agrega entradas en orden ordenado.

`TarFile.addfile(tarinfo, fileobj=None)`

Añade el objeto *TarInfo tarinfo* al archivo. Si se proporciona *fileobj*, debería ser un *binary file*, y los bytes de *tarinfo.size* se leen y se agregan al archivo. Puedes crear objetos *TarInfo* directamente o usando *gettartinfo()*.

`TarFile.gettarinfo(name=None, arcname=None, fileobj=None)`

Crea un objeto *TarInfo* a partir del resultado de *os.stat()* o equivalente en un archivo existente. El archivo es nombrado por *name* o especificado como un objeto *file object fileobj* con un descriptor de archivo. *name* puede ser un objeto *path-like object*. Si es proporcionado, *arcname* especifica un nombre alternativo para

el archivo en el archivo, de otra forma, el nombre es tomado del atributo *fileobj*'s *name*, o el argumento *name*. El nombre puede ser una cadena de texto.

Puedes modificar algunos de los atributos de la clase *TarInfo* antes de que lo añadas utilizando el método *addfile()*. Si el archivo objeto no es un archivo objeto ordinario posicionado al principio del archivo, atributos como *size* puede que necesiten modificaciones. Este es el caso para objetos como *GzipFile*. El atributo *name* también puede ser modificado, en cuyo caso *arcname* podría ser una cadena ficticia.

Distinto en la versión 3.6: El parámetro *name* acepta un objeto *path-like object*.

TarFile.close()

Cierra *TarFile*. En el modo de escritura, se añaden dos bloques de cero de finalización al archivo.

TarFile.pax_headers

Un diccionario que contiene pares de *pax global headers* en key-value.

13.6.2 Objetos TarInfo

Un objeto *TarInfo* representa un miembro en *TarFile*. Además de almacenar todos los atributos requeridos de un archivo (como tipo de archivo, tamaño, fecha, permisos, dueño, etc.), provee de algunos métodos útiles para determinar su tipo. No contiene los datos del archivo por sí mismo.

TarInfo objects are returned by *TarFile*'s methods *getmember()*, *getmembers()* and *gettarinfo()*.

Modifying the objects returned by *getmember()* or *getmembers()* will affect all subsequent operations on the archive. For cases where this is unwanted, you can use *copy.copy()* or call the *replace()* method to create a modified copy in one step.

Several attributes can be set to *None* to indicate that a piece of metadata is unused or unknown. Different *TarInfo* methods handle *None* differently:

- The *extract()* or *extractall()* methods will ignore the corresponding metadata, leaving it set to a default.
- *addfile()* will fail.
- *list()* will print a placeholder string.

Distinto en la versión 3.8.17: Added *replace()* and handling of *None*.

class tarfile.TarInfo (name='')

Crea un objeto *TarInfo*.

classmethod TarInfo.frombuf (buf, encoding, errors)

Crea y retorna un objeto *TarInfo*.

Lanza una excepción *HeaderError* si el buffer es inválido.

classmethod TarInfo.fromtarfile (tarfile)

Lee el siguiente miembro del objeto de la clase *TarFile* *tarfile* y lo retorna como un objeto *TarInfo*.

TarInfo.tobuf (format=DEFAULT_FORMAT, encoding=ENCODING, errors='surrogateescape')

Crear un *buffer* de cadena a partir de un objeto *TarInfo*. Para información sobre los argumentos consulta el constructor de la clase *TarFile*.

Distinto en la versión 3.2: Utiliza 'surrogateescape' como valor predeterminado del argumento *errors*.

Un objeto *TarInfo* tiene los siguientes atributos de dato públicos:

TarInfo.name: str

Nombre del archivo miembro.

TarInfo.size: int

Tamaño en bytes.

TarInfo.mtime: int | float

Time of last modification in seconds since the *epoch*, as in *os.stat_result.st_mtime*.

Distinto en la versión 3.8.17: Can be set to `None` for `extract()` and `extractall()`, causing extraction to skip applying this attribute.

TarInfo.mode: int

Permission bits, as for `os.chmod()`.

Distinto en la versión 3.8.17: Can be set to `None` for `extract()` and `extractall()`, causing extraction to skip applying this attribute.

TarInfo.type

Tipo de archivo. *type* es por lo general una de las siguientes constantes: `REGTYPE`, `AREGTYPE`, `LNKTYPE`, `SYMTYPE`, `DIRTYPE`, `FIFOTYPE`, `CONTTYPE`, `CHRTYPE`, `BLKTYPE`, `GNUTYPE_SPARSE`. Para determinar el tipo de un objeto *TarInfo* de forma más conveniente, utiliza los métodos `is*()` de abajo.

TarInfo.linkname: str

Nombre del archivo objetivo, el cual solo está presente en los objetos *TarInfo* de tipo `LNKTYPE` y del tipo `SYMTYPE`.

For symbolic links (`SYMTYPE`), the *linkname* is relative to the directory that contains the link. For hard links (`LNKTYPE`), the *linkname* is relative to the root of the archive.

TarInfo.uid: int

ID de usuario que originalmente almacenó este miembro.

Distinto en la versión 3.8.17: Can be set to `None` for `extract()` and `extractall()`, causing extraction to skip applying this attribute.

TarInfo.gid: int

ID de grupo del usuario que originalmente almacenó este miembro.

Distinto en la versión 3.8.17: Can be set to `None` for `extract()` and `extractall()`, causing extraction to skip applying this attribute.

TarInfo.uname: str

Nombre de usuario.

Distinto en la versión 3.8.17: Can be set to `None` for `extract()` and `extractall()`, causing extraction to skip applying this attribute.

TarInfo.gname: str

Nombre del grupo.

Distinto en la versión 3.8.17: Can be set to `None` for `extract()` and `extractall()`, causing extraction to skip applying this attribute.

TarInfo.pax_headers: dict

Un diccionario que contiene pares *key-value* de un encabezado *pax extended*.

TarInfo.replace(name=..., mtime=..., mode=..., linkname=..., uid=..., gid=..., uname=..., gname=..., deep=True)

Nuevo en la versión 3.8.17.

Return a *new* copy of the *TarInfo* object with the given attributes changed. For example, to return a *TarInfo* with the group name set to `'staff'`, use:

```
new_tarinfo = old_tarinfo.replace(gname='staff')
```

By default, a deep copy is made. If *deep* is false, the copy is shallow, i.e. `pax_headers` and any custom attributes are shared with the original *TarInfo* object.

Un objeto *TarInfo* también provee de algunos métodos de consulta convenientes:

TarInfo.isfile()

Retorna `True` si el objeto *Tarinfo* es un archivo regular.

TarInfo.isreg()

Igual que `isfile()`.

`TarInfo.isdir()`
 Retorna *True* si es un directorio.

`TarInfo.issym()`
 Retorna *True* si es un enlace simbólico.

`TarInfo.islnk()`
 Retorna *True* si es un enlace duro.

`TarInfo.ischr()`
 Retorna *True* si es un dispositivo de caracter.

`TarInfo.isblk()`
 Retorna *True* si es un dispositivo de bloque.

`TarInfo.isfifo()`
 Retorna *True* si es un FIFO.

`TarInfo.isdev()`
 Retorna *True* si es uno de los caracteres de dispositivo, dispositivo de bloque o FIFO.

13.6.3 Extraction filters

Nuevo en la versión 3.8.17.

The *tar* format is designed to capture all details of a UNIX-like filesystem, which makes it very powerful. Unfortunately, the features make it easy to create tar files that have unintended – and possibly malicious – effects when extracted. For example, extracting a tar file can overwrite arbitrary files in various ways (e.g. by using absolute paths, . . path components, or symlinks that affect later members).

In most cases, the full functionality is not needed. Therefore, *tarfile* supports extraction filters: a mechanism to limit functionality, and thus mitigate some of the security issues.

Ver también:

PEP 706 Contains further motivation and rationale behind the design.

The *filter* argument to `TarFile.extract()` or `extractall()` can be:

- the string `'fully_trusted'`: Honor all metadata as specified in the archive. Should be used if the user trusts the archive completely, or implements their own complex verification.
- the string `'tar'`: Honor most *tar*-specific features (i.e. features of UNIX-like filesystems), but block features that are very likely to be surprising or malicious. See `tar_filter()` for details.
- the string `'data'`: Ignore or block most features specific to UNIX-like filesystems. Intended for extracting cross-platform data archives. See `data_filter()` for details.
- `None` (default): Use `TarFile.extraction_filter`.

If that is also `None` (the default), the `'fully_trusted'` filter will be used (for compatibility with earlier versions of Python).

In Python 3.12, the default will emit a `DeprecationWarning`.

In Python 3.14, the `'data'` filter will become the default instead. It's possible to switch earlier; see `TarFile.extraction_filter`.

- A callable which will be called for each extracted member with a `TarInfo` describing the member and the destination path to where the archive is extracted (i.e. the same path is used for all members):

```
filter(/, member: TarInfo, path: str) -> TarInfo | None
```

The callable is called just before each member is extracted, so it can take the current state of the disk into account. It can:

- return a `TarInfo` object which will be used instead of the metadata in the archive, or

- return `None`, in which case the member will be skipped, or
- raise an exception to abort the operation or skip the member, depending on `errorlevel`. Note that when extraction is aborted, `extractall()` may leave the archive partially extracted. It does not attempt to clean up.

Default named filters

The pre-defined, named filters are available as functions, so they can be reused in custom filters:

`tarfile.fully_trusted_filter(/, member, path)`

Return *member* unchanged.

This implements the 'fully_trusted' filter.

`tarfile.tar_filter(/, member, path)`

Implements the 'tar' filter.

- Strip leading slashes (/ and `os.sep`) from filenames.
- *Refuse* to extract files with absolute paths (in case the name is absolute even after stripping slashes, e.g. `C:/foo` on Windows). This raises `AbsolutePathError`.
- *Refuse* to extract files whose absolute path (after following symlinks) would end up outside the destination. This raises `OutsideDestinationError`.
- Clear high mode bits (setuid, setgid, sticky) and group/other write bits (`S_IWOTH`).

Return the modified `TarInfo` member.

`tarfile.data_filter(/, member, path)`

Implements the 'data' filter. In addition to what `tar_filter` does:

- *Refuse* to extract links (hard or soft) that link to absolute paths, or ones that link outside the destination. This raises `AbsoluteLinkError` or `LinkOutsideDestinationError`.
Note that such files are refused even on platforms that do not support symbolic links.
- *Refuse* to extract device files (including pipes). This raises `SpecialFileError`.
- For regular files, including hard links:
 - Set the owner read and write permissions (`S_IWUSR`).
 - Remove the group & other executable permission (`S_IXOTH`) if the owner doesn't have it (`S_IXUSR`).
- For other files (directories), set `mode` to `None`, so that extraction methods skip applying permission bits.
- Set user and group info (`uid`, `gid`, `uname`, `gname`) to `None`, so that extraction methods skip setting it.

Return the modified `TarInfo` member.

Filter errors

When a filter refuses to extract a file, it will raise an appropriate exception, a subclass of `FilterError`. This will abort the extraction if `TarFile.errorlevel` is 1 or more. With `errorlevel=0` the error will be logged and the member will be skipped, but extraction will continue.

Hints for further verification

Even with `filter='data'`, `tarfile` is not suited for extracting untrusted files without prior inspection. Among other issues, the pre-defined filters do not prevent denial-of-service attacks. Users should do additional checks.

Here is an incomplete list of things to consider:

- Extract to a *new temporary directory* to prevent e.g. exploiting pre-existing links, and to make it easier to clean up after a failed extraction.
- When working with untrusted data, use external (e.g. OS-level) limits on disk, memory and CPU usage.
- Check filenames against an allow-list of characters (to filter out control characters, confusables, foreign path separators, etc.).
- Check that filenames have expected extensions (discouraging files that execute when you “click on them”, or extension-less files like Windows special device names).
- Limit the number of extracted files, total size of extracted data, filename length (including symlink length), and size of individual files.
- Check for files that would be shadowed on case-insensitive filesystems.

Also note that:

- Tar files may contain multiple versions of the same file. Later ones are expected to overwrite any earlier ones. This feature is crucial to allow updating tape archives, but can be abused maliciously.
- `tarfile` does not protect against issues with “live” data, e.g. an attacker tinkering with the destination (or source) directory while extraction (or archiving) is in progress.

Supporting older Python versions

Extraction filters were added to Python 3.12, and are backported to older versions as security updates. To check whether the feature is available, use e.g. `hasattr(tarfile, 'data_filter')` rather than checking the Python version.

The following examples show how to support Python versions with and without the feature. Note that setting `extraction_filter` will affect any subsequent operations.

- Fully trusted archive:

```
my_tarfile.extraction_filter = (lambda member, path: member)
my_tarfile.extractall()
```

- Use the 'data' filter if available, but revert to Python 3.11 behavior ('fully_trusted') if this feature is not available:

```
my_tarfile.extraction_filter = getattr(tarfile, 'data_filter',
                                       (lambda member, path: member))
my_tarfile.extractall()
```

- Use the 'data' filter; *fail* if it is not available:

```
my_tarfile.extractall(filter=tarfile.data_filter)
```

or:

```
my_tarfile.extraction_filter = tarfile.data_filter
my_tarfile.extractall()
```

- Use the 'data' filter; *warn* if it is not available:


```
if hasattr(tarfile, 'data_filter'):
    my_tarfile.extractall(filter='data')
else:
    # remove this when no longer needed
    warn_the_user('Extracting may be unsafe; consider updating Python')
    my_tarfile.extractall()
```

Stateful extraction filter example

While *tarfile*'s extraction methods take a simple *filter* callable, custom filters may be more complex objects with an internal state. It may be useful to write these as context managers, to be used like this:

```
with StatefulFilter() as filter_func:
    tar.extractall(path, filter=filter_func)
```

Such a filter can be written as, for example:

```
class StatefulFilter:
    def __init__(self):
        self.file_count = 0

    def __enter__(self):
        return self

    def __call__(self, member, path):
        self.file_count += 1
        return member

    def __exit__(self, *exc_info):
        print(f'{self.file_count} files extracted')
```

13.6.4 Interfaz de línea de comandos

Nuevo en la versión 3.4.

El módulo *tarfile* provee una interfaz de línea de comandos sencilla para interactuar con archivos tar.

Si quieres crear un nuevo archivo tar, especifica su nombre después de la opción *-c* y después lista el nombre de archivo(s) que deberían ser incluidos:

```
$ python -m tarfile -c monty.tar spam.txt eggs.txt
```

Proporcionar un directorio también es aceptable:

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

Si deseas extraer un archivo tar en el directorio actual, utiliza la opción *-e*:

```
$ python -m tarfile -e monty.tar
```

También puedes extraer un archivo tar en un directorio diferente pasando el nombre del directorio como parámetro:

```
$ python -m tarfile -e monty.tar other-dir/
```

Para obtener una lista de archivos dentro de un archivo tar, utiliza la opción *-l*:

```
$ python -m tarfile -l monty.tar
```


Opciones de línea de comandos

```
-l <tarfile>
--list <tarfile>
    Listar archivos en un tar.

-c <tarfile> <source1> ... <sourceN>
--create <tarfile> <source1> ... <sourceN>
    Crear archivo tar desde archivos fuente.

-e <tarfile> [<output_dir>]
--extract <tarfile> [<output_dir>]
    Extrae el archivo tar en el directorio actual si output_dir no es especificado.

-t <tarfile>
--test <tarfile>
    Probar si el archivo tar es valido o no.

-v, --verbose
    Output verbose.

--filter <filtername>
    Specifies the filter for --extract. See Extraction filters for details. Only string names are accepted (that is,
    fully_trusted, tar, and data).

    Nuevo en la versión 3.8.17.
```

13.6.5 Ejemplos

Cómo extraer un archivo tar entero al directorio de trabajo actual:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

Cómo extraer un subconjunto de un archivo tar con `TarFile.extractall()` utilizando una función generadora en lugar de una lista:

```
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()
```

Cómo crear un archivo tar sin comprimir desde una lista de nombres de archivo:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

El mismo ejemplo utilizando la declaración `with`:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

Cómo leer un archivo tar comprimido con gzip y desplegar un poco de información del miembro:

```
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 ", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

Cómo crear un archivo y reiniciar la información del usuario usando el parámetro *filter* en `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()
```

13.6.6 Formatos tar con soporte

Hay tres formatos tar que puede ser creados con el módulo `tarfile`:

- El formato (`USTAR_FORMAT`) POSIX.1-1988 `ustar`. Admite nombres de archivos de hasta una longitud de 256 caracteres y nombres de enlace de hasta 100 caracteres. El tamaño máximo de archivo es de 8 GiB. Este es un formato viejo y limitado pero con amplio soporte.
- El formato GNU tar (`GNU_FORMAT`). Admite nombres de archivo largos y nombres de enlace, archivos más grandes que 8 GiB de tamaño y archivos dispersos. Es el estándar *de facto* en sistemas GNU/Linux. `tarfile` admite de forma completa las extensiones de GNU tar para nombres largos, el soporte para archivos dispersos es de solo lectura.
- El formato pax POSIX.1-2001 (`PAX_FORMAT`). Es el formato más flexible prácticamente sin límites. Admite nombres de archivo largos y nombres de enlaces, archivos grandes y almacena nombres de ruta de forma portátil. Las implementaciones modernas de tar, incluyendo GNU tar, `bsdtar/libarchive` y `star`, son totalmente compatibles con las características extendidas `pax`; Es posible que algunas bibliotecas antiguas o no mantenidas no lo hagan, pero deberían tratar los archivos `pax` como si estuvieran en el formato `ustar` compatible universalmente. Es el formato predeterminado actual para archivos nuevos.

Extiende el formato existente `ustar` con cabeceras adicionales para información que no puede ser almacenada de otra manera. Hay dos variaciones de encabezados `pax`: Los encabezados extendidos solo afectan al encabezado del archivo posterior, los encabezados globales son válidos para el archivo entero y afectan todos los siguientes archivos. Todos los datos en un encabezado `pax` son codificados en `UTF-8` por razones de portabilidad.

Existen más variantes del formato tar que puede ser leídas, pero no creadas:

- El antiguo formato V7. Este es el primer formato tar de *Unix Seventh Edition*, almacena solo archivos y directorios normales. Los nombres no deben tener más de 100 caracteres, no hay información de nombre de usuario/grupo disponible. Algunos archivos tienen sumas de comprobación de encabezado mal calculadas en el caso de campos con caracteres que no son ASCII.

- El formato extendido tar de SunOS. Este formato es una variante del formato POSIX.1-2001 pax, pero no es compatible.

13.6.7 Problemas Unicode

El formato tar fue originalmente concebido para realizar respaldos en unidades de cinta con el objetivo principal de preservar la información del sistema de archivos. Hoy en día los archivos tar son comúnmente utilizados para distribución de archivos e intercambio de archivos sobre redes. Un problema del formato original (el cual es la base de todos los demás formatos) es que no hay concepto de soporte para diferentes codificaciones de caracteres. Por ejemplo, un archivo tar ordinario creado en un sistema *UTF-8* no puede ser leído correctamente en un sistema *Latin-1* si este contiene caracteres de tipo no ASCII. Los meta-datos textuales (como nombres de archivos, nombres de enlace, nombres de usuario/grupo) aparecerán dañados. Desafortunadamente, no hay manera de detectar de forma automática la codificación de un archivo. El formato *pax* fue diseñado para resolver este problema. Almacena los metadatos no ASCII usando una codificación de caracteres universal *UTF-8*.

Los detalles de la conversión de caracteres en el módulo *tarfile* son controlados por los argumentos de palabra clave *encoding* y *errors* de la clase *TarFile*.

encoding define la codificación de caracteres a utilizar para los metadatos del archivo. El valor predeterminado es *sys.getfilesystemencoding()* o 'ascii' como una alternativa. Dependiendo de si el archivo se lee o escribe, los metadatos deben decodificarse o codificarse. Si el *encoding* no se configura correctamente, esta conversión puede fallar.

El argumento *errors* define como van a ser tratados los caracteres que no pueden ser transformados. Los valores admitidos están listados en la sección *Manejadores de errores*. El esquema predeterminado es 'surrogateescape' el cual Python también utiliza para sus llamadas al sistema de archivos. Consulta *Nombres de archivos, argumentos de la línea de comandos y variables de entorno*.

Para archivos *PAX_FORMAT* (el formato default), el *encoding* generalmente no es necesario porque todos los metadatos son almacenados utilizando *UTF-8*, el *encoding* solo es utilizado en aquellos casos cuando las cabeceras binarias PAX son decodificadas o cuando se almacenan cadenas de texto con caracteres sustitutos.

Los módulos descritos en este capítulo analizan varios formatos de archivo que no son lenguajes de marcado y no están relacionados con el correo electrónico.

14.1 `csv` — Lectura y escritura de archivos CSV

Código fuente: [Lib/csv.py](#)

El tan llamado CSV (Valores Separados por Comas) es el formato más común de importación y exportación de hojas de cálculo y bases de datos. El formato CSV se utilizó durante muchos años antes de intentar describir el formato de manera estandarizada en **RFC 4180**. La falta de un estándar bien definido significa que a veces existen pequeñas diferencias en la información producida y consumida por diferentes aplicaciones. Estas diferencias pueden ser molestas al momento de procesar archivos CSV desde múltiples fuentes. Aún así, aunque los delimitadores y separadores varíen, el formato general es lo suficientemente similar como para que sea posible un sólo módulo que puede manipular tal información eficientemente, escondiendo los detalles de lectura y escritura de datos del programador.

El módulo `csv` implementa clases para leer y escribir datos tabulares en formato CSV. Permite a los programadores decir, «escribe estos datos en el formato preferido por Excel», o «lee datos de este archivo que fue generado por Excel», sin conocer los detalles precisos del formato CSV usado por Excel. Los programadores también pueden describir los formatos CSV entendidos por otras aplicaciones o definir sus propios formatos CSV para fines particulares.

Los objetos `reader` y `writer` del módulo `csv` leen y escriben secuencias. Los programadores también pueden leer y escribir datos en forma de diccionario usando las clases `DictReader` y `DictWriter`.

Ver también:

PEP 305 - API de archivo CSV La Propuesta de Mejora de Python (PEP, por sus siglas en inglés) que propone esta adición a Python.

14.1.1 Contenidos del módulo

El módulo `csv` define las siguientes funciones:

`csv.reader (csvfile, dialect='excel', **fmtparams)`

Retorna un objeto *reader* que iterará sobre las líneas del *csvfile* proporcionado. *csvfile* puede ser cualquier objeto que soporte el protocolo *iterator* y retorna una cadena de caracteres siempre que su método `__next__()` sea llamado — tanto *objetos de archivo* como objetos de lista son adecuados. Si *csvfile* es un objeto de archivo, debería ser abierto con `newline=''`.¹ Se puede proporcionar un parámetro opcional *dialect*, el cual se utiliza para definir un conjunto de parámetros específicos para un dialecto de CSV particular. Puede ser una instancia de una subclase de la clase *Dialect* o una de las cadenas retornadas por la función `list_dialects()`. Los otros argumentos nombrados opcionales *fmtparams* pueden ser dados para sustituir parámetros de formato individuales del dialecto actual. Para detalles completos sobre el dialecto y los parámetros de formato, vea la sección *Dialectos y parámetros de formato*.

Cada fila leída del archivo csv es retornada como una lista de cadenas. No se realiza conversión automática de tipo de datos a menos que la opción de formato `QUOTE_NONNUMERIC` esté especificada (en ese caso los campos no citados son transformados en flotantes).

Un pequeño ejemplo de uso:

```
>>> import csv
>>> with open('eggs.csv', newline='') 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
```

`csv.writer (csvfile, dialect='excel', **fmtparams)`

Retorna un objeto *writer* responsable de convertir los datos del usuario a cadenas de caracteres delimitadas en el objeto proporcionado. *csvfile* puede ser cualquier objeto con un método `write()`. Si *csvfile* es un objeto de archivo, debería ser abierto con `newline=''`.¹ Se puede proporcionar un parámetro opcional *dialect*, el cual se utiliza para definir un conjunto de parámetros específicos para un dialecto de CSV particular. Puede ser una instancia de una subclase de la clase *Dialect* o una de las cadenas retornadas por la función `list_dialects()`. Los otros argumentos por palabra clave opcionales *fmtparams* pueden ser dados para sustituir parámetros de formato individuales del dialecto actual. Para detalles completos sobre el dialecto y los parámetros de formato, vea la sección *Dialectos y parámetros de formato*. Para facilitar la interacción con módulos que implementan la API DB, el valor `None` es escrito como una cadena vacía. A pesar de que esta no es una transformación reversible, facilita el volcado de valores de datos SQL NULL en archivos CSV sin tener que preprocesar los datos retornados por una llamada a `cursor.fetch*`. Todos los demás datos que no son cadenas son convertidos con la función `str()` antes de ser escritos.

Un pequeño ejemplo de uso:

```
import csv
with open('eggs.csv', 'w', newline='') 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]])`

Asocia el *dialect* con un *name*. *name* debe ser una cadena. El dialecto puede ser especificado ya sea pasando una subclase de *Dialect*, o con los argumentos de palabras clave *fmtparams*, o ambos, con los argumentos de palabras clave sobrescribiendo los parámetros del dialecto. Para más detalles sobre el dialecto y los parámetros de formato, vea la sección *Dialectos y parámetros de formato*.

¹ Si `newline=''` no es especificado, las nuevas líneas dentro de los campos citados no serán interpretadas correctamente y, en plataformas que utilicen finales de línea `\r\n` en la escritura, se añadirá un `\r` extra. Siempre debería ser seguro especificar `newline=''`, ya que el módulo `csv` realiza su propio manejo de nuevas líneas (*universal*).

`csv.unregister_dialect(name)`

Borra el dialecto asociado a *name* del registro de dialectos. Un *Error* es lanzado si *name* no está registrado como el nombre de un dialecto.

`csv.get_dialect(name)`

Retorna el dialecto asociado a *name*. Un *Error* es lanzado si *name* no está registrado como el nombre de un dialecto. Esta función retorna un objeto *Dialect* inmutable.

`csv.list_dialects()`

Retorna los nombres de todos los dialectos registrados.

`csv.field_size_limit([new_limit])`

Retorna el tamaño máximo de campo permitido actualmente por el intérprete. Si *new_limit* es dado, este se convierte en el nuevo límite.

El módulo *csv* define las siguientes clases:

class `csv.DictReader(f, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kws)`

Crea un objeto que opera como un *reader* común, pero mapea la información en cada fila a un *dict* cuyas claves son provistas en el parámetro opcional *fieldnames*.

El parámetro *fieldnames* es una *sequence*. Si se omite *fieldnames*, los valores en la primera fila del archivo *f* serán usados como nombres de campo. Independientemente de como se determinen los nombres de campo, el diccionario preserva su orden original.

Si una fila tiene más campos que nombres de campo, los datos restantes son colocados en una lista y guardados con el nombre de campo especificado por *restkey* (que por defecto es *None*). Si una fila que no esta en blanco tiene menos campos que nombres de campo, los valores faltantes son rellenados con el valor de *restval* (que por defecto es *None*).

Todos los demás argumentos de palabra clave u opcionales son pasados a la instancia subyacente de *reader*.

Distinto en la versión 3.6: Returned rows are now of type *OrderedDict*.

Distinto en la versión 3.8: Las filas retornadas son ahora de tipo *dict*.

Un pequeño ejemplo de uso:

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
{'first_name': 'John', 'last_name': 'Cleese'}
```

class `csv.DictWriter(f, fieldnames, restval="", extrasaction='raise', dialect='excel', *args, **kws)`

Crea un objeto que opera como un *writer* común, pero mapea diccionarios a filas de salida. El parámetro *fieldnames* es una *secuencia* de claves que identifican el orden en el cual los valores en el diccionario pasados al método *writerow()* son escritos en el archivo *f*. El parámetro opcional *restval* especifica el valor a ser escrito si al diccionario le falta una clave en *fieldnames*. Si el diccionario pasado al método *writerow()* contiene una clave no encontrada en *fieldnames*, el parámetro opcional *extrasaction* indica que acción ejecutar. Si esta configurado en *'raise'*, el valor por defecto, es lanzado un *ValueError*. Si esta configurado con *'ignore'*, los valores extra en el diccionario son ignorados. Cualquier otro argumento de palabra clave u opcional es pasado a la instancia subyacente de *reader*.

Nótese que a diferencia de la clase *DictReader*, el parámetro *fieldnames* de la clase *DictWriter* no es opcional.

Un pequeño ejemplo de uso:

```
import csv

with open('names.csv', 'w', newline='') 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

La clase *Dialect* es una clase contenedora basada principalmente en sus atributos, que son usados para definir los parámetros para una instancia *reader* o *writer* específica.

class csv.excel

La clase *excel* define las propiedades usuales de un archivo CSV generado por Excel. Esta registrada con el nombre de dialecto 'excel'.

class csv.excel_tab

La clase *excel_tab* define las propiedades usuales de un archivo delimitado por tabulaciones generado por Excel. Esta registrada con el nombre de dialecto 'excel-tab'.

class csv.unix_dialect

La clase *unix_dialect* define las propiedades usuales de un archivo CSV generado en sistemas UNIX, es decir, usando '\n' como terminador de línea y citando todos los campos. Esta registrada con el nombre de dialecto 'unix'.

Nuevo en la versión 3.2.

class csv.Sniffer

La clase *Sniffer* es usada para deducir el formato de un archivo CSV.

La clase *Sniffer* provee 2 métodos:

sniff (*sample*, *delimiters=None*)

Analiza la muestra dada y retorna una subclase *Dialect* reflejando los parámetros encontrados. Si el parámetro opcional *delimiters* es dado, este será interpretado como una cadena que contiene posibles caracteres delimitadores válidos.

has_header (*sample*)

Analiza el texto de muestra (presumiblemente en formato CSV) y retorna *True* si la primera fila parece ser una serie de encabezados de columna.

Un ejemplo para el uso de *Sniffer*:

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

El módulo *csv* define las siguientes constantes:

csv.QUOTE_ALL

Ordena a los objetos *writer* citar todos los campos.

csv.QUOTE_MINIMAL

Ordena a los objetos *writer* citar solo aquellos campos que contengan caracteres especiales como por ejemplo *delimiter*, **quotechar* o cualquiera de los caracteres en *lineterminator*.

csv.QUOTE_NONNUMERIC

Ordena a los objetos *writer* citar todos los campos no numéricos.

Ordena al *reader* a convertir todos los campos no citados al tipo *float*.

csv.QUOTE_NONE

Ordena a los objetos *writer* nunca citar campos. Cuando el *delimiter* actual aparece en los datos de salida es precedido por el carácter *escapechar* actual. Si *escapechar* no está definido, el *writer* lanzará *Error* si cualquier carácter que requiere escaparse es encontrado.

Ordena a *reader* no ejecutar un procesamiento especial de caracteres citados.

El módulo *csv* define la siguiente excepción:

exception csv.Error

Lanzada por cualquiera de las funciones cuando se detecta un error.

14.1.2 Dialectos y parámetros de formato

Para facilitar la especificación del formato de registros de entrada y salida, los parámetros específicos de formateo son agrupados en dialectos. Un dialecto es una subclase de la clase *Dialect* con un conjunto de métodos específicos y un solo método *validate()*. Cuando se crean objetos *reader* o *writer*, el programador puede especificar una cadena de caracteres o una subclase de la clase *Dialect* como el parámetro de dialecto. Además de, o en vez de, el parámetro *dialect*, el programador también puede especificar parámetros de formateo individuales, con los mismos nombres que los atributos definidos debajo para la clase *Dialect*.

Los dialectos soportan los siguientes atributos:

Dialect.delimiter

Una cadena de un solo carácter usada para separar campos. Por defecto es `' '`.

Dialect.doublequote

Controla como las instancias de *quotechar* que aparecen dentro de un campo deben estar citadas. Cuando es *True*, el carácter es duplicado. Cuando es *False*, el *escapechar* es usado como un prefijo al *quotechar*. Por defecto es *True*.

En la salida, si el *doublequote* es *False* y el *escapechar* no está configurado, un *Error* es lanzado si se encuentra un *quotechar* en algún campo.

Dialect.escapechar

Una cadena de un solo carácter usada por el *writer* para escapar al *delimiter* si *quoting* está configurado como *QUOTE_NONE* y al *quotechar* si *doublequote* es *False*. En la lectura, el *escapechar* elimina cualquier significado especial del siguiente carácter. Por defecto es *None*, lo que deshabilita el escape.

Dialect.lineterminator

La cadena de caracteres usada para terminar las líneas producidas por *writer*. Por defecto es `'\r\n'`.

Nota: El *reader* está codificado para reconocer tanto `'\r'` o `'\n'` como final de línea, e ignora *lineterminator*. Este comportamiento puede cambiar en el futuro.

Dialect.quotechar

Una cadena de un solo carácter usada para citar campos que contienen caracteres especiales, como lo son *delimiter* o *quotechar*, o que contienen caracteres de nueva línea. Por defecto es `'\"'`.

Dialect.quoting

Controla cuando las comillas deberían ser generadas por el *writer* y ser reconocidas por el *reader*. Puede tomar cualquiera de las constantes *QUOTE_** (ver sección *Contenidos del módulo*) y por defecto es *QUOTE_MINIMAL*.

Dialect.skipinitialspace

Cuando es *True*, el espacio en blanco que sigue después del *delimiter* es ignorado. Por defecto es *False*.

Dialect.strict

Cuando es *True*, lanza una excepción *Error* sobre una mala entrada CSV. Por defecto es *False*.

14.1.3 Objetos *Reader*

Los objetos *reader* (instancias de *DictReader* y objetos retornados por la función *reader()*) contienen los siguientes métodos públicos:

`csvreader.__next__()`

Retorna la siguiente fila del objeto iterable del *reader* como una lista (si el objeto fue retornado de *reader()*) o como un diccionario (si es una instancia de *DictReader*), analizada acorde al dialecto actual. Normalmente deberías llamarlo como `next(reader)`.

Los objetos *reader* contienen los siguientes atributos públicos:

`csvreader.dialect`

Una descripción de sólo lectura del dialecto en uso por el intérprete.

`csvreader.line_num`

El número de líneas leídas del iterador fuente. Esto no es lo mismo que el número de registros retornado, ya que los registros pueden abarcar múltiples líneas.

Los objetos *DictReader* tienen los siguientes atributos públicos:

`csvreader.fieldnames`

Si no son pasados como parámetros cuando se crea el objeto, este atributo es inicializado en el primer acceso o cuando es leído el primer registro del archivo.

14.1.4 Objetos *Writer*

Los objetos *Writer* (instancias de *DictWriter* y objetos retornados por la función *writer()*) contienen los siguientes métodos públicos. Una *row* debe ser un iterable de cadenas de caracteres o números para objetos *Writer* y un diccionario que mapea nombres de campo a cadenas de caracteres o números (pasándolos primero a través de *str()*) para objetos *DictWriter*. Note que los números complejos se escriben rodeados de paréntesis. Esto puede causar algunos problemas para otros programas que leen archivos CSV (asumiendo que soportan números complejos).

`csvwriter.writerow(row)`

Escribe el parámetro *row* al objeto de archivo del *writer*, formateado según el dialecto actual. Retorna el valor de retorno de la llamada al método *write* del objeto de archivo subyacente.

Distinto en la versión 3.5: Agregado soporte para iterables.

`csvwriter.writerows(rows)`

Escribe todos los elementos en *rows* (un iterable de objetos *row* como se describe anteriormente) al objeto de archivo del *writer*, formateados según el dialecto actual.

Los objetos *writer* contienen los siguientes atributos públicos:

`csvwriter.dialect`

Una descripción de solo lectura del dialecto en uso por el *writer*.

Los objetos *DictWriter* contienen los siguientes métodos públicos:

`DictWriter.writeheader()`

Escribe una fila con los nombres de los campos (como se especifica en el constructor) al objeto de archivo del *writer*, formateada según el dialecto actual. Retorna el valor de retorno de la llamada a `csvwriter.writerow()` usada internamente.

Nuevo en la versión 3.2.

Distinto en la versión 3.8: *writeheader()* ahora también retorna el valor retornado por el método `csvwriter.writerow()` que usa internamente.

14.1.5 Ejemplos

El ejemplo más simple de lectura de un archivo CSV:

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Lectura de un archivo con un formato alternativo:

```
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

El correspondiente ejemplo de escritura más simple es:

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

Ya que `open()` es usado para abrir un archivo CSV para lectura, el archivo será decodificado por defecto en unicode usando la codificación por defecto del sistema (ver `locale.getpreferredencoding()`). Para decodificar un archivo usando una codificación diferente, usa el argumento `encoding` de `open`:

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Lo mismo aplica a escribir en algo diferente a la codificación por defecto del sistema: especifique el argumento de codificación cuando abra el archivo de salida.

Registrando un nuevo dialecto:

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

Un uso ligeramente más avanzado del `reader` — captura y reporte de errores:

```
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit('file {}, line {}: {}'.format(filename, reader.line_num, e))
```

Y a pesar de que el módulo no soporta el análisis de cadenas de caracteres directamente, puede ser realizado fácilmente:

```
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

Notas al pie

14.2 configparser — *Parser* para archivos de configuración

Código fuente: [Lib/configparser.py](#)

Este módulo provee la clase `ConfigParser`, la cual implementa un lenguaje básico de configuración que proporciona una estructura similar a la encontrada en los archivos INI de Microsoft Windows. Puedes utilizarla para escribir programas Python que los usuarios finales puedan personalizar con facilidad.

Nota: Esta biblioteca *no* interpreta o escribe los prefijos valor-tipo usados en la versión extendida de la sintaxis INI, utilizada en el registro de Windows.

Ver también:

Módulo `shlex` Soporta la creación de un mini-lenguaje parecido a shell de Unix, que puede utilizarse como formato alternativo para archivos de configuración de aplicaciones.

Módulo `json` El módulo json implementa un subconjunto de la sintaxis de Javascript, que también puede utilizarse para este propósito.

14.2.1 Inicio Rápido

Tomemos un archivo de configuración muy básico, el cual luce así:

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[bitbucket.org]
User = hg

[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

La estructura de los archivos INI es descrita *en la siguiente sección*. En esencia, el archivo consiste de secciones, cada una de las cuales contiene claves con valores. Las clases `configparser` pueden leer y escribir dichos archivos. Comencemos creando el anterior archivo de configuración de forma programática.

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                    'Compression': 'yes',
...                    'CompressionLevel': '9'}
>>> config['bitbucket.org'] = {}
>>> config['bitbucket.org']['User'] = 'hg'
>>> config['topsecret.server.com'] = {}
>>> topsecret = config['topsecret.server.com']
>>> topsecret['Port'] = '50022'      # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...     config.write(configfile)
... 
```

Como puedes ver, podemos tratar al *config parser* como a un diccionario. Existen diferencias, *descritas posteriormente*, pero su comportamiento es muy parecido al que esperarías de un diccionario.

Ahora que hemos creado y guardado el archivo de configuración, vamos a releerlo y analizar los datos que contiene.

```
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['bitbucket.org', 'topsecret.server.com']
>>> 'bitbucket.org' in config
True
>>> 'bytebong.com' in config
False
>>> config['bitbucket.org']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.com']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['bitbucket.org']:
...     print(key)
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['bitbucket.org']['ForwardX11']
'yes'
```

Como podemos apreciar, la API es muy clara. La única “porción de magia” está en la sección `DEFAULT`, la cual proporciona los valores por defecto para todas las demás secciones¹. Observe también que las claves de las secciones son insensibles a mayúsculas y minúsculas, pero se almacenan en minúscula¹.

14.2.2 Tipos de Datos Soportados

Los *config parsers* no especulan respecto a los tipos de datos de los valores en los archivos de configuración, siempre los almacenan internamente como cadenas de caracteres. Esto significa que si necesitas otros tipos de datos, deberás hacer la conversión por ti mismo:

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

Dado que esta tarea es muy común, los *config parsers* proporcionan una amplia variedad de útiles métodos de lectura (*getter*) que manejan enteros, números de punto flotante y booleanos. Este último es el más interesante, porque puede no ser correcto pasar simplemente el valor a `bool()`, ya que `bool('False')` resultará en `True`. Por esta razón los *config parsers* proporcionan también el método `getboolean()`. Este método es insensible a mayúsculas y minúsculas, y reconoce como valores booleanos a los valores `'yes'/'no'`, `'on'/'off'`, `'true'/'false'` and `'1'/'0'`¹. Por ejemplo:

¹ Los *config parsers* permiten una gran personalización. Si estás interesado en modificar el comportamiento descrito por la referencia de la nota al pie, consulta la sección *Customizing Parser Behaviour*.

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['bitbucket.org'].getboolean('ForwardX11')
True
>>> config.getboolean('bitbucket.org', 'Compression')
True
```

Además de `getboolean()`, los `config parsers` también proporcionan los métodos equivalentes `getint()` y `getfloat()`. Puedes registrar tus propios conversores, y personalizar los que se proporcionan.¹

14.2.3 Valores de contingencia

Similar a un diccionario, puedes utilizar el método `get()` de una sección para especificar valores de contingencia (*fallback values*):

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

Por favor, fíjate que los valores por defecto tienen prioridad sobre los valores de contingencia (*fallback*). Así, en nuestro ejemplo, la clave `'CompressionLevel'` sólo fue especificada en la sección `'DEFAULT'`. Si tratamos de obtener su valor de la sección `'topsecret.server.com'`, obtendremos siempre el valor por defecto, incluso si especificamos un valor de contingencia:

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

Otra cuestión que hay que tener en cuenta, es que el método a nivel analizador (*parser*) `get()` proporciona una interfaz personalizada, más compleja, que se mantiene por compatibilidad con versiones anteriores. Cuando se utiliza este método, se puede proporcionar un valor de contingencia mediante el argumento de sólo-palabra clave (*keyword-only*) `fallback`:

```
>>> config.get('bitbucket.org', 'monster',
...           fallback='No such things as monsters')
'No such things as monsters'
```

El mismo argumento `fallback` puede utilizarse con los métodos `getint()`, `getfloat()` y `getboolean()`, por ejemplo:

```
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

14.2.4 Estructura Soportada para el Archivo INI

Un archivo de configuración consiste de secciones, cada una iniciada por una cabecera `[section]`, seguida por registros clave-valor, separados por una cadena de caracteres específica (= ó : por defecto¹). De forma predeterminada, los nombres de sección son sensibles a mayúsculas y minúsculas pero las claves no¹. Los espacios al inicio y final de las claves y valores son eliminados. Los valores pueden ser omitidos, en cuyo caso el delimitador del dato clave-valor puede ser omitido también. Los valores pueden ocupar varias líneas, siempre y cuando las líneas tengan una indentación mayor que la primera. Dependiendo del modo del *parser*, las líneas en blanco pueden tratarse como parte de un valor multilínea o ser ignoradas.

Los archivos de configuración pueden incluir comentarios, con caracteres específicos como prefijos (`#` y `;` por defecto¹). Los comentarios pueden ocupar su propia línea, posiblemente indentada.¹

Por ejemplo:

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
      I sleep all night and I work all day

[No Values]
key_without_value
empty string value here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

[Sections Can Be Indented]
    can_values_be_as_well = True
    does_that_mean_anything_special = False
    purpose = formatting for readability
    multiline_values = are
        handled just fine as
        long as they are indented
        deeper than the first line
        of a value
    # Did I mention we can indent comments, too?
```

14.2.5 Interpolación de valores

En el nivel superior de su funcionalidad central, *ConfigParser* soporta la interpolación. Esto significa que los valores pueden ser preprocesados, antes de ser retornados por los llamados a `get()`.

class configparser.BasicInterpolation

Es la implementación por defecto que utiliza *ConfigParser*. Permite valores que contengan cadenas de formato, que hacen referencia a otros valores en la misma sección o a valores presentes en la sección especial *default*¹. Valores por defecto adicionales pueden ser proporcionados en la inicialización.

Por ejemplo:

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures

[Escape]
gain: 80%% # use a %% to escape the % sign (% is the only character that
↳needs to be escaped)
```

En el ejemplo anterior, *ConfigParser* con la *interpolación* establecida a `BasicInterpolation()` puede resolver `%(home_dir)s` al valor `home_dir` (`/Users` en este caso). En efecto, `%(my_dir)s` podría resolverse como `/Users/lumberjack`. Todas las interpolaciones son realizadas bajo demanda, de modo que las claves utilizadas en la cadena de referencias no requieren un orden específico en el archivo de configuración.

Con *interpolation* establecida al valor `None`, el *parser* retornará simplemente `%(my_dir)s/Pictures` como el valor de `my_pictures` y `%(home_dir)s/lumberjack` como el valor de `my_dir`.

class configparser.ExtendedInterpolation

Es un gestor alternativo para la interpolación, que implementa una sintaxis más avanzada; es utilizado, por ejemplo, en `zc.buildout`. La interpolación es extendida al utilizar `${section:option}`, a fin de especificar un valor que proviene de una sección externa. La interpolación puede cubrir múltiples niveles. Por conveniencia, si la parte `section:` es omitida, la interpolación utilizará la sección actual (y posiblemente los valores por defecto establecidos en la sección especial).

Por ejemplo, la configuración indicada anteriormente, con interpolación básica, luciría de la siguiente manera utilizando interpolación extendida:

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures

[Escape]
cost: $$80 # use a $$ to escape the $ sign ($ is the only character that
↳needs to be escaped)
```

También pueden buscarse valores en otras secciones:

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
```

(continué en la próxima página)

(proviene de la página anterior)

```

nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}

```

14.2.6 Acceso por Protocolo de Mapeo

Nuevo en la versión 3.2.

El acceso por protocolo de mapeo es un nombre genérico asignado a la funcionalidad que permite el uso de objetos personalizados como si fuesen diccionarios. En el caso de `configparser`, la implementación de la interfaz de mapeo utiliza la notación `parser['section']['option']`.

En particular, `parser['section']` retorna un proxy para los datos de la sección en el `parser`. Esto significa que los valores no son copiados, sino que son tomados del `parser` original sobre la marcha.

Los objetos de `configparser` se comportan de forma tan similar a un diccionario como se puede. La interfaz de mapeo es completa y se ciñe a la `MutableMapping` ABC. Sin embargo, existen unas pequeñas diferencias que deben tomarse en cuenta:

- Por defecto, todas las claves de las secciones se pueden acceder de una forma insensible a mayúsculas y minúsculas¹. Ej. `for option in parser["section"]` entrega solamente nombres de claves de opción que han pasado por `optionxform`. Esto significa, claves en minúscula por defecto. A la vez, para una sección que contiene la clave 'a', ambas expresiones retornan `True`:

```

"a" in parser["section"]
"A" in parser["section"]

```

- Todas las secciones también incluyen valores `DEFAULTSECT`, lo que significa que `.clear()` en una sección puede no significar que esta quede vacía de forma visible. Ello debido a que los valores por defecto no pueden ser borrados de la sección (ya que técnicamente no están allí). Si fueron redefinidas en la sección, su borrado ocasiona que los valores por defecto sean visibles de nuevo. Cualquier intento de borrar un valor por defecto ocasiona una excepción `KeyError`.
- `DEFAULTSECT` no puede ser eliminado del `parser`:
 - el intento de borrarlo lanza una excepción `ValueError`,
 - `parser.clear()` lo deja intacto,
 - `parser.popitem()` nunca lo retorna.
- `parser.get(section, option, **kwargs)` - el segundo argumento **no** es un valor de contingencia. Observe, sin embargo, que los métodos `get()` a nivel de sección son compatibles tanto con el protocolo de mapeo como con la API clásica de `configparser`.
- `parser.items()` es compatible con el protocolo de mapeo (retorna una lista de pares `section_name, section_proxy` que estén incluidos en `DEFAULTSECT`). Sin embargo, este método también puede ser invocado con los argumentos: `parser.items(section, raw, vars)`. Esta última llamada retorna una lista de pares `option, value` para una `section` específica, con todas las interpolaciones expandidas (a menos que se especifique `raw=True`).

El protocolo de mapeo se implementa en el nivel superior de la actual API heredada, de modo que esas subclases que sobre-escriben la interfaz original aún deberían hacer funcionar el mapeo como se esperaría.

14.2.7 Personalizando el Comportamiento del Parser

Existen casi tantas variantes del formato INI como aplicaciones que lo usen. `configparser` llega muy lejos al proporcionar soporte al mayor conjunto sensible de estilos de INI disponibles. La funcionalidad predeterminada es impuesta principalmente por antecedentes históricos y es muy probable que quieras personalizar algunas de las funcionalidades.

La forma más común para modificar cómo funciona un *config parser* específico consiste en el uso de las opciones de `__init__()`:

- `defaults`, valor por defecto: `None`

Esta opción acepta un diccionario de pares clave-valor, que debe ser colocado inicialmente en la sección `DEFAULT`. De este modo se obtiene una manera elegante de apoyar los archivos de configuración concisos, que no especifican valores que sean los mismos documentados por defecto.

Consejo: si quieres especificar valores por defecto para una sección específica, usa `read_dict()` antes de leer el archivo real.

- `dict_type`, valor por defecto: `dict`

Esta opción tiene un gran impacto en cómo funcionará el protocolo de mapeo, y cómo lucirán los archivos de configuración a escribir. Con el diccionario estándar, cada sección se almacena en el orden en que se añadieron al *parser*. Lo mismo ocurre con las opciones dentro de las secciones.

Un tipo alternativo de diccionario puede ser utilizado, por ejemplo, para ordenar las secciones y opciones en un modo a posteriori (*write-back*).

Por favor, tome en cuenta que: existen formas de agregar un par clave-valor en una sola operación. Cuando se utiliza un diccionario común en tales operaciones, el orden de las claves será el empleado en la creación. Por ejemplo:

```
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
...                               'key2': 'value2',
...                               'key3': 'value3'},
...                  'section2': {'keyA': 'valueA',
...                               'keyB': 'valueB',
...                               'keyC': 'valueC'},
...                  'section3': {'foo': 'x',
...                               'bar': 'y',
...                               'baz': 'z'}})
>>> parser.sections()
['section1', 'section2', 'section3']
>>> [option for option in parser['section3']]
['foo', 'bar', 'baz']
```

- `allow_no_value`, valor por defecto: `False`

Se sabe que algunos archivos de configuración incluyen elementos sin indicar un valor, aunque cumplan con la sintaxis soportada por `configparser` en todo lo demás. El parámetro `allow_no_value` le indica al constructor que tales valores deben ser aceptados:

```
>>> import configparser

>>> sample_config = """
... [mysqld]
...   user = mysql
...   pid-file = /var/run/mysqld/mysqld.pid
...   skip-external-locking
...   old_passwords = 1
...   skip-bdb
...   # we don't need ACID today
```

(continúe en la próxima página)

(proviene de la página anterior)

```

... skip-innodb
... """
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)

>>> # Settings with values are treated as before:
>>> config["mysqld"]["user"]
'mysql'

>>> # Settings without values provide None:
>>> config["mysqld"]["skip-bdb"]

>>> # Settings which aren't specified still raise an error:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
...
KeyError: 'does-not-exist'

```

- *delimiters*, valor por defecto: ('=', ':')

Los *delimiters* son cadenas de caracteres que separan las claves de los valores dentro de una sección. La primera ocurrencia de una cadena de separación en una línea se considera como un separador. Esto significa que los valores pueden contener separadores, no así las claves.

Vea también el argumento *space_around_delimiters* de `ConfigParser.write()`.

- *comment_prefixes*, valor por defecto: ('# ', '; ')
- *inline_comment_prefixes*, valor por defecto: None

Los prefijos de comentario son cadenas de caracteres que indican el inicio de un comentario válido dentro de un archivo de configuración. Los *comment_prefixes* son utilizados solamente en lo que serían líneas en blanco (opcionalmente indentadas), mientras que *inline_comment_prefixes* pueden ser utilizados después de cada valor válido (ej. nombres de sección, opciones y también líneas en blanco). De forma predeterminada, los comentarios en la misma línea están deshabilitados, y tanto '#' como ';' se utilizan como prefijos para comentarios que ocupan toda la línea.

Distinto en la versión 3.2: En versiones previas de `configparser` el comportamiento correspondía a `comment_prefixes=('# ', '; ')` e `inline_comment_prefixes=(';', ')`.

Por favor, observe que los *config parsers* no soportan el escapado de los prefijos de comentarios, de modo que la utilización de los *inline_comment_prefixes* puede impedir que los usuarios especifiquen valores de opciones que incluyan caracteres empleados como prefijos de comentarios. Ante la duda, evite especificar los *inline_comment_prefixes*. En cualquier circunstancia, la única forma de almacenar caracteres prefijos de comentario al inicio de una línea, en valores multilínea, es mediante la interpolación del prefijo, por ejemplo:

```

>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
...     ${hash}!/usr/bin/env python
...     ${hash} -*- coding: utf-8 -*-
...
... extensions =
...     enabled_extension
...     another_extension
...     #disabled_by_comment
...     yet_another_extension
...

```

(continué en la próxima página)

(proviene de la página anterior)

```

...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
...     line #2
...     line #3
...     """
>>> print(parser['hashes']['shebang'])

#!/usr/bin/env python
# -*- coding: utf-8 -*-
>>> print(parser['hashes']['extensions'])

enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3

```

- *strict*, valor por defecto: `True`

Cuando tiene el valor `True`, el *parser* no permitirá duplicados en ninguna sección u opción, al efectuar la lectura desde una sola fuente (utilizando `read_file()`, `read_string()` ó `read_dict()`). Se recomienda el uso de *strict parsers* en las aplicaciones nuevas.

Distinto en la versión 3.2: En versiones previas de *configparser* el comportamiento correspondía a `strict=False`.

- *empty_lines_in_values*, valor por defecto: `True`

En los *config parsers*, los valores pueden abarcar varias líneas, siempre y cuando estén indentadas a un nivel superior al de la clave que las contiene. De forma predeterminada, los *parsers* permiten que las líneas en blanco formen parte de los valores. Igualmente, las claves pueden estar indentadas a sí mismas de forma arbitraria, a fin de mejorar la legibilidad. Por lo tanto, cuando los archivos de configuración se vuelven grandes y complejos, es fácil para el usuario el perder la pista de la estructura del archivo. Tomemos como ejemplo:

```

[Section]
key = multiline
    value with a gotcha

    this = is still a part of the multiline value of 'key'

```

Esto puede ser especialmente problemático de observar por parte del usuario, en caso que se esté utilizando un tipo de fuente proporcional para editar el archivo. Y es por ello que, cuando tu aplicación no necesite valores con líneas en blanco, debes considerar invalidarlas. Esto ocasionaría que las líneas en blanco sirvan para dividir a las claves, siempre. En el ejemplo anterior, produciría dos claves: `key` y `this`.

- *default_section*, valor por defecto: `configparser.DEFAULTSECT` (es decir: `"DEFAULT"`)

El convenio de permitir una sección especial con valores por defecto para otras secciones, o con propósito de interpolación, es un concepto poderoso de esta librería, el cual permite a los usuarios crear configuraciones declarativas complejas. Esta sección se suele denominar `"DEFAULT"`, pero puede personalizarse para corresponder a cualquier otro nombre de sección. Algunas valores comunes son: `"general"` ó `"common"`. El nombre proporcionado es utilizado para reconocer las secciones por defecto al momento de realizar la lectura desde cualquier fuente, y es utilizado ante cualquier escritura al archivo de configuración. Su valor actual puede obtenerse utilizando el atributo `parser_instance.default_section` y puede ser modificado en tiempo de ejecución (es decir, para convertir archivos de un formato a otro).

- *interpolation*, valor por defecto: `configparser.BasicInterpolation`

El comportamiento de la interpolación puede ser personalizado al proporcionar un gestor personalizado mediante el argumento *interpolation*. El valor `None` se utiliza para desactivar la interpolación por completo, mientras que `ExtendedInterpolation()` proporciona una variante más avanzada, inspirada por `zc.buildout`. Más información al respecto en la [sección dedicada de la documentación](#). El `RawConfigParser` tiene un valor por defecto de `None`.

- *converters*, valor por defecto: no definido

Los *config parsers* proporcionan métodos para lectura (*getters*) de valores de opciones, que llevan a cabo la conversión de tipo. Están implementados los métodos `getint()`, `getfloat()`, y `getboolean()`, de forma predeterminada. Si se desean otros métodos de lectura (*getters*), los usuarios pueden definirlos en una subclase o pasar un diccionario donde cada clave sea el nombre del conversor y cada valor es un invocable (*callable*) que implemente la conversión requerida. Por ejemplo, al pasar `{'decimal': decimal.Decimal}` se agregaría `getdecimal()` tanto en el objeto *parser* como en todas las secciones proxies. En otras palabras, sería posible escribir tanto `parser_instance.getdecimal('section', 'key', fallback=0)` como `parser_instance['section'].getdecimal('key', 0)`.

Si el conversor necesita acceder al estado del *parser*, este puede ser implementado como un método en una subclase del *config parser*. Si el nombre de este método comienza con `get`, estará disponible en todas las secciones proxy, en su forma compatible con diccionarios (vea el ejemplo anterior de `getdecimal()`).

Una personalización más avanzada puede conseguirse al sustituir los valores por defecto de estos atributos del *parser*. Los valores por defecto son definidos en las clases, de modo que pueden ser sustituidos por las subclases o mediante la asignación de atributos.

`ConfigParser.BOOLEAN_STATES`

Por defecto, al utilizar el método `getboolean()`, los *config parsers* consideran a los siguientes valores como `True`: `'1'`, `'yes'`, `'true'`, `'on'` y a los siguientes valores como `False`: `'0'`, `'no'`, `'false'`, `'off'`. Puedes cambiar ello proporcionando un diccionario personalizado de cadenas de caracteres, con sus correspondientes valores booleanos. Por ejemplo:

```
>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
Traceback (most recent call last):
...
ValueError: Not a boolean: nope
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope': False}
>>> custom['section1'].getboolean('funky')
False
```

Otros pares booleanos comunes incluyen `accept/reject` ó `enabled/disabled`.

`ConfigParser.optionxform(option)`

Este método realiza la transformación de los nombres de opciones para cada operación de lectura, obtención o asignación. Por defecto convierte los nombres a minúsculas. Esto significa que, cuando un archivo de configuración es escrito, todas las claves son convertidas a minúsculas. Sobre-escribe este método si tal comportamiento no es adecuado. Por ejemplo:

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

Nota: La función *optionxform* transforma los nombres de opción a una forma canónica. Esta debería ser una función idempotente: si el nombre ya está en su forma canónica, debería retornarse sin cambios.

ConfigParser.SECTCRE

Es una expresión regular compilada que se utiliza para parsear cabeceras de sección. Por defecto hace corresponder `[section]` con el nombre "section". Los espacios en blanco son considerados parte del nombre de sección, por lo que `[larch]` será leído como la sección de nombre " larch ". Sobre-escribe este atributo si tal comportamiento no es adecuado. Por ejemplo:

```
>>> import re
>>> config = """
... [Section 1]
... option = value
...
... [ Section 2 ]
... another = val
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', ' Section 2 ']
>>> custom = configparser.ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^\]]+?) *\]")
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

Nota: Mientras que los objetos *ConfigParser* también utilizan un atributo `OPTCRE` para reconocer las líneas de opciones, no se recomienda su sobre-escritura porque puede interferir con las opciones *allow_no_value* y *delimiters* del constructor.

14.2.8 Ejemplos de la API heredada

configparser proporciona también una API heredada con métodos *get/set* explícitos; ello, principalmente debido a asuntos de compatibilidad con versiones anteriores. Aunque existen casos de uso válidos para los métodos descritos a continuación, se prefiere el acceso por protocolo de mapeo para los proyectos nuevos. La API heredada es al mismo tiempo más avanzada, de bajo nivel y sumamente contradictoria.

Un ejemplo de escritura a un archivo de configuración:

```
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser's 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. Setting
# values using the mapping protocol or ConfigParser's set() does not allow
```

(continué en la próxima página)

(proviene de la página anterior)

```
# 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', 'w') as configfile:
    config.write(configfile)
```

Un ejemplo de lectura de un archivo de configuración, nuevamente:

```
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'))
```

Para obtener la interpolación, utilice `ConfigParser`:

```
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True if you wish to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False)) # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))  # -> "%(bar)s is %(baz)s!"

# The optional *vars* argument is a dict with members that will take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value
print(cfg.get('Section1', 'foo'))
# -> "Python is fun!"

print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
# -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things as monsters.'))
# -> "No such things as monsters."

# A bare print(cfg.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:
```

(continué en la próxima página)

(proviene de la página anterior)

```
print(cfg.get('Section1', 'monster', fallback=None))
# -> None
```

Los valores por defecto están disponibles en ambos tipos de ConfigParsers. Ellos son utilizados en la interpolación cuando una opción utilizada no está definida en otro lugar.

```
import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = configparser.ConfigParser({'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!"
```

14.2.9 Objetos ConfigParser

```
class configparser.ConfigParser (defaults=None, dict_type=dict, allow_no_value=False,
                                delimiters=('=', ':'), comment_prefixes=(';',
                                ';'), inline_comment_prefixes=None,
                                strict=True, empty_lines_in_values=True, de-
                                fault_section=configparser.DEFAULTSECT, interpola-
                                tion=BasicInterpolation(), converters={})
```

La *config parser* principal. Si se proporciona *defaults*, su valor es inicializado en el diccionario de valores por defecto intrínsecos. Si se proporciona *dict_type*, se utiliza para crear el diccionario de objetos para la lista de secciones, las opciones dentro de una sección, y los valores por defecto.

Si se proporciona *delimiters*, se utiliza como un conjunto de cadenas de caracteres que dividen las claves de los valores. Si se proporciona *comment_prefixes*, se utiliza como un conjunto de cadenas de caracteres que preceden a los comentarios, en líneas que estarían, de otro modo, vacías. Los comentarios pueden estar indentados. Si se proporciona *inline_comment_prefixes*, se utiliza como un conjunto de cadenas de caracteres que preceden a los comentarios en líneas que no están vacías.

Cuando *strict* es *True* (por defecto), el *parser* no permitirá duplicados en ninguna sección u opción, al realizar la lectura de una sola fuente (archivo, cadena de caracteres o diccionario), generando una excepción *DuplicateSectionError* ó *DuplicateOptionError*. Cuando *empty_lines_in_values* es *False* (valor por defecto: *True*), cada línea en blanco indica el fin de una opción. De otro modo, las líneas en blanco de una opción multilínea son tratadas como parte del valor de esta. Cuando *allow_no_value* es *True* (valor por defecto: *False*), se aceptan opciones sin valores; el valor que toman esas opciones es *None* y serán serializadas sin el delimitador final.

Cuando se proporciona *default_section*, se define el nombre de la sección especial que contiene valores por defecto para otras secciones y con propósito de interpolación (habitualmente denominada "DEFAULT"). Este valor puede obtenerse y modificarse en tiempo de ejecución utilizando el atributo de instancia *default_section*.

La interpolación puede personalizarse al proporcionar un gestor personalizado, mediante el argumento *interpolation*. El valor *None* se utiliza para desactivar la interpolación completamente, *ExtendedInterpolation()* proporciona una variante avanzada, inspirada en *zc.buildout*. Más al respecto puede encontrarse en la *correspondiente sección de la documentación*.

Todos los nombres de opción que se utilizan en la interpolación pasarán por el método *optionxform()*, igual que cualquier otra referencia a un nombre de opción. Por lo tanto, si se utiliza la implementación por defecto de *optionxform()* (la cual convierte los nombres de opción a minúsculas), los valores *foo % (bar)* s y *foo % (BAR)* s son equivalentes.

Cuando se proporciona *converters*, este debe ser un diccionario, donde cada clave representa el nombre de un conversor, y cada valor un invocable que implementa la conversión de la cadena de caracteres al tipo de

datos deseado. Cada conversor recibe su método `get *()` correspondiente en el objeto *parser* y los proxies de sección.

Distinto en la versión 3.1: El valor por defecto de *dict_type* es `collections.OrderedDict`.

Distinto en la versión 3.2: se agregaron los argumentos *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* y *interpolation*.

Distinto en la versión 3.5: Se agregó el argumento *converters*.

Distinto en la versión 3.7: El argumento *defaults* es leído con `read_dict()`, proporcionando un comportamiento consistente en el *parser*: las claves y valores que no sean cadenas de caracteres son convertidas a tales.

Distinto en la versión 3.8: El valor por defecto para *dict_type* es `dict`, dado que este ahora preserva el orden de inserción.

defaults()

Retorna un diccionario que contiene los valores por defecto para toda la instancia.

sections()

Retorna una lista de las secciones disponibles; *default section* no se incluye en la lista.

add_section(section)

Agrega una sección llamada *section* a la instancia. Si ya existe una sección con el nombre proporcionado, se genera la excepción `DuplicateSectionError`. Si se suministra el nombre *default section*, se genera la excepción `ValueError`. El nombre de la sección debe ser una cadena de caracteres, de lo contrario, se genera la excepción `TypeError`.

Distinto en la versión 3.2: Nombres de sección que no sean del tipo cadena de caracteres generan la excepción `TypeError`.

has_section(section)

Indica si la sección de nombre *section* existe en la configuración. No se permite *default section*.

options(section)

Retorna una lista de opciones disponibles en la sección especificada.

has_option(section, option)

Si la sección indicada existe, y esta contiene la opción proporcionada, retorna `True`; de lo contrario, retorna `False`. Si la sección especificada es `None` o una cadena de caracteres vacía, se asume `DEFAULT`.

read(filenames, encoding=None)

Intenta leer y analizar sintácticamente (*parse*) un iterable de nombres de archivos, retornando una lista de nombres de archivos que han sido analizados (*parsed*) con éxito.

Si *filenames* es una cadena de caracteres, un objeto `bytes` o un *path-like object*, es tratado como un simple nombre de archivo. Si un archivo mencionado en *filenames* no puede ser abierto, será ignorado. Está diseñado de forma que puedas especificar un iterable de potenciales ubicaciones para archivos de configuración (por ejemplo, el directorio actual, el directorio *home* del usuario, o algún directorio del sistema), y todos los archivos de configuración existentes serán leídos.

Si no existe ninguno de los archivos mencionados, la instancia de `ConfigParser` contendrá un conjunto de datos vacío. Una aplicación que requiera valores iniciales, que sean cargados desde un archivo, deberá cargar el(los) archivo(s) requerido(s) utilizando `read_file()` antes de llamar a `read()` para cualquier otro archivo opcional:

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

Nuevo en la versión 3.2: El parámetro *encoding*. Anteriormente, todos los archivos eran leídos utilizando la codificación por defecto de la la función `open()`.

Nuevo en la versión 3.6.1: El parámetro *filenames* acepta un *path-like object*.

Nuevo en la versión 3.7: El parámetro *filenames* acepta un objeto *bytes*.

read_file (*f*, *source=None*)

Leer y analizar (*parse*) los datos de configuración de *f*, el cual debe ser un iterable que retorne cadenas de caracteres Unicode (por ejemplo, archivos abiertos en modo texto).

El argumento opcional *source* especifica el nombre del archivo que se está leyendo. Si no se proporciona y *f* tiene un atributo `name`, este es utilizado como *source*; el valor por defecto es '`<???`'.

Nuevo en la versión 3.2: Reemplaza a `readfp()`.

read_string (*string*, *source='<string>'*)

Analiza (*parse*) los datos de configuración desde una cadena de caracteres.

El argumento opcional *source* especifica un nombre para la cadena de caracteres proporcionada, relativo al contexto. Si no se proporciona, se utiliza '`<string>`'. Esto, por lo general, debería ser una ruta de archivo o una URL.

Nuevo en la versión 3.2.

read_dict (*dictionary*, *source='<dict>'*)

Carga la configuración a partir de cualquier objeto que proporcione un método `items()`, similar a un diccionario. Las claves son nombres de secciones, los valores son diccionarios con claves y valores que deben estar presentes en la sección. Si el tipo de diccionario utilizado preserva el orden, las secciones y sus claves serán agregados en orden. Los valores son convertidos a cadenas de caracteres de forma automática.

El argumento opcional *source* especifica un nombre para el diccionario proporcionado, relativo al contexto. Si no se proporciona, se utiliza `<dict>`.

Este método puede utilizarse para copiar el estado entre *parsers*.

Nuevo en la versión 3.2.

get (*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

Obtiene el valor de una *option* para la sección indicada. Si se proporciona *vars*, tiene que ser un diccionario. El valor de *option* será buscado en *vars* (si se proporciona), en *section* y en `DEFAULTSECT`, en ese orden. Si la clave no se encuentra, y se ha proporcionado *fallback*, este es utilizado como un valor de contingencia. Se puede utilizar `None` como valor de contingencia (*fallback*).

Todas las interpolaciones '`%`' son expandidas en el valor retornado, a menos que el argumento *raw* sea *true*. Los valores para la interpolación de las claves son buscados de la misma forma que para la opción.

Distinto en la versión 3.2: Los argumentos *raw*, *vars* y *fallback* son argumentos nombrados solamente, a fin de proteger a los usuarios de los intentos de emplear el tercer argumento como el valor de contingencia de *fallback* (especialmente cuando se utiliza el protocolo de mapeo).

getint (*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

Un cómodo método para forzar a entero el valor de la opción de la sección indicada. Revise `get()` para una explicación acerca de *raw*, *vars* y *fallback*.

getfloat (*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

Un cómodo método para forzar a número de punto flotante el valor de la opción de la sección indicada. Revise `get()` para una explicación acerca de *raw*, *vars* y *fallback*.

getboolean (*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

Un cómodo método para forzar a booleano el valor de la opción de la sección indicada. Tome nota que los valores aceptados para la opción son '`1`', '`yes`', '`true`', and '`on`', para que el método retorne `True`, y '`0`', '`no`', '`false`', y '`off`' para que el método retorne `False`. Esos valores de cadenas de caracteres son revisados sin diferenciar mayúsculas de minúsculas. Cualquier otro valor ocasionará que se genere la excepción `ValueError`. Revise `get()` para una explicación acerca de *raw*, *vars* y *fallback*.

items (*raw=False, vars=None*)

items (*section, raw=False, vars=None*)

Cuando no se proporciona el argumento *section*, retorna una lista de pares *section_name, section_proxy*, incluyendo DEFAULTSECT.

De lo contrario, retorna una lista de pares *name, value* para las opciones de la sección especificada. Los argumentos opcionales tienen el mismo significado que en el método `get()`.

Distinto en la versión 3.8: Los elementos que estén en *vars* no aparecen en el resultado. El comportamiento previo mezcla las opciones actuales del *parser* con las variables proporcionadas para la interpolación.

set (*section, option, value*)

Si la sección indicada existe, asigna el valor especificado a la opción indicada; en caso contrario, genera la excepción `NoSectionError`. *option* y *value* deben ser cadenas de caracteres; de lo contrario, se genera la excepción `TypeError`.

write (*fileobject, space_around_delimiters=True*)

Escribe una representación de la configuración al *file object* especificado, el cual debe ser abierto en modo texto (aceptando cadenas de caracteres). Esta representación puede ser analizada (*parsed*) por una posterior llamada a `read()`. Si *space_around_delimiters* es *true*, los delimitadores entre claves y valores son rodeados por espacios.

remove_option (*section, option*)

Elimina la opción especificada de la sección indicada. Si la sección no existe, se genera una excepción `NoSectionError`. Si la opción existía antes de la eliminación, retorna *True*; de lo contrario retorna *False*.

remove_section (*section*)

Elimina la sección especificada de la configuración. Si la sección existía, retorna *True*. De lo contrario, retorna *False*.

optionxform (*option*)

Transforma el nombre de opción *option* de la forma en que se encontraba en el archivo de entrada o como fue pasada por el código del cliente, a la forma en que debe ser utilizada en las estructuras internas. La implementación por defecto retorna una versión en minúsculas de *option*; las subclases pueden sobrescribirla o el código del cliente puede asignar un atributo de su nombre en las instancias, para afectar su comportamiento.

No necesitas crear una subclase del *parser* para utilizar este método, ya que también puedes asignarlo en una instancia a una función, la cual reciba una cadena de caracteres como argumento y retorne otra. Por ejemplo, estableciéndola a `str`, se hará que los nombres de opciones sean sensibles a mayúsculas y minúsculas:

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

Tome en cuenta que cuando se leen archivos de configuración, los espacios en blanco alrededor de los nombres de opción son eliminados antes de llamar a `optionxform()`.

readfp (*fp, filename=None*)

Obsoleto desde la versión 3.2: Utilice `read_file()` en su lugar.

Distinto en la versión 3.2: Ahora, `readfp()` itera sobre *fp*, en lugar de llamar a `fp.readline()`.

Para el código existente, que llama a `readfp()` sin argumentos que soporten la iteración, el siguiente generador puede utilizarse como un envoltorio (*wrapper*) para el objeto semejante a archivo:

```
def readline_generator(fp):
    line = fp.readline()
    while line:
        yield line
        line = fp.readline()
```

Utilice `parser.read_file(readline_generator(fp))` en lugar de `parser.readfp(fp)`.

`configparser.MAX_INTERPOLATION_DEPTH`

La profundidad máxima de interpolación para `get()` cuando el parámetro `raw` es `false`. Esto es de importancia solamente cuando la interpolación por defecto es empleada.

14.2.10 Objetos `RawConfigParser`

```
class configparser.RawConfigParser (defaults=None, dict_type=dict, allow_no_value=False,
                                     *, delimiters=('=', ':'), comment_prefixes=(';',
                                     ';'), inline_comment_prefixes=None,
                                     strict=True, empty_lines_in_values=True, de-
                                     fault_section=configparser.DEFAULTSECT[, inter-
                                     polation])
```

Variante heredada de `ConfigParser`. Tiene la interpolación deshabilitada por defecto y permite nombres de sección que no sean cadenas de caracteres, nombres de opciones, y valores a través de sus métodos inseguros `add_section` y `set`, así como el manejo heredado del argumento nombrado `defaults=`.

Distinto en la versión 3.8: El valor por defecto para `dict_type` es `dict`, dado que este ahora preserva el orden de inserción.

Nota: Considere el uso de `ConfigParser` en su lugar, el cual verifica los tipos de datos de los valores que se almacenarán internamente. Si no quieres la interpolación, puedes utilizar `ConfigParser(interpolation=None)`.

add_section (*section*)

Agrega a la instancia una sección de nombre *section*. Si ya existe una sección con el nombre proporcionado, se genera la excepción `DuplicateSectionError`. Si se suministra el nombre *default section*, se genera la excepción `ValueError`.

No se comprueba el tipo de datos de *section*, con lo cual se permite que los usuarios creen secciones con nombres que no sean cadenas de caracteres. Este comportamiento no está soportado y puede ocasionar errores internos.

set (*section, option, value*)

Si existe la sección indicada, asigna el valor especificado a la sección indicada; de lo contrario, genera la excepción `NoSectionError`. Aunque es posible utilizar `RawConfigParser` (ó `ConfigParser` con parámetros `raw` con valor `true`) como almacenamiento interno para valores que no sean cadenas de caracteres, el funcionamiento completo (incluyendo la interpolación y escritura en archivos) sólo puede lograrse utilizando valores del tipo cadena de caracteres.

Este método permite que los usuarios asignen valores que no sean cadenas de caracteres a las claves, internamente. Este comportamiento no está soportado y causará errores cuando se intente escribir en un archivo o cuando se intente obtenerlo en un modo no `raw`. **Utilice la API del protocolo de mapeo**, la cual no permite ese tipo de asignaciones.

14.2.11 Excepciones

exception `configparser.Error`

Clase base para todas las otras excepciones `configparser`.

exception `configparser.NoSectionError`

Excepción generada cuando no se encuentra una sección especificada.

exception `configparser.DuplicateSectionError`

Excepción generada si el método `add_section()` es llamado proporcionando el nombre de una sección que ya existe, o, en caso de *parsers* estrictos, si una sección se encuentra más de una vez en un solo archivo de entrada, cadena de caracteres o diccionario.

Nuevo en la versión 3.2: Al método `__init__()` se agregaron los atributos y argumentos opcionales `source` y `lineno`.

exception `configparser.DuplicateOptionError`

Excepción generada por *parsers* estrictos si una sola opción aparece dos veces durante la lectura de un solo archivo, cadena de caracteres o diccionario. Captura errores de escritura o relacionados con el uso de mayúsculas y minúsculas, ej. un diccionario podría tener dos claves que representan la misma clave de configuración bajo un esquema insensible a mayúsculas y minúsculas.

exception `configparser.NoOptionError`

Excepción generada cuando una opción especificada no se encuentra en una sección indicada.

exception `configparser.InterpolationError`

Clase base para excepciones generadas por problemas que ocurren al realizar la interpolación de cadenas de caracteres.

exception `configparser.InterpolationDepthError`

Excepción generada cuando la interpolación de cadenas de caracteres no puede completarse, debido a que el número de iteraciones excede a `MAX_INTERPOLATION_DEPTH`. Subclase de `InterpolationError`.

exception `configparser.InterpolationMissingOptionError`

Excepción generada cuando no existe una opción referida por un valor. Subclase de `InterpolationError`.

exception `configparser.InterpolationSyntaxError`

Excepción generada cuando el texto fuente, donde se realizan las sustituciones, no se ajusta a la sintaxis requerida. Subclase de `InterpolationError`.

exception `configparser.MissingSectionHeaderError`

Excepción generada cuando se intenta analizar (*parse*) un archivo que no tiene encabezados de sección.

exception `configparser.ParsingError`

Excepción generada cuando ocurren errores intentando analizar (*parse*) un archivo.

Distinto en la versión 3.2: El atributo `filename` y el argumento `__init__()` fueron renombrados a `source` por consistencia.

Notas al pie

14.3 netrc — procesamiento del fichero netrc

Código fuente: [Lib/netrc.py](#)

La clase `netrc` analiza y encapsula el formato del fichero `netrc`, usado por el programa Unix **ftp** y otros clientes FTP.

class `netrc.netrc` (`[file]`)

Una instancia de `netrc` o una instancia de una subclase encapsula la información del fichero `netrc`. El argumento de inicialización, si está presente, especifica el fichero a analizar. Si no se pasan argumentos, se leerá el fichero `.netrc` del directorio home del usuario – determinado por `os.path.expanduser()`. De lo contrario, se lanzará una excepción `FileNotFoundError`. Los errores de análisis lanzarán una excepción `NetrcParseError` con información de diagnóstico que incluye nombre del fichero, número de línea y token de finalización. Si no se especifica ningún argumento en un sistema POSIX, la presencia de contraseñas en el fichero `.netrc` lanzará una excepción `NetrcParseError` si la propiedad del fichero o los permisos son inseguros (el propietario del fichero es distinto del usuario que ejecuta el proceso, o puede ser leído o escrito por cualquier otro usuario). Esto implementa un nivel de seguridad equivalente al de `ftp` y otros programas que usan `.netrc`.

Distinto en la versión 3.4: Añadida la comprobación de permisos POSIX.

Distinto en la versión 3.7: `os.path.expanduser()` se usa para encontrar la localización del fichero `.netrc` cuando `file` no se pasa como argumento.

exception `netrc.NetrcParseError`

Excepción lanzada por la clase `netrc` cuando se encuentran errores sintácticos en el texto origen. Las instancias de esta excepción ofrecen tres atributos interesantes: `msg` es una explicación textual del error, `filename` es el nombre del fichero origen, y `lineno` indica el número de línea en el que se encontró el error.

14.3.1 Objetos `netrc`

Una instancia `netrc` tiene los siguientes métodos:

`netrc.authenticators` (*host*)

Retorna una 3-tupla (`login`, `account`, `password`) para autenticarse contra *host*. Si el fichero `netrc` no contiene una entrada para el *host* dado, retorna una tupla asociada con la entrada por defecto. Si no están disponibles ni el *host* correspondiente ni la entrada por defecto, retorna `None`.

`netrc.__repr__` ()

Vuelca los datos de la clase como una cadena de caracteres en el formato de un fichero `netrc`. (Esto descarta comentarios y puede reordenar las entradas.)

Las instancias de `netrc` tienen variables de instancia públicas:

`netrc.hosts`

Diccionario que asocia nombres de *hosts* a tuplas (`login`, `account`, `password`). La entrada por defecto, si existe, está representada como un pseudo-*host* por ese nombre.

`netrc.macros`

Diccionario que asocia nombres de macros a listas de cadenas de caracteres.

Nota: Las contraseñas están limitadas a un subconjunto del conjunto de caracteres ASCII. En las contraseñas se permiten todos los símbolos de puntuación ASCII. Sin embargo, no se permiten espacios en blanco ni caracteres no imprimibles. Esto es una limitación de la manera en que se analiza el fichero `.netrc` y puede que se elimine en el futuro.

14.4 `xdrlib` — Codificar y decodificar datos XDR

Código fuente: [Lib/xdrlib.py](#)

El módulo `xdrlib` soporta el estándar de representación externa de datos (*External Data Representation Standard*) como se describe en [RFC 1014](#), escrito por Sun Microsystems, Inc. en junio de 1987. Soporta la mayoría de los tipos de datos descritos en el RFC.

El módulo `xdrlib` define dos clases, una para empaquetar variables en la representación XDR, y otra para desempaquetar valores de la representación XDR. También hay dos clases de excepciones.

class `xdrlib.Packer`

`Packer` es la clase para empaquetar datos en la representación XDR. La clase `Packer` es instanciada sin argumentos.

class `xdrlib.Unpacker` (*data*)

`Unpacker` es la clase complementaria que desempaqueta los valores de datos XDR de un búfer de cadena. El búfer de entrada se proporciona como *data*.

Ver también:

RFC 1014 - XDR: estándar de representación externa de datos Este RFC definió la codificación de datos que era XDR en el momento en que este módulo fue escrito originalmente. Aparentemente ha quedado obsoleto, siendo reemplazado por [RFC 1832](#).

RFC 1832 - XDR: estándar de representación externa de datos El RFC más reciente que proporciona una definición revisada de XDR.

14.4.1 Instancias de la clase *Packer*

Las instancias de la clase *Packer* poseen los siguientes métodos:

`Packer.get_buffer()`

Retorna el búfer del paquete actual como una cadena de caracteres.

`Packer.reset()`

Restablece el búfer del paquete a la cadena de caracteres vacía.

En general, puedes empaquetar cualquiera de los tipos de datos XDR más comunes, invocando el método `pack_tipo()` apropiado. Cada método toma un solo argumento, el valor a empaquetar. Los siguientes métodos simples de empaquetado de tipos de datos son soportados: `pack_uint()`, `pack_int()`, `pack_enum()`, `pack_bool()`, `pack_uhyper()`, y `pack_hyper()`.

`Packer.pack_float(value)`

Empaqueta el número de punto flotante de precisión simple, *value*.

`Packer.pack_double(value)`

Empaqueta el número de punto flotante de doble precisión, *value*.

Los siguientes métodos soportan el empaquetado de cadenas de caracteres, bytes y datos opacos:

`Packer.pack_fstring(n, s)`

Empaqueta una cadena de caracteres de longitud fija, *s*. *n* es la longitud de la cadena de caracteres pero *no* es empaquetada en el búfer de datos. La cadena de caracteres es rellenada con bytes nulos si es necesario, para garantizar que la longitud de los datos sea un múltiplo de 4 bytes.

`Packer.pack_fopaque(n, data)`

Empaqueta un flujo de datos opaco de longitud fija, similar a `pack_fstring()`.

`Packer.pack_string(s)`

Empaqueta una cadena de caracteres de longitud variable, *s*. En primer lugar, la longitud de la cadena de caracteres es empaquetada como un entero sin signo, luego los datos de la cadena de caracteres son empaquetados con `pack_fstring()`.

`Packer.pack_opaque(data)`

Empaqueta una cadena de caracteres de datos opacos de longitud variable, similar a `pack_string()`.

`Packer.pack_bytes(bytes)`

Empaqueta una secuencia de bytes de longitud variable, similar a `pack_string()`.

Los siguientes métodos soportan el empaquetamiento de arreglos y listas:

`Packer.pack_list(list, pack_item)`

Empaqueta una *lista* de elementos homogéneos. Este método es útil para listas con longitud indeterminada; en otras palabras, el tamaño no está disponible hasta que se haya recorrido toda la lista. Para cada elemento de la lista, un entero sin signo 1 se empaqueta primero, seguido del valor de datos de la lista. *pack_item* es la función que se llama para empaquetar el elemento individual. Al final de la lista, se empaqueta un entero sin signo 0.

Por ejemplo, para empaquetar una lista de enteros, el código podría parecerse a esto:

```
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

`Packer.pack_farray(n, array, pack_item)`

Empaqueta una lista (*arreglo*) de elementos homogéneos de longitud fija. *n* es la longitud de la lista; *no* es empaquetada en el búfer, pero una excepción *ValueError* es lanzada si `len(array)` no es igual a *n*. Al igual que en el método anterior, *pack_item* es la función usada para empaquetar cada elemento.

`Packer.pack_array(list, pack_item)`

Empaqueta una *lista* de elementos homogéneos de longitud variable. Primero, la longitud de la lista es empaquetada como un entero sin signo, luego cada elemento es empaquetado como en el método `pack_farray()` de arriba.

14.4.2 Instancias de la clase *Unpacker*

La clase *Unpacker* ofrece los siguientes métodos:

`Unpacker.reset(data)`

Restablece el búfer de cadena de caracteres a la *data* especificada.

`Unpacker.get_position()`

Retorna la posición actual desempaketada en el búfer de datos.

`Unpacker.set_position(position)`

Establece la posición de desempaketado del búfer de datos en la posición *position*. Debes tener cuidado al usar los métodos `get_position()` y `set_position()`.

`Unpacker.get_buffer()`

Retorna el búfer de datos actual desempaketado como una cadena de caracteres.

`Unpacker.done()`

Indica se ha completado el proceso de desempaketado. Lanza una excepción *Error* si no se han desempaketado todos los datos.

Además, cada tipo de datos que puede ser empaquetados con un *Packer*, puede ser desempaketado con un *Unpacker*. Los métodos de desempaketados son de la forma `unpack_tipo()`, y no tienen argumentos. Estos retornan el objeto desempaketado.

`Unpacker.unpack_float()`

Desempaketa un número de punto flotante de precisión simple.

`Unpacker.unpack_double()`

Desempaketa un número de punto flotante de doble precisión, similar a `unpack_float()`.

Adicionalmente, los siguientes métodos desempaketan cadenas de caracteres, bytes y datos opacos:

`Unpacker.unpack_fstring(n)`

Desempaketa y retorna una cadena de caracteres de longitud fija. *n* es el número de caracteres esperados. Se asume un relleno con bytes nulos hasta que la longitud de los datos sea un múltiplo de 4 bytes.

`Unpacker.unpack_fopaque(n)`

Desempaketa y retorna un flujo de datos opaco de longitud fija, similar a `unpack_fstring()`.

`Unpacker.unpack_string()`

Desempaketa y retorna una cadena de caracteres de longitud variable. La longitud de la cadena de caracteres es desempaketada primero como un entero sin signo, luego los datos de la cadena de caracteres son desempaketados con el método `unpack_fstring()`.

`Unpacker.unpack_opaque()`

Desempaketa y retorna un flujo de datos opaco de longitud variable, similar a `unpack_string()`.

`Unpacker.unpack_bytes()`

Desempaketa y retorna una secuencia de bytes de longitud variable, similar a `unpack_string()`.

Los siguientes métodos soportan el desempaketado de arreglos y listas:

`Unpacker.unpack_list(unpack_item)`

Desempaketa y retorna una lista de elementos homogéneos. La lista se desempaketa un elemento a la vez desempaketando primero un indicador entero sin signo. Si el flag es 1, el elemento se desempaketa y se agrega a la lista. Un flag de 0 indica el final de la lista. `unpack_item` es la función a la que se invoca para desempaketar los elementos.

`Unpacker.unpack_farray(n, unpack_item)`

Desempaketa y retorna (como una lista) un arreglo de elementos homogéneos de longitud fija. *n* es el número de elementos de la lista que se esperan en el búfer. Como se mostró anteriormente, `unpack_item` es la función empleada para desempaketar cada elemento.

`Unpacker.unpack_array(unpack_item)`

Desempaketa y retorna una *lista* de elementos homogéneos de longitud variable. En primer lugar, la longitud

de la lista se desempaqueta como un entero sin signo, luego cada elemento se desempaqueta como en el método `unpack_farray()` de arriba.

14.4.3 Excepciones

En este módulo, las excepciones se codifican como instancias de clase:

exception `xdrlib.Error`

La clase de excepción base. `Error` tiene un único atributo público, `msg`, que contiene la descripción del error.

exception `xdrlib.ConversionError`

Clase derivada de `Error`. No contiene variables de instancia adicionales.

A continuación se muestra un ejemplo de cómo detectarías una de estas excepciones:

```
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print('packing the double failed:', instance.msg)
```

14.5 plistlib — Genera y analiza archivos .plist de Mac OS X

Código fuente: [Lib/plistlib.py](#)

Este módulo provee una interfaz para lectura y escritura de archivos de «listas de propiedades» usados principalmente por Mac OS X y soporta tanto archivos plist binarios como XML.

El formato de lista de propiedades (`.plist`) es una serialización simple que soporta tipos de objetos básicos, como diccionarios, listas, números y cadenas. Generalmente el objeto de nivel superior es un diccionario.

Para escribir y analizar un archivo plist usa las funciones `dump()` y `load()`.

Para trabajar con datos plist en objetos de bytes usa `dumps()` y `loads()`.

Los valores pueden ser cadenas de caracteres, enteros, coma flotantes, booleanos, tuplas, listas, diccionarios (pero sólo con cadenas como claves), `Data`, `bytes`, `bytesarray` u objetos `datetime.datetime`.

Distinto en la versión 3.4: Nueva API, API antigua obsoleta. Añadido soporte para plists en formato binario.

Distinto en la versión 3.8: Añadido soporte para lectura y escritura de tokens `UID` en plists binarios como lo usan `NSKeyedArchiver` y `NSKeyedUnarchiver`.

Ver también:

Página del manual PList Documentación de Apple del formato de archivo.

Este módulo define las siguientes funciones:

`plistlib.load(fp, *, fmt=None, use_builtin_types=True, dict_type=dict)`

Lee un archivo plist. `fp` debe ser un objeto de archivo binario y legible. Retorna el objeto raíz desempquetado (el cual generalmente es un diccionario).

El `fmt` es el formato del archivo y los siguientes valores son válidos:

- `None`: Autodetecta el formato de archivo
- `FMT_XML`: Formato de archivo XML
- `FMT_BINARY`: Formato binario plist

Si `use_builtin_types` es verdadero (el valor por defecto) los datos binarios serán retornados como instancias de `bytes`, si no serán retornados como instancias de `Data`.

EL `dict_type` es el tipo usado por los diccionarios que son leídos del archivo plist.

Los datos XML para el formato `FMT_XML` son analizados usando el analizador Expat desde `xml.parsers.expat` – consulte su documentación para conocer las posibles excepciones en XML mal formado. Elementos desconocidos serán simplemente ignorados por el analizador plist.

El analizar para el formato binario lanza `InvalidFileException` cuando el archivo no puede ser analizado.

Nuevo en la versión 3.4.

`plistlib.loads (data, *, fmt=None, use_builtin_types=True, dict_type=dict)`

Carga un plist desde un objeto de bytes. Consulte `load()` para una explicación de los argumentos de palabra clave.

Nuevo en la versión 3.4.

`plistlib.dump (value, fp, *, fmt=FMT_XML, sort_keys=True, skipkeys=False)`

Escribe `value` a un archivo plist. `Fp` debe ser un objeto de archivo binario escribible.

El argumento `fmt` especifica el formato del archivo plist y puede ser uno de los siguientes valores:

- `FMT_XML`: Archivo plist con formato XML
- `FMT_BINARY`: Archivo plist con formato binario

Cuando `sort_keys` es verdadero (el valor por defecto) las claves para los diccionarios serán escritas al plist ordenadamente, si no serán escritas en el orden de iteración del diccionario.

Cuando `skipkeys` es falso (el valor por defecto) la función levanta `TypeError` cuando una clave del diccionario no es una cadena de caracteres, si no tales claves serán omitidas.

Una excepción `TypeError` será levantada si el objeto es un tipo no admitido o el contenedor que contiene tipos no admitidos.

Una excepción `OverflowError` será levantada para valores enteros que no pueden ser representados en archivos plist (binarios).

Nuevo en la versión 3.4.

`plistlib.dumps (value, *, fmt=FMT_XML, sort_keys=True, skipkeys=False)`

Retorna `value` como un objeto de bytes con formato plist. Consulte la documentación de `dump()` para una explicación de los argumentos de palabra clave de esta función.

Nuevo en la versión 3.4.

La siguientes funciones están obsoletas:

`plistlib.readPlist (pathOrFile)`

Lee un archivo plist. `pathOrFile` puede ser tanto un nombre de archivo o un (legible y binario) objeto de archivo. Retorna el objeto raíz desempaquetado (el cual generalmente es un diccionario).

Esta función llama a `load()` para hacer el trabajo actual, consulte la documentación de *esa función* para una explicación de los argumentos de palabra clave.

Obsoleto desde la versión 3.4: Usa `load()` en su lugar.

Distinto en la versión 3.7: Los valores de diccionario en el resultado son ahora diccionarios normales. No puedes usar más el acceso mediante atributo para acceder a elementos de esos diccionarios.

`plistlib.writePlist (rootObject, pathOrFile)`

Escribe `rootObject` a un archivo plist XML. `pathOrFile` puede ser tanto un nombre de archivo o un (escribible y binario) archivo de objeto

Obsoleto desde la versión 3.4: Usa `dump()` en su lugar.

`plistlib.readPlistFromBytes(data)`

Lee datos plist de un objeto de bytes. Retorna el objeto raíz.

Consulta `load()` para una descripción de los argumentos de palabra clave.

Obsoleto desde la versión 3.4: Usa `loads()` en su lugar.

Distinto en la versión 3.7: Los valores de diccionario en el resultado son ahora diccionarios normales. No puedes usar más el acceso mediante atributo para acceder a elementos de esos diccionarios.

`plistlib.writePlistToBytes(rootObject)`

Retorna `rootObject` como un objeto de bytes con formato XML plist.

Obsoleto desde la versión 3.4: Usa `dumps()` en su lugar.

Las siguientes clases están disponibles:

class `plistlib.Data(data)`

Retorna un objeto contenedor de «datos» alrededor del objeto de bytes `data`. Este es usado en funciones convirtiendo desde/hacia plists para representar el tipo `<data>` disponible en plists.

Tiene un atributo, `data`, que puede ser usado para recuperar el objeto de bytes de Python almacenado en él.

Obsoleto desde la versión 3.4: Usa un objeto `bytes` en su lugar.

class `plistlib.UID(data)`

Envuelve un `int`. Este es usado leyendo o escribiendo datos codificados `NSKeyedArchiver`, los cuales contienen `UID` (ver manual `PList`).

Tiene un atributo, `data`, el cual puede ser usado para recuperar el valor `int` del `UID`. `data` debe estar en el rango $0 \leq data < 2^{**64}$.

Nuevo en la versión 3.8.

Las siguientes constantes están disponibles:

`plistlib.FMT_XML`

El formato XML para archivos plist.

Nuevo en la versión 3.4.

`plistlib.FMT_BINARY`

El formato binario para archivos plist

Nuevo en la versión 3.4.

14.5.1 Ejemplos

Generar un 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!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.fromtimestamp(time.mktime(time.gmtime())),
)
with open(fileName, 'wb') as fp:
    dump(pl, fp)
```

Analizar un plist:

```
with open(fileName, 'rb') as fp:
    pl = load(fp)
print(pl["aKey"])
```

Servicios Criptográficos

Los módulos descritos en este capítulo implementan varios algoritmos de naturaleza criptográfica. Están disponibles a discreción de la instalación. En sistema Unix, el módulo *crypt* también puede estar disponible. Aquí una descripción:

15.1 *hashlib* — Hashes seguros y resúmenes de mensajes

Código fuente: [Lib/hashlib.py](#)

Este módulo implementa una interfaz común a diferentes algoritmos de hash y resúmenes de mensajes seguros. Están incluidos los algoritmos de hash FIPS seguros SHA1, SHA224, SHA256, SHA384 y SHA512 (definidos en FIPS 180-2) además del algoritmo MD5 de RSA (definido en Internet [RFC 1321](#)). Los términos «hash seguro» y «resumen de mensaje» son intercambiables. Los algoritmos más antiguos fueron denominados resúmenes de mensajes. El término moderno es hash seguro.

Nota: Si quieres las funciones de hash *adler32* o *crc32*, están disponibles en el módulo *zlib*.

Advertencia: Algunos algoritmos tienen conocidas debilidades de colisión de hash, consulte la sección «Ver también» al final.

15.1.1 Algoritmos de hash

Hay un método constructor nombrado para cada tipo de *hash*. Todos retornan un objeto de hash con la misma interfaz simple. Por ejemplo, usa `sha256()` para crear un objeto de hash SHA-256. Ahora puedes alimentar este objeto con *objetos como bytes* (normalmente *bytes*) usando el método `update()`. En cualquier punto puedes pedir el resumen (*digest*) de la concatenación de los datos alimentados al mismo usando los métodos `digest()` o `hexdigest()`.

Nota: Para un rendimiento multihilo mejor, el *GIL* de Python es liberado para datos superiores a 2047 bytes en la creación o actualización de objetos.

Nota: La alimentación de objetos de cadenas en `update()` no está soportada, ya que los hashes funcionan en bytes, no en caracteres.

Los constructores para algoritmos de hash siempre presentes en este módulo son `sha1()`, `sha224()`, `sha256()`, `sha384()`, `sha512()`, `blake2b()` y `blake2s()`. `md5()` también está disponible normalmente, aunque puede faltar si estás usando una construcción de Python que cumple de forma rara con los requisitos FIPS. Algoritmos adicionales también pueden estar disponibles dependiendo de la biblioteca OpenSSL que Python usa en tu plataforma. En la mayoría de plataformas las funciones `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()` y `shake_256()` también están disponibles.

Nuevo en la versión 3.6: Constructores SHA3 (Keccak) y SHAKE `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()`.

Nuevo en la versión 3.6: Fueron añadidas `blake2b()` y `blake2s()`.

Por ejemplo, para obtener el resumen de la cadena de bytes `b'Nobody inspects the spammish repetition'`:

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xddAe\x15\x93\xc5\xfe\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\
↪x95\x0fK\x94\x06'
>>> m.digest_size
32
>>> m.block_size
64
```

Más resumido:

```
>>> hashlib.sha224(b"Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

`hashlib.new(name[, data])`

Es un constructor genérico que toma la cadena *name* del algoritmo deseado como su primer parámetro. También existe para permitir acceso a los hashes arriba listados así como cualquiera de los otros algoritmos que tu biblioteca OpenSSL puede ofrecer. Los constructores nombrados son mucho más rápidos que `new()` y deberían preferirse.

Usando `new()` con un algoritmo provisto por OpenSSL:

```
>>> h = hashlib.new('sha512_256')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'19197dc4d03829df858011c6c87600f994a858103bbc19005f20987aa19a97e2'
```

Hashlib provee los siguientes atributos constantes:

`hashlib.algorithms_guaranteed`

Un conjunto que contiene los nombres de los algoritmos garantizados a ser soportados por este módulo en todas las plataformas. Ten en cuenta que “md5” se encuentra en esta lista a pesar de que algunos proveedores ofrecen una extraña construcción Python «compatible con FIPS» que la excluye.

Nuevo en la versión 3.2.

`hashlib.algorithms_available`

Un conjunto que contiene los nombres de los algoritmos de hash que están disponibles en el intérprete de Python en ejecución. Estos nombres serán reconocidos cuando sean pasados a `new()`. `algorithms_guaranteed` siempre será un subconjunto. El mismo algoritmo puede aparecer múltiples veces en este conjunto bajo diferentes nombres (gracias a OpenSSL).

Nuevo en la versión 3.2.

Los siguientes valores son provistos como atributos constantes de los objetos hash retornados por los constructores:

`hash.digest_size`

El tamaño del hash resultante en bytes.

`hash.block_size`

El tamaño del bloque interno del algoritmo de hash en bytes.

Un objeto hash tiene los siguientes atributos:

`hash.name`

El nombre canónico de este hash, siempre en minúsculas y siempre adecuado como un parámetro a `new()` para crear otro hash de este tipo.

Distinto en la versión 3.4: El atributo *name* ha estado presente en CPython desde su inicio, pero desde Python 3.4 no fue especificado formalmente, por lo que puede no existir en algunas plataformas.

Un objeto hash tiene los siguientes métodos:

`hash.update(data)`

Actualiza el objeto de hash con el *bytes-like object*. Invocaciones repetidas son equivalentes a una única invocación con la concatenación de todos los argumentos: `m.update(a)` ; `m.update(b)` es equivalente a `m.update(a+b)`.

Distinto en la versión 3.1: El GIL de Python es liberado para permitir a otros hilos ejecutarse mientras ocurren actualizaciones de hash en datos con tamaños superiores a 2047 bytes cuando se usan algoritmos de hash suministrados por OpenSSL.

`hash.digest()`

Retorna el resumen de los datos pasados al método `update()` hasta el momento. Este es un objeto de bytes de tamaño *digest_size* el cual puede contener bytes en el rango completo desde 0 a 255.

`hash.hexdigest()`

Como `digest()` excepto que el resumen es retornado como un objeto de cadena del doble de largo, conteniendo sólo dígitos hexadecimales. Este puede ser usado para intercambiar el valor de forma segura en correos electrónicos u otros entornos no binarios.

`hash.copy()`

Retorna una copia («clon») del objeto hash. Este puede ser usado para calcular eficientemente los resúmenes de datos compartiendo una subcadena inicial común.

15.1.2 Resúmenes SHAKE de largo variable

Los algoritmos `shake_128()` y `shake_256()` proveen resúmenes de largo variable con `largo_en_bits//2` hasta 128 ó 256 bits de seguridad. Como tales, sus métodos de resumen requieren un largo. El largo máximo no está limitado por el algoritmo SHAKE.

`shake.digest(length)`

Retorna el resumen de los datos pasados al método `update()` hasta el momento. Este es un objeto de bytes de tamaño *length* el cual puede contener bytes en el rango completo desde 0 a 255.

`shake.hexdigest(length)`

Como `digest()` excepto que el resumen es retornado como un objeto de cadena del doble de largo, conteniendo sólo dígitos hexadecimales. Este puede ser usado para intercambiar el valor de forma segura en correos electrónicos u otros entornos no binarios.

15.1.3 Derivación de clave

Los algoritmos de derivación de clave y estiramiento de clave están diseñados para el cifrado seguro de contraseña. Algoritmos ingenuos como `sha1(password)` no son resistentes contra ataques de fuerza bruta. Una buena función hash de contraseña debe ser afinable, lenta e incluir una *sal*.

`hashlib.pbkdf2_hmac` (*hash_name*, *password*, *salt*, *iterations*, *dklen=None*)

La función provee contraseñas PKCS#5 basadas en función de derivación de clave 2. Usa HMAC como función de pseudoaleatoriedad.

La cadena *hash_name* es el nombre deseado del algoritmo de resumen de hash para HMAC, ej. “sha1” o “sha256”. *password* y *salt* son interpretados como búferes de bytes. Aplicaciones y bibliotecas deberían limitar *password* a un largo razonable (ej. 1024). *salt* debería ser sobre 16 o más bytes desde una fuente adecuada, ej. `os.urandom()`.

El número de *iterations* debería ser elegido basado en el algoritmo de hash y el poder de cómputo. A partir del 2013, se sugiere al menos 100,000 iteraciones de SHA-256.

dklen es el largo de la clave derivada. Si *dklen* es `None` entonces el tamaño de resumen del algoritmo de hash *hash_name* es usado, ej. 64 para SHA-512.

```
>>> import hashlib
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
>>> dk.hex()
'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f9d79a5'
```

Nuevo en la versión 3.4.

Nota: Una implementación rápida de *pbkdf2_hmac* está disponible con OpenSSL. La implementación Python usa una versión en línea de *hmac*. Es aproximadamente tres veces más lenta y no libera el GIL.

`hashlib.scrypt` (*password*, *, *salt*, *n*, *r*, *p*, *maxmem=0*, *dklen=64*)

La función provee una contraseña *scrypt* basada en una función derivación de clave como es definida en **RFC 7914**.

password y *salt* deben ser *objetos de bytes*. Aplicaciones y bibliotecas deberían limitar *password* a un largo razonable (ej. 1024). *salt* debería ser aproximadamente 16 o más bytes de una fuente adecuada, ej. `os.urandom()`.

n es el factor de coste de CPU/Memoria, *r* el tamaño de bloque, *p* el factor de paralelización y *maxmem* limita la memoria (OpenSSL 1.1.0 por defecto a 32 MiB). *dklen* es el largo de la clave derivada.

Disponibilidad: OpenSSL 1.1+.

Nuevo en la versión 3.6.

15.1.4 BLAKE2

BLAKE2 es una función de hash criptográfico definida en **RFC 7693** que viene en dos sabores:

- **BLAKE2b**, optimizada para plataformas de 64 bits y produce resúmenes de cualquier tamaño entre 1 y 64 bytes,
- **BLAKE2s**, optimizada para plataformas de 8 a 32 bits y produce resúmenes de cualquier tamaño entre 1 y 32 bytes.

BLAKE2 proporciona el **modo keyed** (un remplazamiento más simple rápido para **HMAC**), **cifrado salado** (*salted hashing*), **personalización** y **cifrado de árbol**.

Los objetos hash de este módulo siguen los estándares de los objetos de la biblioteca *hashlib*.

Creando objetos hash

Se crean nuevos objetos hash invocando a las funciones de constructor:

```
hashlib.blake2b(data=b", *, digest_size=64, key=b", salt=b", person=b", fanout=1, depth=1,
                leaf_size=0, node_offset=0, node_depth=0, inner_size=0, last_node=False)
```

```
hashlib.blake2s(data=b", *, digest_size=32, key=b", salt=b", person=b", fanout=1, depth=1,
                leaf_size=0, node_offset=0, node_depth=0, inner_size=0, last_node=False)
```

Estas funciones retornan los objetos hash correspondientes para calcular BLAKE2b o BLAKE2s. Ellas toman opcionalmente estos parámetros generales:

- *data*: trozo inicial de datos a cifrar, el cual debe ser un *bytes-like object*. Puede ser pasado sólo como argumento posicional.
- *digest_size*: tamaño del resumen de salida en bytes.
- *key*: clave para el cifrado de clave (*keyed hashing*) (hasta 64 bytes para BLAKE2b, hasta 32 bytes para BLAKE2s).
- *salt*: sal para el cifrado aleatorio (hasta 16 bytes para BLAKE2b, hasta 8 bytes para BLAKE2s).
- *person*: cadena de personalización (hasta 16 bytes para BLAKE2b, hasta 8 bytes para BLAKE2s).

La siguiente tabla muestra los límites para parámetros generales (en bytes):

Cifrado	digest_size	len(key)	len(salt)	len(person)
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

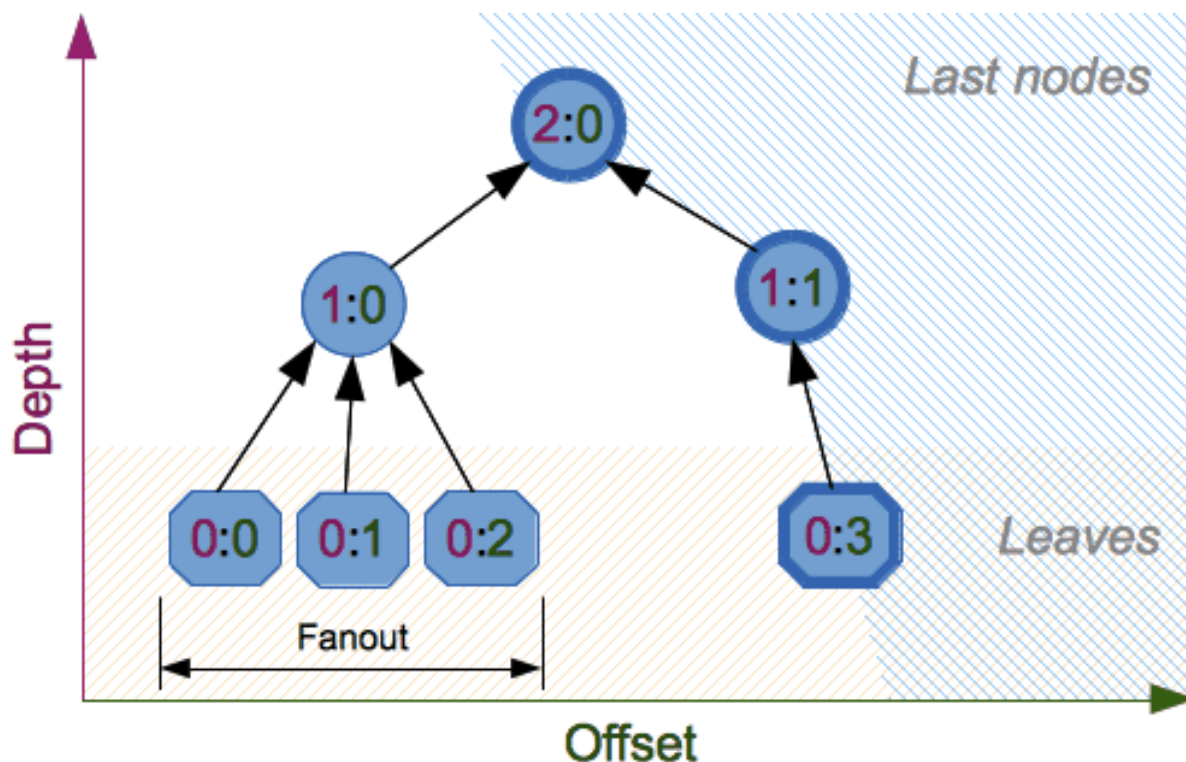
Nota: La especificación BLAKE2 define largos constantes para los parámetros de sal y personalización, sin embargo, por conveniencia, esta implementación acepta cadenas de bytes de cualquier tamaño hasta el largo especificado. Si el largo del parámetro es menor que el especificado, es acolchado con ceros, por lo tanto, por ejemplo, `b'salt'` y `b'salt\x00'` es el mismo valor. (Este no es el caso para *key*.)

Estos tamaños están disponibles como *constantes* del módulo descritas abajo.

Las funciones constructoras también aceptan los siguientes parámetros de cifrado de árbol:

- *fanout*: despliegue en abanico (0 a 255, 0 si ilimitado, 1 en modo secuencial).
- *depth*: profundidad máxima del árbol (1 a 255, 255 si ilimitado, 1 en modo secuencial).
- *leaf_size*: tamaño máximo en bytes de hoja (0 a $2^{32}-1$, 0 si ilimitado o en modo secuencial).
- *node_offset*: desplazamiento del nodo (0 a $2^{64}-1$ para BLAKE2b, 0 a $2^{48}-1$ para BLAKE2s, 0 para la primera, la hoja más a la izquierda o en modo secuencial).
- *node_depth*: profundidad de nodo (0 a 255, 0 para hojas o en modo secuencial).
- *inner_size*: tamaño interno del resumen (0 a 64 para BLAKE2b, 0 a 32 para BLAKE2s, 0 en modo secuencial).
- *last_node*: booleano indicando si el nodo procesado es el último (*False* para modo secuencial).

Consulta la sección 2.10 en la *especificación BLAKE2* <https://blake2.net/blake2_20130129.pdf> para una revisión integral del cifrado en árbol.



Constantes

`blake2b.SALT_SIZE`

`blake2s.SALT_SIZE`

Largo de sal (largo máximo aceptado por los constructores).

`blake2b.PERSON_SIZE`

`blake2s.PERSON_SIZE`

Largo de cadena de personalización (largo máximo aceptado por los constructores).

`blake2b.MAX_KEY_SIZE`

`blake2s.MAX_KEY_SIZE`

Tamaño máximo de clave.

`blake2b.MAX_DIGEST_SIZE`

`blake2s.MAX_DIGEST_SIZE`

Tamaño máximo de resumen que puede producir la función hash.

Ejemplos

Cifrado simple

Para calcular el hash de algunos datos, primero debes construir un objeto hash invocando a la función del constructor apropiada (`blake2b()` o `blake2s()`), entonces actualizarlo con los datos invocando `update()` en el objeto y, finalmente, obtener el resumen del objeto invocando `digest()` (o `hexdigest()` para una cadena codificada en hexadecimal).

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()

↪ '6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb4'
↪ '
```

Como atajo, puedes pasar el primer trozo de datos para actualizar directamente el constructor como el argumento posicional:

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()

↪ '6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb4'
↪ '
```

Puedes invocar `hash.update()` tantas veces como necesites para actualizar el hash iterativamente:

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
>>> h.hexdigest()

↪ '6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb4'
↪ '
```

Usar diferentes tamaños de resumen

BLAKE2 tiene tamaño de resúmenes configurables de hasta 64 bytes para BLAKE2b y 32 bytes para BLAKE2s. Por ejemplo, para remplazar SHA-1 con BLAKE2b sin cambiar el tamaño de la salida, puedes decirle a BLAKE2b que produzca resúmenes de 20 bytes:

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

Objetos hash con diferentes tamaños de resumen tienen salidas completamente diferentes (hashes más cortos *no* son prefijos de hashes más largos); BLAKE2b y BLAKE2s producen salidas diferentes incluso si el largo de salida es el mismo:

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

Cifrado de clave

El cifrado de clave puede ser usado para autenticación como remplazo más rápido y simple para [Código de autenticación de mensajes en clave-hash](#) (HMAC). BLAKE2 puede ser usado de forma segura en modo de prefijo MAC gracias a la propiedad de indiferenciabilidad heredada de BLAKE.

Este ejemplo muestra como obtener un código de autenticación (codificado como hexadecimal) de 128 bits para el mensaje `b'message data'` con la clave `b'pseudorandom key'`:

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

Como ejemplo práctico, una aplicación web puede firmar simétricamente cookies enviadas a los usuarios y verificarlas más tarde para asegurar que no fueron manipuladas con:

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
>>>
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

Incluso aunque hay un modo de cifrado de claves nativo, BLAKE2 puede, por supuesto, ser usado en construcción de HMAC con el módulo `hmac`:

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

Cifrado aleatorio

Definiendo el parámetro *salt* los usuarios pueden introducir aleatoriedad a la función hash. El cifrado aleatorio es útil para proteger contra ataques de colisión en la función hash usada en firmas digitales.

El cifrado aleatorio está diseñado para situaciones en las que una parte, el preparador del mensaje, genera todo o parte de un mensaje para ser firmado por una segunda parte, el firmante del mensaje. Si el preparador del mensaje es capaz de encontrar colisiones de funciones hash criptográficas (ej., dos mensajes produciendo el mismo valor de hash), entonces ellos pueden preparar versiones significativas del mensaje que producirían el mismo valor de hash y firma digital, pero con diferentes resultados (ej., transfiriendo 1,000,000 \$ a una cuenta, en lugar de 10 \$). Las funciones de hash criptográfico han sido diseñadas con resistencia de colisión como objetivo principal, pero la concentración actual en el ataque a las funciones hash criptográficas puede resultar en una función hash criptográfica dada que provea menor resistencia de colisión de la esperada. El cifrado aleatorio ofrece al firmante protección adicional reduciendo la probabilidad de que un preparador puede generar dos o más mensajes que en última instancia producen el mismo valor hash durante el proceso de generación de la firma digital, — incluso si es práctico encontrar colisiones para la función hash. Sin embargo, el uso de cifrado aleatorio puede reducir la cantidad de seguridad provista por una firma digital cuando todas las porciones del mensaje son preparadas por el firmante.

(NIST SP-800-106 «Randomized Hashing for Digital Signatures»)

En BLAKE2 la sal es procesada como una entrada de una vez a la función hash durante la inicialización, en lugar de como una entrada para cada función de compresión.

Advertencia: El *cifrado salado* (o sólo cifrado) con BLAKE2 o cualquier otra función de hash criptográfico de propósito general, como SHA-256, no son aptas para cifrar contraseñas. Ver [BLAKE2 FAQ](#) para más información.

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

Personalización

A veces es útil forzar a la función hash para producir diferentes resúmenes para la misma entrada para diferentes propósitos. Citando a los autores de la función hash Skein:

Recomendamos que todos los diseñadores de aplicaciones consideren seriamente hacer esto; hemos visto muchos protocolos donde un hash que es calculado en una parte del protocolo puede ser usado en una parte completamente diferente porque dos cálculos hash fueron realizados en datos similares o relacionados, y el atacante puede forzar a la aplicación a hacer las entradas hash iguales. Personalizar cada función hash usada en el protocolo resumidamente detiene este tipo de ataque.

(The Skein Hash Function Family, p. 21)

BLAKE2 puede ser personalizado pasando bytes al argumento *person*:

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383a3'
```

Se puede usar también personalización en conjunto con el modo de clave para derivar diferentes claves desde una sola.

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21KM=')
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPlYk1e/nWfu0WSEb0KRcjhDeP/o=
```

Modo de árbol

Aquí hay un ejemplo de cifrar un árbol mínimo con dos nodos de hoja:

```
  10
 /  \
00  01
```

Este ejemplo usa resúmenes internos de 64 bytes, y retorna el resumen final de 32 bytes:

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
>>> buf = bytearray(6000)
>>>
>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=0, last_node=False)
>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=1, node_depth=0, last_node=True)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```

Créditos

BLAKE2 fue diseñado por *Jean-Philippe Aumasson*, *Samuel Neves*, *Zooko Wilcox-O'Hearn* y *Christian Winnerlein* basado en el **SHA-3** finalista **BLAKE** creado por *Jean-Philippe Aumasson*, *Luca Henzen*, *Willi Meier* y *Raphael C.-W. Phan*.

Usa el algoritmo núcleo del cifrado **ChaCha** diseñado por *Daniel J. Bernstein*.

La implementación stdlib está basada en el módulo **pyblake2**. Fue escrita por *Dmitry Chestnykh* basada en la implementación C escrita por *Samuel Neves*. La documentación fue copiada desde **pyblake2** y escrita por *Dmitry Chestnykh*.

El código C fue parcialmente reescrito para Python por *Christian Heimes*.

La siguiente dedicación de dominio público aplica tanto para la implementación de la función hash C, el código de extensión y su documentación:

En la medida en que la ley lo permite, el/los autor/es han dedicado todos los derechos de autor y los derechos relacionados y vecinos de este software al dominio público mundial. Este software se distribuye sin ninguna garantía.

Deberías haber recibido una copia de la Dedicación CC0 de Dominio Público junto a este software. Si no, consulta <https://creativecommons.org/publicdomain/zero/1.0/>.

Las siguientes personas han ayudado con el desarrollo o contribuyeron con sus cambios al proyecto y el dominio público de acuerdo a Creative Commons Public Domain Dedication 1.0 Universal:

- *Alexandr Sokolovskiy*

Ver también:

Módulo *hmac* Un módulo para generar mensajes de códigos de autenticación usando hashes.

Módulo *base64* Otra forma de codificar hashes binarios para entornos no binarios.

<https://blake2.net> Sitio web oficial de BLAKE2.

<https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf> La publicación FIPS 180-2 sobre Algoritmos de Cifrado Seguros.

https://en.wikipedia.org/wiki/Cryptographic_hash_function#Cryptographic_hash_algorithms Artículo de Wikipedia con información sobre cuáles algoritmos tienen errores conocidos y lo que eso significa con respecto a su uso.

<https://www.ietf.org/rfc/rfc2898.txt> PKCS #5: Password-Based Cryptography Specification Version 2.0

15.2 hmac — Hash con clave para autenticación de mensajes

Código fuente: `Lib/hmac.py`

Este módulo implementa el algoritmo HMAC como se describe en la **RFC 2104**.

`hmac.new(key, msg=None, digestmod=)`

Retorna un nuevo objeto `hmac`. `key` es un objeto `bytes` o `bytearray` que proporciona la clave secreta. Si `msg` está presente, se realiza la llamada al método `update(msg)`. `digestmod` es el nombre del resumen, constructor o módulo del resumen para el objeto HMAC que se va a usar. Puede ser cualquier nombre adecuado para `hashlib.new()`. Se requiere este argumento a pesar de su posición.

Distinto en la versión 3.4: El parámetro `key` puede ser un objeto `bytes` o `bytearray`. El parámetro `msg` puede ser de cualquier tipo soportado por `hashlib`. El parámetro `digestmod` puede ser el nombre del algoritmo de `hash`.

Deprecated since version 3.4, removed in version 3.8: MD5 como resumen por defecto implícito para `digestmod` está obsoleto. Ahora se requiere el parámetro `digestmod`. Páselo como un argumento de palabra clave para evitar dificultades cuando no tiene un `msg` inicial.

`hmac.digest (key, msg, digest)`

Retorna el resumen de *msg* para una clave *key* secreta y un resumen *digest* dados. La función es equivalente a `HMAC(key, msg, digest).digest()`, pero utiliza una implementación optimizada en C o *inline*, que es más rápida para mensajes que caben en memoria. Los parámetros *key*, *msg* y *digest* tienen el mismo significado que en `new()`.

Un detalle de la implementación de CPython: la implementación optimizada en C solo se usa cuando *digest* es una cadena de caracteres y el nombre de un algoritmo de resumen, que está soportado por OpenSSL.

Nuevo en la versión 3.7.

Un objeto HMAC tiene los siguientes métodos:

`HMAC.update (msg)`

Actualiza el objeto hmac con *msg*. Las llamadas repetidas equivalen a una sola llamada con la concatenación de todos los argumentos: `m.update(a) ; m.update(b)` es equivalente a `m.update(a + b)`.

Distinto en la versión 3.4: El parámetro *msg* puede ser de cualquier tipo soportado por *hashlib*.

`HMAC.digest ()`

Retorna el resumen de los *bytes* que se pasaron al método `update()` hasta el momento. Este objeto *bytes* será de la misma longitud que el *digest_size* del resumen que se pasa al constructor. Puede contener *bytes* no ASCII, incluyendo *bytes* NUL.

Advertencia: Cuando se compara la salida de `digest()` a un resumen provisto externamente durante una rutina de verificación, se recomienda utilizar la función `compare_digest()` en lugar del operador `==` para reducir la vulnerabilidad a ataques de temporización.

`HMAC.hexdigest ()`

Como `digest()` excepto que el resumen se retorna como una cadena de caracteres de dos veces la longitud conteniendo solo dígitos hexadecimales. Esto se puede utilizar para intercambiar el valor de forma segura en email u otros entornos no binarios.

Advertencia: Cuando se compara la salida de `hexdigest()` a un resumen provisto externamente durante una rutina de verificación, se recomienda utilizar la función `compare_digest()` en lugar del operador `==` para reducir la vulnerabilidad a ataques de temporización.

`HMAC.copy ()`

Retorna una copia («clon») del objeto hmac. Esto se puede utilizar para calcular de forma eficiente los resúmenes de las cadenas de caracteres que comparten una subcadena de caracteres inicial común.

Un objeto *hash* tiene los siguientes atributos:

`HMAC.digest_size`

El tamaño del resumen HMAC resultante en *bytes*.

`HMAC.block_size`

El tamaño de bloque interno del algoritmo de *hash* en *bytes*.

Nuevo en la versión 3.4.

`HMAC.name`

El nombre canónico de este HMAC, siempre en minúsculas, por ejemplo `hmac-md5`.

Nuevo en la versión 3.4.

Este módulo también provee las siguiente funciones auxiliares:

`hmac.compare_digest (a, b)`

Retorna `a == b`. Esta función utiliza un enfoque diseñado para prevenir el análisis de temporización evitando el comportamiento de cortocircuito basado en contenido, haciéndolo adecuado para criptografía. *a* y *b* deben

ser del mismo tipo: ya sea *str* (solo ASCII, como por ejemplo retornado por `HMAC.hexdigest()`), o un *objeto tipo binario*.

Nota: Si *a* y *b* son de diferente longitud, o si ocurre un error, un ataque de temporización teóricamente podría revelar información sobre los tipos y longitudes de *a* y *b*—pero no sus valores.

Nuevo en la versión 3.3.

Ver también:

Módulo `hashlib` El módulo de Python que provee funciones de *hash* seguras.

15.3 `secrets` — Genera números aleatorios seguros para trabajar con secretos criptográficos

Nuevo en la versión 3.6.

Código fuente: [Lib/secrets.py](#)

El módulo `secrets` se usa para generar números aleatorios criptográficamente fuertes, apropiados para trabajar con datos como contraseñas, autenticación de cuentas, tokens de seguridad y secretos relacionados.

In particular, `secrets` should be used in preference to the default pseudo-random number generator in the `random` module, which is designed for modelling and simulation, not security or cryptography.

Ver también:

PEP 506

15.3.1 Números aleatorios

El módulo `secrets` provee acceso a la fuente más segura de aleatoriedad que proporciona su sistema operativo.

class `secrets.SystemRandom`

Una clase para generar números aleatorios utilizando las fuentes de mayor calidad que proporciona el sistema operativo. Ver `random.SystemRandom` para más detalles.

`secrets.choice(sequence)`

Retorna un elemento aleatorio de una secuencia no vacía.

`secrets.randbelow(n)`

Retorna un entero aleatorio en el rango $[0, n)$.

`secrets.randbits(k)`

Retorna un entero con *k* bits aleatorios.

15.3.2 Generando tokens

El módulo `secrets` provee funciones para generar tokens seguros, adecuados para aplicaciones como el restablecimiento de contraseñas, URLs difíciles de adivinar, y similares.

`secrets.token_bytes([nbytes=None])`

Retorna una cadena de bytes aleatorios que contiene *nbytes* número de bytes. Si *nbytes* es `None` o no se suministra, se utiliza un valor por defecto razonable.

```
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

`secrets.token_hex([nbytes=None])`

Retorna una cadena de caracteres aleatoria, en hexadecimal. La cadena de caracteres tiene *nbytes* bytes aleatorios, cada byte convertido a dos dígitos hexadecimales. Si *nbytes* es `None` o no se suministra, se utiliza un valor por defecto razonable.

```
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

`secrets.token_urlsafe([nbytes=None])`

Retorna una cadena de caracteres aleatoria para usarse en URLs, que contiene *nbytes* bytes aleatorios. El texto está codificado en Base64, por lo que en promedio cada byte resulta en aproximadamente 1,3 caracteres. Si *nbytes* es `None` o no se suministra, se utiliza un valor por defecto razonable.

```
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

¿Cuántos bytes deben tener los tokens?

Para estar seguros contra los [ataques de fuerza bruta](#), los tokens deben tener suficiente aleatoriedad. Desafortunadamente, lo que es considerado suficiente necesariamente aumentará a medida que las computadoras se vuelvan más potentes y capaces de hacer más pruebas en un período más corto. Desde 2015 se cree que 32 bytes (256 bits) de aleatoriedad son considerados suficientes para el típico caso de uso del módulo `secrets`.

Para quienes quieran gestionar la longitud de sus propios tokens, pueden especificar explícitamente cuánta aleatoriedad se utiliza para los tokens dando un argumento `int` a las funciones `token_*`. Ese argumento se toma como el número de bytes de aleatoriedad a utilizar.

En caso contrario, si no se proporciona ningún argumento, o si el argumento es `None`, las funciones `token_*` utilizarán en su lugar un valor por defecto razonable.

Nota: El valor por defecto está sujeto a cambios en cualquier momento, incluso en los lanzamientos de mantenimiento.

15.3.3 Otras funciones

`secrets.compare_digest(a, b)`

Retorna `True` si las cadenas de caracteres *a* y *b* son iguales, de lo contrario, `False`, de forma tal que se reduzca el riesgo de [ataques de análisis temporal](#). Ver `hmac.compare_digest`()` para detalles adicionales.

15.3.4 Recetas y mejores prácticas

Esta sección muestra las recetas y las mejores prácticas para usar `secrets` para conseguir un nivel mínimo de seguridad.

Generar una contraseña alfanumérica de ocho caracteres:

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
password = ''.join(secrets.choice(alphabet) for i in range(8))
```

Nota: Las aplicaciones no deben [almacenar contraseñas en un formato recuperable](#), ya sea en texto plano o encriptado. Deberían ser saladas y hasheadas usando una función hash unidireccional (irreversible) y criptográficamente fuerte.

Generar una contraseña alfanumérica de diez caracteres con al menos un carácter en minúscula, al menos un carácter en mayúscula y al menos tres dígitos:

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(secrets.choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
        and any(c.isupper() for c in password)
        and sum(c.isdigit() for c in password) >= 3):
        break
```

Generar una contraseña al estilo XKCD:

```
import secrets
# On standard Linux systems, use a convenient dictionary file.
# Other platforms may need to provide their own word-list.
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(secrets.choice(words) for i in range(4))
```

Generar una URL temporal difícil de adivinar que contenga un token de seguridad adecuado para la recuperación de contraseñas:

```
import secrets
url = 'https://mydomain.com/reset=' + secrets.token_urlsafe()
```

Servicios genéricos del sistema operativo

Los módulos descritos en este capítulo proporcionan interfaces a las características del sistema operativo que están disponibles en (casi) todos los sistemas operativos, como archivos y un reloj. Las interfaces se modelan, por norma general, según las interfaces Unix o C, pero también están disponibles en la mayoría de los otros sistemas. Esta es una visión general:

16.1 `os` — Interfaces misceláneas del sistema operativo

Código fuente: [Lib/os.py](#)

Este módulo provee una manera versátil de usar funcionalidades dependientes del sistema operativo. Si quieres leer o escribir un archivo mira `open()`, si quieres manipular rutas, mira el módulo `os.path`, y si quieres leer todas las líneas en todos los archivos en la línea de comandos mira el módulo `fileinput`. Para crear archivos temporales y directorios mira el módulo `tempfile`, y para el manejo de alto nivel de archivos y directorios puedes ver el módulo `shutil`.

Notas sobre la disponibilidad de estas funciones:

- El diseño de todos los módulos incorporados de Python dependientes del sistema operativo es tal que, mientras funcionalidad esté disponible, usará la misma interfaz; por ejemplo, la función `os.stat(path)` retorna estadísticas sobre la ruta (*path*) en el mismo formato (lo que sucede originalmente con la interfaz POSIX).
- Las extensiones propias de un sistema operativo en particular también están disponibles a través del módulo `os`, pero usarlas, por supuesto, es un riesgo a la portabilidad.
- Todas las funciones que aceptan rutas o nombres de archivos aceptan *bytes* o cadenas de texto, y el resultado es un objeto del mismo tipo, siempre que se retorna una ruta o un archivo.
- En VxWorks, no están soportados `os.fork`, `os.execv` y `os.spawn*p*`.

Nota: Todas las funciones en este módulo lanzan `OSError` (o subclases), en el caso de archivos o rutas inaccesibles o inválidas, u otros argumentos que tienen el tipo correcto, pero que no son aceptados por el sistema operativo.

exception `os.error`

Un alias de la excepción incorporada `OSError`.

os.name

El nombre del módulo dependiente del sistema operativo importado. Los siguientes nombres están registrados: 'posix', 'nt', 'java'.

Ver también:

`sys.platform` tiene un mayor nivel de detalle. `os.uname()` proporciona información de la versión dependiendo del sistema operativo.

El módulo `platform` proporciona verificaciones detalladas de la identidad del sistema.

16.1.1 Nombres de archivos, argumentos de la línea de comandos y variables de entorno

En Python, los nombres de archivo, los argumentos de la línea de comandos y las variables de entorno están representados usando cadena de caracteres. En algunos sistemas, decodificar esas cadenas desde y hacia *bytes* es necesario para pasárselos al sistema operativo. Python usa la codificación del sistema operativo para realizar esta conversión (ver `sys.getfilesystemencoding()`).

Distinto en la versión 3.1: En algunos sistemas, la conversión usando la codificación del sistema de archivos puede fallar. En este caso, Python usa el *controlador de error de codificación de *subrogateescape**, lo que significa que los *bytes* no codificables se reemplazan por un carácter Unicode U + DCxx en la decodificación, y estos se traducen nuevamente al byte original en la codificación.

La codificación del sistema de archivos debe garantizar la decodificación exitosa de todos los *bytes* por debajo de 128. Si la codificación del sistema de archivos no proporciona esta garantía, las funciones de la API pueden generar errores Unicode.

16.1.2 Parámetros de proceso

Estas funciones y elementos de datos proporcionan información y operan en el proceso y con el usuario actuales.

os.ctermid()

Retorna el nombre del archivo correspondiente al terminal que controla el proceso.

Disponibilidad: Unix.

os.environ

Un objeto *mapeado* que representa el entorno en cadenas de texto. Por ejemplo, `environ['HOME']` es la ruta de tu directorio personal (en algunas plataformas), y es equivalente a `getenv("HOME")` en C.

Este mapeo se captura la primera vez que se importa el módulo `os`, típicamente durante el inicio de Python como parte de procesar `site.py`. Los cambios realizados en el ambiente luego de este primer momento no se ven reflejados en `os.environ`, exceptuando aquellos que se realizan modificando directamente a `os.environ`.

Si la plataforma soporta la función `putenv()`, este mapeo se puede usar para modificar el entorno como también para consultarlo. La función `putenv()` será llamada automáticamente cuando este mapeo sea modificado.

En Unix, claves y valores usan la función `sys.getfilesystemencoding()` y el controlador de errores 'surrogateescape'. Hay que utilizar `environb` si se quiere usar una codificación diferente.

Nota: Llamar directamente a la función `putenv()` no cambia a `os.environ`, así que es mejor modificar `os.environ`.

Nota: En algunas plataformas, como FreeBSD y Mac OS X, establece `environ` pueden generar pérdidas de memoria. Hay que referirse a la documentación del sistema para la función `putenv()`.

Si la función `putenv()` no está provista, una copia modificada de este mapeo se puede pasarse a las funciones adecuadas de creación de procesos para generar que los procesos hijos usen un entorno modificado.

Si la plataforma suporta la función `unsetenv()`, se pueden eliminar elementos de este mapeo para quitar variables de entorno. Se va a llamar automáticamente a `unsetenv()` cuando un elemento sea eliminado de `os.environ`, así como también cuando se llamen a los métodos `pop()` o `clear()`.

`os.environb`

Versión en *bytes* de `environ`; un objeto *mapeado* representando el entorno como cadena de *bytes*. `environ` y `environb` están sincronizados (modificar `environb` actualiza `environ` y viceversa).

`environb` está disponible sólo si `supports_bytes_environ` está establecido en `True`.

Nuevo en la versión 3.2.

`os.chdir(path)`

`os.fchdir(fd)`

`os.getcwd()`

Estas funciones están detalladas en *Archivos y directorios*.

`os.fsencode(filename)`

Codifica un nombre de archivo *tipo ruta* con la codificación del sistema de archivos usando el controlador de errores `'surrogateescape'`, o `'strict'` en Windows; retorna *bytes* sin alterar.

`fsdecode()` es la función inversa.

Nuevo en la versión 3.2.

Distinto en la versión 3.6: Soporte agregado para aceptar objetos que implementan una interfaz `os.PathLike`.

`os.fsdecode(filename)`

Decodifica un nombre de archivo *tipo ruta* desde la codificación del sistema de archivos usando el controlador de errores `'surrogateescape'`, o `'strict'` en Windows; retorna *str* sin alterar.

`fsencode()` es la función inversa.

Nuevo en la versión 3.2.

Distinto en la versión 3.6: Soporte agregado para aceptar objetos que implementan una interfaz `os.PathLike`.

`os.fspath(path)`

Retorna la representación en el sistema de archivos de la ruta.

Si se le pasa *str* o *bytes*, retorna sin alterar. De lo contrario se llama a `__fspath__()` y se retorna su valor siempre que sea un objeto *str* o *bytes*. En los demás casos se lanza una excepción del tipo `TypeError`.

Nuevo en la versión 3.6.

`class os.PathLike`

Una *clase base abstracta* para objetos que representan una ruta del sistema de archivos, i.e. `pathlib.PurePath`.

Nuevo en la versión 3.6.

`abstractmethod __fspath__()`

Retorna la representación de la ruta del sistema de archivos del objeto.

Este método sólo retornará objetos *str* o *bytes*, preferentemente *str*.

`os.getenv(key, default=None)`

Retorna el valor de la variable de entorno especificado como clave (*key*), si existe, o *default* si no existe. *key*, *default* y el resultado son cadenas de texto.

En Unix, claves y valores se decodifican con la función `sys.getfilesystemencoding()` y con el controlador de errores `'surrogateescape'`. Usar `os.getenvb()` si se quiere usar una codificación diferente.

Disponibilidad: sistemas tipo Unix, Windows.

`os.getenv (key, default=None)`

Retorna el valor de la variable de entorno especificado como clave (*key*), si existe, o *default* si no existe. *key*, *default* y el resultado son *bytes*.

`getenv()` está disponible sólo si `supports_bytes_environ` está establecido en `True`.

Disponibilidad: sistemas tipo Unix.

Nuevo en la versión 3.2.

`os.get_exec_path (env=None)`

Retorna una lista de directorios en la que se buscará un ejecutable, similar a una *shell*, cuando se ejecuta un proceso. *env*, cuando se especifica, tienen que ser un diccionario de variables de entorno donde buscar el *PATH*. Por defecto cuando *env* es `None`, se usa `environ`.

Nuevo en la versión 3.2.

`os.getegid ()`

Retorna el *id* del grupo (*gid*) efectivo correspondiente al proceso que se está ejecutando. Esto corresponde al bit de «*set id*» en el archivo que se está ejecutando en el proceso actual.

Disponibilidad: Unix.

`os.geteuid ()`

Retorna el *id* del usuario correspondiente al proceso que se está ejecutando actualmente.

Disponibilidad: Unix.

`os.getgid ()`

Retorna el *id* del grupo real correspondiente al proceso que se está ejecutando actualmente.

Disponibilidad: Unix.

`os.getgrouplist (user, group)`

Retorna la lista de *ids* de grupos al que el usuario pertenece. Si el grupo *group* no está en la lista, se incluirá; típicamente *group* se especifica como en el campo *ID* de grupo del registro de claves del usuario.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

`os.getgroups ()`

Retorna la lista de *ids* de grupos secundarios asociados con el proceso actual.

Disponibilidad: Unix.

Nota: En Mac OS X, la función `getgroups()` se comporta diferente que en otras plataformas del tipo Unix. Si el intérprete de Python se compiló con un objetivo de despliegue igual a 10.5 o anterior, la función `getgroups()` retorna la lista de *ids* de grupos efectivos asociados con el proceso actual; esta lista está limitada a un número de entradas definidas a nivel de sistema, típicamente 16, y puede modificarse con la ejecución de `setgroups()` si se tiene los privilegios adecuados. Si se compila con un objetivo de despliegue mayor que 10.5, `getgroups()` retorna la lista de acceso de grupo actual asociada para el *id* efectivo del usuario del proceso; la lista de acceso de grupo puede cambiar durante el ciclo de vida del proceso, no se ve afectada por las llamadas a `setgroups()`, y su longitud no está limitada a 16. El valor de objetivo de despliegue, `MACOSX_DEPLOYMENT_TARGET`, se puede ver con `sysconfig.get_config_var()`.

`os.getlogin ()`

Retorna el nombre del usuario que inició sesión en el terminal que controla el proceso. Para la mayoría de los casos, es más útil usar `getpass.getuser()` ya que este último verifica las variables de entorno `LOGNAME` o `USERNAME` para averiguar quién es el usuario y recurre a `pwd.getpwuid(os.getuid())[0]` para obtener el nombre de inicio de sesión del *ID* de usuario real actual.

Disponibilidad: Unix, Windows.

- `os.getpgid(pid)`
 Retorna el id del grupo de procesos del proceso con la identificación del proceso *pid*. Si *pid* es 0, se retorna la identificación del grupo de proceso del proceso actual.

Disponibilidad: Unix.
- `os.getpgrp()`
 Retorna el *id* del grupo de proceso actual.

Disponibilidad: Unix.
- `os.getpid()`
 Retorna el *id* del proceso actual.
- `os.getppid()`
 Retorna el *id* del proceso del padre. Cuando el proceso padre ha terminado, en Unix la identificación que retorna es la del proceso *init* (1), en Windows sigue siendo la misma identificación, que ya puede ser reutilizada por otro proceso.

Disponibilidad: Unix, Windows.

Distinto en la versión 3.2: Se agregó soporte para Windows.
- `os.getpriority(which, who)`
 Obtenga la prioridad del programa. El valor *which* es uno de *PRIO_PROCESS*, *PRIO_PGRP*, o *PRIO_USER*, y *who* se interpreta en relación a *which* (un identificador de proceso para *PRIO_PROCESS*, un identificador de grupo de proceso para *PRIO_PGRP*, y un ID de usuario para *PRIO_USER*). Un valor cero para *who* denota (respectivamente) el proceso llamado, el grupo de proceso del proceso llamado o el ID de usuario real del proceso llamado.

Disponibilidad: Unix.

Nuevo en la versión 3.3.
- `os.PRIO_PROCESS`
- `os.PRIO_PGRP`
- `os.PRIO_USER`
 Parámetros para las funciones *getpriority()* y *setpriority()*.

Disponibilidad: Unix.

Nuevo en la versión 3.3.
- `os.getresuid()`
 Retorna una tupla (ruid, euid, suid) que denota los ID de usuario reales, efectivos y guardados del proceso actual.

Disponibilidad: Unix.

Nuevo en la versión 3.2.
- `os.getresgid()`
 Retorna una tupla (rgid, egid, sgid) que denota los ID de grupo reales, efectivos y guardados del proceso actual.

Disponibilidad: Unix.

Nuevo en la versión 3.2.
- `os.getuid()`
 Retorna el *id* del usuario real del proceso actual.

Disponibilidad: Unix.
- `os.initgroups(username, gid)`
 Llamada al sistema *initgroups()* para inicializar la lista de acceso de grupo con todos los grupos de los que es miembro el nombre de usuario especificado, más el ID del grupo especificado.

Disponibilidad: Unix.

Nuevo en la versión 3.2.

os.putenv (*key*, *value*)

Establece la variable de entorno llamada *key* con el valor de la cadena *value*. Dichos cambios en el entorno impactan a los subprocesos iniciados con `os.system()`, `popen()` o `fork()` y `execv()`.

Disponibilidad: sistemas tipo Unix, Windows.

Nota: En algunas plataformas, incluidas FreeBSD y Mac OS X, la configuración de `environment` puede causar pérdidas de memoria. Consulta la documentación del sistema para `putenv`.

Cuando `putenv()` es compatible, las asignaciones de elementos en `os.environ` se traducen automáticamente en llamadas correspondientes a `putenv()`; sin embargo, llamar a `putenv()` no actualiza `os.environ`, por lo que es preferible asignar a elementos de `os.environ`.

Lanza un *evento de auditoría* `os.putenv` con argumentos *key*, *value*.

os.setegid (*egid*)

Establece el *id* de grupo efectivo del proceso actual.

Disponibilidad: Unix.

os.seteuid (*euid*)

Establece el *id* de usuario efectivo del proceso actual.

Disponibilidad: Unix.

os.setgid (*gid*)

Establece el *id* de grupo del proceso actual.

Disponibilidad: Unix.

os.setgroups (*groups*)

Establece la lista de *ids* de grupos secundarios asociados con el proceso actual en *groups*. *groups* debe ser una secuencia y cada elemento debe ser un número entero que identifique un grupo. Esta operación generalmente está disponible sólo para el superusuario.

Disponibilidad: Unix.

Nota: En Mac OS X, la longitud de *groups* no puede exceder el número máximo de identificadores de grupo efectivos definidos por el sistema, generalmente 16. Consulte la documentación de `getgroups()` para casos en los que no puede retornar el mismo conjunto de listas de grupos llamando a `setgroups()`.

os.setpgrp ()

Invoca a la llamada de sistema `setpgrp()` o `setpgrp(0, 0)` dependiendo de la versión que se implemente (si la hay). Vea el manual de Unix para la semántica.

Disponibilidad: Unix.

os.setpgid (*pid*, *pgrp*)

Invoca a la llamada de sistema `setpgid()` para establecer la identificación del grupo de procesos del *id* del proceso como *pid* al grupo de procesos con *id pgrp*. Vea el manual de Unix para la semántica.

Disponibilidad: Unix.

os.setpriority (*which*, *who*, *priority*)

Establecer la prioridad del programa. El valor *which* es uno de `PRIO_PROCESS`, `PRIO_PGRP`, o `PRIO_USER`, y *who* se interpreta en relación con *which* (un identificador de proceso para `PRIO_PROCESS`, un identificador de grupo de proceso para `PRIO_PGRP`, y un ID de usuario para `PRIO_USER`). Un valor cero para *who* denota (respectivamente) el proceso llamado, el grupo de procesos del proceso llamado o el ID del usuario real del proceso llamado. *priority* es un valor en el rango de -20 a 19. La prioridad predeterminada es 0; las prioridades más bajas causan una programación más favorable.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

- `os.setregid(rgid, egid)`
Establece los *ids* de grupos reales y efectivos del proceso actual.
Disponibilidad: Unix.
- `os.setresgid(rgid, egid, sgid)`
Establece los *ids* de grupo reales, efectivos y guardados del proceso actual.
Disponibilidad: Unix.
Nuevo en la versión 3.2.
- `os.setresuid(ruid, euid, suid)`
Establece los *ids* de usuario reales, efectivos y guardados del proceso actual.
Disponibilidad: Unix.
Nuevo en la versión 3.2.
- `os.setreuid(ruid, euid)`
Establece los *ids* de usuario reales y efectivos del proceso actual.
Disponibilidad: Unix.
- `os.getsid(pid)`
Invoca a la llamada de sistema `getsid()`. Vea el manual de Unix para la semántica.
Disponibilidad: Unix.
- `os.setsid()`
Invoca a la llamada de sistema `setsid()`. Vea el manual de Unix para la semántica.
Disponibilidad: Unix.
- `os.setuid(uid)`
Establece *id* del usuario del proceso actual.
Disponibilidad: Unix.
- `os.strerror(code)`
Retorna el mensaje de error correspondiente al código de error en *code*. En plataformas donde `strerror()` retorna `NULL` cuando se le da un número de error desconocido lanza un `ValueError`.
- `os.supports_bytes_environ`
`True` si el tipo de entorno nativo del sistema operativo es bytes (por ejemplo, `False` en Windows).
Nuevo en la versión 3.2.
- `os.umask(mask)`
Establece la *umask* numérica actual y retorna la *umask* anterior.
- `os.uname()`
Retorna información que identifica el sistema operativo actual. El valor retornado es un objeto con cinco atributos:
- `sysname` - nombre del sistema operativo
 - `nodename` - nombre de la máquina en la red (definida por la implementación)
 - `release` - *release* del sistema operativo
 - `version` - versión del sistema operativo
 - `machine` - identificador de hardware
- Por compatibilidad con versiones anteriores, este objeto también es iterable, se comporta como una tupla que contiene `sysname`, `nodename`, `release`, `version`, y `machine` en ese orden.
- Algunos sistemas se truncan `nodename` a 8 caracteres o al componente principal; una mejor manera de obtener el nombre de host es usar `socket.gethostname()` o incluso `socket.gethostbyaddr(socket.gethostname())`.

Disponibilidad: sistemas tipo Unix más nuevos.

Distinto en la versión 3.3: El tipo de objeto retornado cambió de una tupla a un objeto tipo tupla con atributos con nombre.

`os.unsetenv(key)`

Desestablece (elimine) la variable de entorno llamada *key*. Dichos cambios en el entorno afectan a los subprocesos iniciados con `os.system()`, `popen()` o `fork()` y `execv()`.

Cuando `unsetenv()` es compatible, la eliminación de elementos en `os.environ` se traduce automáticamente en una llamada correspondiente a `unsetenv()`; sin embargo, las llamadas a `unsetenv()` no actualizan `os.environ`, por lo que en realidad es preferible eliminar elementos de `os.environ`.

Lanza un *evento de auditoría* `os.unsetenv` con argumento *key*.

Disponibilidad: sistemas tipo Unix.

16.1.3 Creación de objetos de tipo archivo

Estas funciones crean nuevos *objetos de archivo*. (Consulte también `open()` para abrir los descriptores de archivos).

`os.fdopen(fd, *args, **kwargs)`

Retorna un objeto de archivo abierto conectado al descriptor de archivo *fd*. Este es un alias de la función incorporada `open()` y acepta los mismos argumentos. La única diferencia es que el primer argumento de `fdopen()` siempre debe ser un número entero.

16.1.4 Operaciones de descriptores de archivos

Estas funciones operan en flujos de E/S a los que se hace referencia mediante descriptores de archivo.

Los descriptores de archivo son enteros pequeños que corresponden a un archivo que ha sido abierto por el proceso actual. Por ejemplo, la entrada estándar suele ser el descriptor de archivo 0, la salida estándar es el 1 y el error estándar es el 2. A los archivos abiertos por un proceso se les asignará 3, 4, 5, y así sucesivamente. El nombre «descriptor de archivo» es ligeramente engañoso; en las plataformas Unix, los descriptores de archivo también hacen referencia a *sockets* y tuberías.

El método `fileno()` se puede utilizar para obtener el descriptor de archivo asociado con un *file object* cuando sea necesario. Tenga en cuenta que el uso del descriptor de archivo directamente omitirá los métodos de objeto de archivo, ignorando aspectos como el almacenamiento interno de datos.

`os.close(fd)`

Cierra el descriptor de archivo *fd*.

Nota: Esta función está diseñada para E/S de bajo nivel y debe aplicarse a un descriptor de archivo tal como lo retorna `os.open()` o `pipe()`. Para cerrar un «objeto de archivo» retornado por la función incorporada `open()` o por `popen()` o `fdopen()`, use el método `close()`.

`os.closerange(fd_low, fd_high)`

Cierra todos los descriptores de archivo desde *fd_low* (inclusive) hasta *fd_high* (exclusivo), ignorando los errores. Equivalente a (pero mucho más rápido que):

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

`os.copy_file_range(src, dst, count, offset_src=None, offset_dst=None)`

Copia *count* bytes del descriptor de archivo *src*, comenzando desde *offset_src*, al descriptor de archivo

dst, comenzando desde offset *offset_dst*. Si *offset_src* es *None*, entonces *src* se lee desde la posición actual; respectivamente para *offset_dst*. Los archivos señalados por *src* y *dst* deben estar en el mismo sistema de archivos; de lo contrario, se genera una *OSError* con *errno* establecido en *errno.EXDEV*.

Esta copia se realiza sin el costo adicional de transferir datos desde el kernel al espacio del usuario y luego nuevamente al kernel. También, algunos sistemas de archivos podrían implementar optimizaciones adicionales. La copia se realiza como si ambos archivos se abrieran como binarios.

El valor de retorno es la cantidad de bytes copiados. Esto podría ser menor que la cantidad solicitada.

Disponibilidad: Kernel de Linux ≥ 4.5 o glibc ≥ 2.27 .

Nuevo en la versión 3.8.

○ **os.device_encoding** (*fd*)

Retorna una cadena que describe la codificación del dispositivo asociado con *fd* si está conectado a una terminal; sino retorna *None*.

○ **os.dup** (*fd*)

Retorna un duplicado del descriptor de archivo *fd*. El nuevo descriptor de archivo es *no heredable*.

En Windows, al duplicar un flujo estándar (0: stdin, 1: stdout, 2: stderr), el nuevo descriptor de archivo es *heredable*.

Distinto en la versión 3.4: El nuevo descriptor de archivo ahora es no heredable.

○ **os.dup2** (*fd*, *fd2*, *inheritable=True*)

Duplicar el descriptor de archivo *fd* a *fd2*, cerrando el anterior si es necesario. Retorna *fd2*. El nuevo descriptor de archivo es *heredable* por defecto o no heredable si *inheritable* es *False*.

Distinto en la versión 3.4: Agrega el parámetro opcional *inheritable*.

Distinto en la versión 3.7: Retorna *fd2* en caso de éxito. Anteriormente se retornaba siempre *None*.

○ **os.fchmod** (*fd*, *mode*)

Cambia el modo del archivo dado por *fd* al modo numérico *mode*. Consulte los documentos para *chmod()* para conocer los posibles valores de *mode*. A partir de Python 3.3, esto es equivalente a *os.chmod(fd, mode)*.

Lanza un *evento de auditoría* *os.chmod* con argumentos *path*, *mode*, *dir_fd*.

Disponibilidad: Unix.

○ **os.fchown** (*fd*, *uid*, *gid*)

Cambia el propietario y el *id* del grupo del archivo proporcionado por *fd* a los numéricos dados por *uid* y *gid*. Para dejar uno de los identificadores sin cambios, configúrelo en -1. Ver *chown()*. A partir de Python 3.3, esto es equivalente a *os.chown(fd, uid, gid)*.

Lanza un *evento de auditoría* *os.chown* con argumentos *path*, *uid*, *gid*, *dir_fd*.

Disponibilidad: Unix.

○ **os.fdatasync** (*fd*)

Fuerza la escritura del archivo con el descriptor de archivo *fd* en el disco. No fuerza la actualización de meta-datos.

Disponibilidad: Unix.

Nota: Esta función no está disponible en MacOS.

○ **os.pathconf** (*fd*, *name*)

Retorna la información de configuración del sistema relevante para un archivo abierto. *name* especifica el valor de configuración para recuperar; puede ser una cadena que es el nombre de un valor de sistema definido; estos nombres se especifican en varios estándares (POSIX.1, Unix 95, Unix 98 y otros). Algunas plataformas también definen nombres adicionales. Los nombres conocidos por el sistema operativo anfitrión se dan en el diccionario *pathconf_names*. Para las variables de configuración no incluidas en esa asignación, también se acepta pasar un número entero para *name*.

Si *name* es una cadena y no se conoce, se lanza un *ValueError*. Si el sistema anfitrión no admite un valor específico para *name*, incluso si está incluido en `pathconf_names`, se genera un *OSError* con *errno.EINVAL* para el número de error.

A partir de Python 3.3, esto es equivalente a `os.pathconf(fd, name)`.

Disponibilidad: Unix.

os.fstat(*fd*)

Obtiene el estado del descriptor de archivo *fd*. Retorna un objeto *stat_result*.

A partir de Python 3.3, esto es equivalente a `os.stat(fd)`.

Ver también:

La función *stat()*.

os.fstatvfs(*fd*)

Retorna información sobre el sistema de archivos que contiene el archivo asociado con el descriptor de archivo *fd*, como *statvfs()*. A partir de Python 3.3, esto es equivalente a `os.statvfs(fd)`.

Disponibilidad: Unix.

os.fsync(*fd*)

Fuerza la escritura del archivo con el descriptor de archivo *fd* en el disco. En Unix, esto llama a la función nativa *fsync()*; en Windows, la función *MS_commit()*.

Si está comenzando con un Python almacenado en búfer *file object f*, primero haga *f.flush()*, y luego haga *os.fsync(f.fileno())*, para garantizar que todas las memorias intermedias internas asociadas con *f* se escriban en disco.

Disponibilidad: Unix, Windows.

os.ftruncate(*fd, length*)

Trunca el archivo correspondiente al descriptor de archivo *fd*, para que tenga como máximo *length* bytes de tamaño. A partir de Python 3.3, esto es equivalente a `os.truncate(fd, length)`.

Lanza un *evento de auditoría* *os.truncate* con argumentos *fd, length*.

Disponibilidad: Unix, Windows.

Distinto en la versión 3.5: Se agregó soporte para Windows

os.get_blocking(*fd*)

Obtiene el modo de bloqueo del descriptor de archivo: *False* si se establece el indicador *O_NONBLOCK*, *True* si el indicador se borra.

Consulte también *set_blocking()* y *socket.socket.setblocking()*.

Disponibilidad: Unix.

Nuevo en la versión 3.5.

os.isatty(*fd*)

retorna *True* si el descriptor de archivo *fd* está abierto y conectado a un dispositivo tipo *tty*, de lo contrario, *False*.

os.lockf(*fd, cmd, len*)

Aplique, pruebe o elimine un bloqueo POSIX en un descriptor de archivo abierto. *fd* es un descriptor de archivo abierto. *cmd* especifica el comando a usar - uno de *F_LOCK*, *F_TLOCK*, *F_ULOCK* o *F_TEST*. *len* especifica la sección del archivo a bloquear.

Lanza un *evento de auditoría* *os.lockf* con argumentos *fd, cmd, len*.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

os.F_LOCK

os.F_TLOCK

os.F_ULOCK

os.F_TEST

Indicadores que especifican qué acción tomará `lockf()`.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

os.lseek (*fd*, *pos*, *how*)

Establece la posición actual del descriptor de archivo *fd* en la posición *pos*, modificada por *how*: `SEEK_SET` o 0 para establecer la posición relativa al comienzo del archivo; `SEEK_CUR` o 1 para establecerlo en relación con la posición actual; `SEEK_END` o 2 para establecerlo en relación con el final del archivo. Retorna la nueva posición del cursor en bytes, comenzando desde el principio.

os.SEEK_SET**os.SEEK_CUR****os.SEEK_END**

Parámetros para la función `lseek()`. Sus valores son 0, 1 y 2, respectivamente.

Nuevo en la versión 3.3: Algunos sistemas operativos pueden admitir valores adicionales, como `os.SEEK_HOLE` o `os.SEEK_DATA`.

os.open (*path*, *flags*, *mode=0o777*, *, *dir_fd=None*)

Abre el archivo *path* y configura varios indicadores según *flags* y su modo según *mode*. Al calcular el modo el valor actual de *umask* se enmascara primero. Retorna el descriptor de archivo para el archivo recién abierto. El nuevo descriptor de archivo es *no heredable*.

Para una descripción de los valores de indicadores (*flags*) y modo (*mode*), consulte la documentación de tiempo de ejecución de C; los indicadores constantes de flag (como `O_RDONLY` y `O_WRONLY`) se definen en el módulo `os`. En particular, en Windows agregar `O_BINARY` es necesario para abrir archivos en modo binario.

Esta función puede admitir *rutasy relativas a descriptores de directorio* con el parámetro *dir_fd*.

Lanza un *evento de auditoría* `open` con argumentos *path*, *mode*, *flags*.

Distinto en la versión 3.4: El nuevo descriptor de archivo ahora es no heredable.

Nota: Esta función está diseñada para E/S de bajo nivel. Para un uso normal, use la función integrada `open()`, que retorna un *file object* con métodos `read()` y `write()` (y mucho mas). Para envolver un descriptor de archivo en un objeto de archivo, use `fdopen()`.

Nuevo en la versión 3.3: El argumento *dir_fd*.

Distinto en la versión 3.5: Si la llamada al sistema se interrumpe y el controlador de señal no genera una excepción, la función vuelve a intentar la llamada del sistema en lugar de generar una excepción `InterruptedError` (ver **PEP 475** para la justificación).

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

Las siguientes constantes son opciones para el parámetro *flags* de la función `open()`. Se pueden combinar con el operador OR a nivel de bit `|`. Algunos de ellas no están disponibles en todas las plataformas. Para obtener descripciones de su disponibilidad y uso, consulte la página de manual `open(2)` en Unix o *la MSDN* en Windows.

os.O_RDONLY**os.O_WRONLY****os.O_RDWR****os.O_APPEND****os.O_CREAT****os.O_EXCL****os.O_TRUNC**

Las constantes anteriores están disponibles en Unix y Windows.

os.O_DSYNC**os.O_RSYNC****os.O_SYNC**

`os.O_NDELAY`
`os.O_NONBLOCK`
`os.O_NOCTTY`
`os.O_CLOEXEC`

Las constantes anteriores sólo están disponibles en Unix.

Distinto en la versión 3.3: Se agregó la constante `O_CLOEXEC`.

`os.O_BINARY`
`os.O_NOINHERIT`
`os.O_SHORT_LIVED`
`os.O_TEMPORARY`
`os.O_RANDOM`
`os.O_SEQUENTIAL`
`os.O_TEXT`

Las constantes anteriores sólo están disponibles en Windows.

`os.O_ASYNC`
`os.O_DIRECT`
`os.O_DIRECTORY`
`os.O_NOFOLLOW`
`os.O_NOATIME`
`os.O_PATH`
`os.O_TMPFILE`
`os.O_SHLOCK`
`os.O_EXLOCK`

Las constantes anteriores son extensiones y no están presentes si no están definidas por la biblioteca de C.

Distinto en la versión 3.4: Se agrega la constante `O_PATH` en los sistemas que lo admiten. Se agrega `O_TMPFILE`, sólo disponible en Linux para el Kernel 3.11 o posterior.

`os.openpty()`

Abre un nuevo par de pseudo-terminal. Retorna un par de descriptores de archivo (`master`, `slave`); para `pty` y `tty`, respectivamente. Los nuevos descriptores de archivo son *no heredable*. Para un enfoque (ligeramente) más portátil, use el módulo `pty`.

Disponibilidad: algunos sistemas tipo Unix.

Distinto en la versión 3.4: Los nuevos descriptores de archivo ahora son no heredables.

`os.pipe()`

Crea una tubería. Retorna un par de descriptores de archivo (`r`, `w`) que se pueden usar para leer y escribir, respectivamente. El nuevo descriptor de archivo es *no heredable*.

Disponibilidad: Unix, Windows.

Distinto en la versión 3.4: Los nuevos descriptores de archivo ahora son no heredables.

`os.pipe2(flags)`

Crea una tubería con *flags* establecidas atómicamente. *flags* pueden construirse juntando uno o más de estos valores: `O_NONBLOCK`, `O_CLOEXEC` con el operador OR. Retorna un par de descriptores de archivo (`r`, `w`) que se pueden usar para leer y escribir, respectivamente.

Disponibilidad: algunos sistemas tipo Unix.

Nuevo en la versión 3.3.

`os.posix_fallocate(fd, offset, len)`

Asegura que se asigne suficiente espacio en disco para el archivo especificado por *fd* a partir de *offset* y se extiende por *len* bytes.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

○ **os.posix_fadvise** (*fd, offset, len, advice*)

Avisa una intención de acceder a los datos en un patrón específico, permitiendo así que el núcleo haga optimizaciones. El consejo se aplica a la región del archivo especificada por *fd* que comienza en *offset* y se extiende para *len* bytes. *advice* es uno de `POSIX_FADV_NORMAL`, `POSIX_FADV_SEQUENTIAL`, `POSIX_FADV_RANDOM`, `POSIX_FADV_NOREUSE`, `POSIX_FADV_WILLNEED` o `POSIX_FADV_DONTNEED`.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

- **os.POSIX_FADV_NORMAL**
- **os.POSIX_FADV_SEQUENTIAL**
- **os.POSIX_FADV_RANDOM**
- **os.POSIX_FADV_NOREUSE**
- **os.POSIX_FADV_WILLNEED**
- **os.POSIX_FADV_DONTNEED**

Indicadores que se pueden usar en *advice* en `posix_fadvise()` que especifican el patrón de acceso que es probable que se use.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

○ **os.pread** (*fd, n, offset*)

Lee como máximo *n* bytes del descriptor de archivo *fd* en una posición de *offset*, sin modificar el desplazamiento (*offset*) del archivo.

Retorna una cadena de bytes que contiene los bytes leídos. Si se alcanza el final del archivo al que hace referencia *fd*, se retorna un objeto de bytes vacío.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

○ **os.readv** (*fd, buffers, offset, flags=0*)

Lee de un descriptor de archivo *fd* en una posición de *offset* en mutable *objetos de tipo bytes buffers*, dejando el desplazamiento del archivo sin cambios. Transfiere datos a cada búfer hasta que esté lleno y luego pase al siguiente búfer en la secuencia para contener el resto de los datos.

El argumento *flags* contiene un operador de bit a bit OR de cero o más de las siguientes *flags*:

- `RWF_HIPRI`
- `RWF_NOWAIT`

Retorna el número total de bytes realmente leídos que puede ser menor que la capacidad total de todos los objetos.

El sistema operativo puede establecer un límite (`sysconf()` valor `'SC_IOV_MAX'`) en el número de búferes que se pueden usar.

Combina la funcionalidad de `os.readv()` y `os.pread()`.

Disponibilidad: Linux 2.6.30 y posterior, FreeBSD 6.0 y posterior, OpenBSD 2.7 y posterior. El uso de *flags* requiere Linux 4.6 o posterior.

Nuevo en la versión 3.7.

○ **os.RWF_NOWAIT**

No espere datos que no estén disponibles de inmediato. Si se especifica este indicador, la llamada al sistema regresará instantáneamente si tuviera que leer datos del almacenamiento de respaldo o esperar por un bloqueo.

Si algunos datos se leyeron con éxito, retornará el número de bytes leídos. Si no se leyeron bytes, retornará `-1` y establecerá *errno* en `errno.EAGAIN`.

Disponibilidad: Linux 4.14 y más nuevos.

Nuevo en la versión 3.7.

os.**RWF_HIPRI**

Alta prioridad de lectura/escritura. Permite que los sistemas de archivos basados en bloques utilicen el sondeo del dispositivo, lo que proporciona una latencia más baja, pero puede usar recursos adicionales.

Actualmente, en Linux, esta función sólo se puede usar en un descriptor de archivo abierto con el indicador `O_DIRECT`.

Disponibilidad: Linux 4.6 y más nuevos.

Nuevo en la versión 3.7.

os.**pwrite** (*fd, str, offset*)

Escribe la cadena de bytes en *str* en el descriptor de archivo *fd* en la posición *offset*, sin modificar el desplazamiento del archivo.

Retorna el número de bytes realmente escritos.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

os.**pwritev** (*fd, buffers, offset, flags=0*)

Escribe los contenidos de los *buffers* en el descriptor de archivo *fd* en un desplazamiento *offset*, dejando el desplazamiento del archivo sin cambios. *buffers* deben ser una secuencia de *objetos tipo bytes*. Los búferes se procesan en orden secuencial. Se escribe todo el contenido del primer búfer antes de pasar al segundo, y así sucesivamente.

El argumento *flags* contiene un operador de bit a bit OR de cero o más de las siguientes flags:

- `RWF_DSYNC`
- `RWF_SYNC`

Retorna el número total de bytes realmente escritos.

El sistema operativo puede establecer un límite (`sysconf()` valor '`SC_IOV_MAX`') en el número de búferes que se pueden usar.

Combina la funcionalidad de `os.writev()` y `os.pwrite()`.

Disponibilidad: Linux 2.6.30 y posterior, FreeBSD 6.0 y posterior, OpenBSD 2.7 y posterior. El uso de flags requiere Linux 4.7 o posterior.

Nuevo en la versión 3.7.

os.**RWF_DSYNC**

Proporciona un equivalente por escritura de la flag `O_DSYNC` `open(2)`. Esta flag sólo se aplica al rango de datos escrito por la llamada al sistema.

Disponibilidad: Linux 4.7 y más nuevos.

Nuevo en la versión 3.7.

os.**RWF_SYNC**

Proporciona un equivalente por escritura de la flag `O_SYNC` `open(2)`. Esta flag sólo se aplica al rango de datos escrito por la llamada al sistema.

Disponibilidad: Linux 4.7 y más nuevos.

Nuevo en la versión 3.7.

os.**read** (*fd, n*)

Lee como máximo *n* bytes del descriptor de archivo *fd*.

Retorna una cadena de bytes que contiene los bytes leídos. Si se alcanza el final del archivo al que hace referencia *fd*, se retorna un objeto de bytes vacío.

Nota: Esta función está diseñada para E/S de bajo nivel y debe aplicarse a un descriptor de archivo tal como lo retorna `os.open()` o `pipe()`. Para leer un «objeto archivo» retornado por la función incorporada `open()` o por `popen()` o `fdopen()`, o `sys.stdin`, use los métodos `read()` o `readline()`.

Distinto en la versión 3.5: Si la llamada al sistema se interrumpe y el controlador de señal no genera una excepción, la función vuelve a intentar la llamada del sistema en lugar de generar una excepción `InterruptedError` (ver [PEP 475](#) para la justificación).

os.**sendfile** (*out, in, offset, count*)

os.**sendfile** (*out, in, offset, count* [, *headers*] [, *trailers*] , *flags=0*)

Copia *count* bytes del descriptor de archivo *in* al descriptor de archivo *out* comenzando en *offset*. Retorna el número de bytes enviados. Cuando se alcanza EOF, retorna 0.

La primera notación de la función es compatible con todas las plataformas que definen `sendfile()`.

En Linux, si *offset* se da como `None`, los bytes se leen desde la posición actual de *in* y se actualiza la posición de *in*.

El segundo caso puede usarse en Mac OS X y FreeBSD donde *headers* y *trailers* son secuencias arbitrarias de búferes que se escriben antes y después de que se escriben los datos de *in*. Retorna lo mismo que el primer caso.

En Mac OS X y FreeBSD, un valor de 0 para *count* especifica enviar hasta el final de *in*.

Todas las plataformas admiten sockets como descriptor de archivo *out*, y algunas plataformas también permiten otros tipos (por ejemplo, archivo normal, tuberías).

Las aplicaciones multiplataforma no deben usar los argumentos *headers*, *trailers* y *flags*.

Disponibilidad: Unix.

Nota: Para un contenedor de alto nivel de `sendfile()`, vea `socket.socket.sendfile()`.

Nuevo en la versión 3.3.

os.**set_blocking** (*fd, blocking*)

Establece el modo de bloqueo del descriptor de archivo especificado. Establece la flag `O_NONBLOCK` si se quiere que el bloqueo sea `False`, borra la flag de lo contrario.

Consulte también `get_blocking()` y `socket.socket.setblocking()`.

Disponibilidad: Unix.

Nuevo en la versión 3.5.

os.**SF_NODISKIO**

os.**SF_MNOWAIT**

os.**SF_SYNC**

Parámetros para la función `sendfile()`, si la implementación los admite.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

os.**readv** (*fd, buffers*)

Leer desde un descriptor de archivo *fd* en una cantidad de mutable *objetos tipo bytes buffers*. Transfiere datos a cada búfer hasta que esté lleno y luego pase al siguiente búfer en la secuencia para contener el resto de los datos.

Retorna el número total de bytes realmente leídos que puede ser menor que la capacidad total de todos los objetos.

El sistema operativo puede establecer un límite (`sysconf()` valor `'SC_IOV_MAX'`) en el número de búferes que se pueden usar.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

`os.tcgetpgrp (fd)`

Retorna el grupo del proceso asociado con la terminal proporcionada por *fd* (un descriptor de archivo abierto como lo retorna `os.open()`).

Disponibilidad: Unix.

`os.tcsetpgrp (fd, pg)`

Establece el grupo del proceso asociado con la terminal dada por *fd* (un descriptor de archivo abierto como lo retorna `os.open()`) a *pg*.

Disponibilidad: Unix.

`os.ttyname (fd)`

Retorna una cadena que especifica el dispositivo de terminal asociado con el descriptor de archivo *fd*. Si *fd* no está asociado con un dispositivo de terminal, se genera una excepción.

Disponibilidad: Unix.

`os.write (fd, str)`

Escribe la cadena de bytes en *str* en el descriptor de archivo *fd*.

Retorna el número de bytes realmente escritos.

Nota: Esta función está diseñada para E/S de bajo nivel y debe aplicarse a un descriptor de archivo tal como lo retorna `os.open()` o `pipe()`. Para escribir un «objeto archivo» retornado por la función incorporada `open()` o por `popen()` o `fdopen()`, o `sys.stdout` o `sys.stderr`, use el método `write()`.

Distinto en la versión 3.5: Si la llamada al sistema se interrumpe y el controlador de señal no genera una excepción, la función vuelve a intentar la llamada del sistema en lugar de generar una excepción `InterruptedError` (ver [PEP 475](#) para la justificación).

`os.writev (fd, buffers)`

Escribe el contenido de *buffers* en el descriptor de archivo *fd*. *buffers* debe ser una secuencia de *objetos tipo bytes*. Los búferes se procesan en orden secuencial. Se escribe todo el contenido del primer búfer antes de pasar al segundo, y así sucesivamente.

Retorna el número total de bytes realmente escritos.

El sistema operativo puede establecer un límite (`sysconf()` valor `'SC_IOV_MAX'`) en el número de búferes que se pueden usar.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

Consultando las dimensiones de una terminal

Nuevo en la versión 3.3.

`os.get_terminal_size (fd=STDOUT_FILENO)`

Retorna el tamaño de la ventana de la terminal como `(columns, lines)`, una tupla del tipo `terminal_size`.

El argumento opcional *fd* (por defecto es `STDOUT_FILENO`, o la salida estándar) especifica qué descriptor de archivo debe consultarse.

Si el descriptor de archivo no está conectado a una terminal, se genera un `OSError`.

`shutil.get_terminal_size()` es la función de alto nivel que normalmente debería usarse, `os.get_terminal_size` es la implementación de bajo nivel.

Disponibilidad: Unix, Windows.

class `os.terminal_size`

Una subclase de tupla, que contiene (`columns`, `lines`) representando el tamaño de la ventana de la terminal.

columns

Ancho de la ventana de la terminal en caracteres.

lines

Alto de la ventana de la terminal en caracteres.

Herencia de los descriptores de archivos

Nuevo en la versión 3.4.

Un descriptor de archivo tiene un indicador heredable (*inheritable*) que indica si el descriptor de archivo puede ser heredado por procesos secundarios. Desde Python 3.4, los descriptores de archivo creados por Python son no heredables por defecto.

En UNIX, los descriptores de archivo no heredables se cierran en procesos hijos en la ejecución de un nuevo programa, otros descriptores de archivos sí se heredan.

En Windows, los descriptores de archivo y los identificadores no heredables se cierran en los procesos hijos, a excepción de los flujos estándar (descriptores de archivo 0, 1 y 2: `stdin`, `stdout` y `stderr`), que siempre se heredan. Usando las funciones *spawn**, todos los identificadores heredables y todos los descriptores de archivos heredables se heredan. Usando el módulo *subprocess*, todos los descriptores de archivo, excepto los flujos estándar, están cerrados, y los identificadores heredables sólo se heredan si el parámetro *close_fds* es `False`.

os.get_inheritable (*fd*)

Obtiene el indicador heredable (*inheritable*) del descriptor de archivo especificado (un valor booleano).

os.set_inheritable (*fd*, *inheritable*)

Establece el indicador heredable (*inheritable*) del descriptor de archivo especificado.

os.get_handle_inheritable (*handle*)

Obtiene el indicador heredable (*inheritable*) del identificador especificado (un valor booleano).

Disponibilidad: Windows.

os.set_handle_inheritable (*handle*, *inheritable*)

Establece el indicador heredable (*inheritable*) del identificador especificado.

Disponibilidad: Windows.

16.1.5 Archivos y directorios

En algunas plataformas Unix, muchas de estas funciones admiten una o más de estas características:

- **especificando un descriptor de archivo:** Normalmente el argumento *path* proporcionado a las funciones en el módulo *os* debe ser una cadena que especifique una ruta de archivo. Sin embargo, algunas funciones ahora aceptan alternativamente un descriptor de archivo abierto para su argumento *path*. La función actuará en el archivo al que hace referencia el descriptor. (Para los sistemas POSIX, Python llamará a la variante de la función con el prefijo *f* (por ejemplo, llamará a *fchdir* en lugar de *chdir*)).

Puede verificar si *path* se puede especificar o no como un descriptor de archivo para una función particular en su plataforma usando *os.supports_fd*. Si esta funcionalidad no está disponible, su uso generará un *NotImplementedError*.

Si la función también admite argumentos *dir_fd* o *follow_symlinks*, es un error especificar uno de esos al suministrar *path* como descriptor de archivo.

- **rutas relativas a los descriptores de directorio:** Si *dir_fd* no es `None`, debería ser un descriptor de archivo que se refiera a un directorio, y la ruta a operar debería ser relativa; entonces la ruta será relativa a ese directorio. Si la ruta es absoluta, *dir_fd* se ignora. (Para los sistemas POSIX, Python llamará a la variante de la función con un sufijo *at* y posiblemente con el prefijo *f* (por ejemplo, llamará a *faccessat* en lugar de *access*)).

Puede verificar si `dir_fd` es compatible o no para una función particular en su plataforma usando `os.supports_dir_fd`. Si no está disponible, usarlo generará un `NotImplementedError`.

- **no seguir los enlaces simbólicos:** Si `follow_symlinks` es `False`, y el último elemento de la ruta para operar es un enlace simbólico, la función operará en el enlace simbólico en lugar del archivo señalado por el enlace. (Para los sistemas POSIX, Python llamará a la variante 1 . . . de la función).

Puede verificar si `follow_symlinks` es compatible o no para una función particular en su plataforma usando `os.supports_follow_symlinks`. Si no está disponible, usarlo generará un `NotImplementedError`.

`os.access` (*path*, *mode*, *, *dir_fd=None*, *effective_ids=False*, *follow_symlinks=True*)

Use el uid/gid real para probar el acceso a *path*. Tenga en cuenta que la mayoría de las operaciones utilizarán el uid/gid efectivo, por lo tanto, esta rutina se puede usar en un entorno suid/sgid para probar si el usuario que invoca tiene el acceso especificado a *path*. *mode* debería ser `F_OK` para probar la existencia de *path*, o puede ser el OR inclusivo de uno o más de `R_OK`, `W_OK`, y `X_OK` para probar los permisos. Retorna `True` si el acceso está permitido, `False` si no. Consulte la página de manual de Unix `access` (2) para obtener más información.

Esta función puede admitir la especificación *rutas relativas a descriptores de directorio y no seguir los enlaces simbólicos* .

Si `effective_ids` es `True`, `access()` realizará sus comprobaciones de acceso utilizando el uid/gid efectivo en lugar del uid/gid real. `effective_ids` puede no ser compatible con su plataforma; puede verificar si está disponible o no usando `os.supports_effective_ids`. Si no está disponible, usarlo generará un `NotImplementedError`.

Nota: Usando `access()` para verificar si un usuario está autorizado para, por ejemplo, abrir un archivo antes de hacerlo usando `open()` crea un agujero de seguridad, porque el usuario podría explotar el breve intervalo de tiempo entre verificar y abrir el archivo para manipularlo es preferible utilizar técnicas *EAFP*. Por ejemplo:

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

está mejor escrito como:

```
try:
    fp = open("myfile")
except PermissionError:
    return "some default data"
else:
    with fp:
        return fp.read()
```

Nota: Las operaciones de E/S pueden fallar incluso cuando `access()` indica que tendrán éxito, particularmente para operaciones en sistemas de archivos de red que pueden tener una semántica de permisos más allá del modelo habitual de bits de permiso POSIX.

Distinto en la versión 3.3: Se agregaron los parámetros `dir_fd`, `effective_ids` y `follow_symlinks`.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

`os.F_OK`
`os.R_OK`
`os.W_OK`
`os.X_OK`

Valores para pasar como parámetro *mode* de `access()` para probar la existencia, legibilidad, escritura y ejecutabilidad de *path*, respectivamente.

os.chdir (*path*)

Cambia el directorio de trabajo actual a *path*.

Esta función puede soportar *especificando un descriptor de archivo*. El descriptor debe hacer referencia a un directorio abierto, no a un archivo abierto.

Esta función puede generar *OSError* y subclases como *FileNotFoundError*, *PermissionError*, y *NotADirectoryError*.

Lanza un *evento de auditoría* `os.chdir` con argumento *path*.

Nuevo en la versión 3.3: Se agregó soporte para especificar *path* como descriptor de archivo en algunas plataformas.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

os.chflags (*path*, *flags*, *, *follow_symlinks=True*)

Establece las flags del *path* a las *flags* numéricas. *flags* puede tomar una combinación (OR bit a bit) de los siguientes valores (como se define en el módulo *stat*):

- `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`

Esta función puede soportar *no seguir enlaces simbólicos*.

Lanza un *evento de auditoría* `os.chflags` con argumentos *path*, *flags*.

Disponibilidad: Unix.

Nuevo en la versión 3.3: El argumento *follow_symlinks*.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

os.chmod (*path*, *mode*, *, *dir_fd=None*, *follow_symlinks=True*)

Cambia el modo de *path* al modo numérico *mode*. *mode* puede tomar uno de los siguientes valores (como se define en el módulo *stat*) o combinaciones OR de bit a bit de ellos:

- `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`

Esta función puede soportar *especificando un descriptor de archivo, rutas relativas a los descriptores de directorio y no seguir enlaces simbólicos*.

Nota: Aunque Windows admite `chmod()`, sólo puede establecer el indicador de sólo lectura del archivo (a través de las constantes `stat.S_IWRITE` y `stat.S_IREAD` o un valor entero correspondiente). Todos los demás bits son ignorados.

Lanza un *evento de auditoría* `os.chmod` con argumentos `path`, `mode`, `dir_fd`.

Nuevo en la versión 3.3: Se agregó soporte para especificar *path* como un descriptor de archivo abierto, y los argumentos `dir_fd` y `follow_symlinks`.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

○ `os.chown(path, uid, gid, *, dir_fd=None, follow_symlinks=True)`

Cambia el propietario y el *id* del grupo de *path* a los numéricos *uid* y *gid*. Para dejar uno de los identificadores sin cambios, configúrelo en -1.

Esta función puede soportar *especificando un descriptor de archivo, rutas relativas a los descriptores de directorio y no seguir enlaces simbólicos*.

Ver `shutil.chown()` para una función de nivel superior que acepta nombres además de identificadores numéricos.

Lanza un *evento de auditoría* `os.chown` con argumentos `path`, `uid`, `gid`, `dir_fd`.

Disponibilidad: Unix.

Nuevo en la versión 3.3: Se agregó soporte para especificar *path* como un descriptor de archivo abierto, y los argumentos `dir_fd` y `follow_symlinks`.

Distinto en la versión 3.6: Admite un *objeto tipo ruta*.

○ `os.chroot(path)`

Cambia el directorio raíz del proceso actual a *path*.

Disponibilidad: Unix.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

○ `os.fchdir(fd)`

Cambia el directorio de trabajo actual al directorio representado por el descriptor de archivo *fd*. El descriptor debe hacer referencia a un directorio abierto, no a un archivo abierto. A partir de Python 3.3, esto es equivalente a `os.chdir(fd)`.

Lanza un *evento de auditoría* `os.chdir` con argumento *path*.

Disponibilidad: Unix.

os.getcwd()

Retorna una cadena que representa el directorio de trabajo actual.

os.getcwdb()

Retorna una cadena de bytes que representa el directorio de trabajo actual.

Distinto en la versión 3.8: La función ahora usa la codificación UTF-8 en Windows, en lugar de los códigos ANSI: consulte [PEP 529](#) para ver la justificación. La función ya no está en desuso en Windows.

os.lchflags(path, flags)

Establece las flags de *path* a las *flags* numéricas, como [chflags\(\)](#), pero no siga los enlaces simbólicos. A partir de Python 3.3, esto es equivalente a `os.chflags(path, flags, follow_symlinks=False)`.

Lanza un *evento de auditoría* `os.chflags` con argumentos *path*, *flags*.

Disponibilidad: Unix.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

os.lchmod(path, mode)

Cambia el modo de *path* al *mode* numérico. Si la ruta es un enlace simbólico, esto afecta al enlace simbólico en lugar del objetivo. Consulte los documentos para [chmod\(\)](#) para conocer los posibles valores de *mode*. A partir de Python 3.3, esto es equivalente a `os.chmod(path, mode, follow_symlinks=False)`.

Lanza un *evento de auditoría* `os.chmod` con argumentos *path*, *mode*, *dir_fd*.

Disponibilidad: Unix.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

os.lchown(path, uid, gid)

Cambia el propietario y la identificación del grupo de *path* a los numéricos *uid* y *gid*. Esta función no seguirá enlaces simbólicos. A partir de Python 3.3, esto es equivalente a `os.chown(path, uid, gid, follow_symlinks=False)`.

Lanza un *evento de auditoría* `os.chown` con argumentos *path*, *uid*, *gid*, *dir_fd*.

Disponibilidad: Unix.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

os.link(src, dst, *, src_dir_fd=None, dst_dir_fd=None, follow_symlinks=True)

Cree un enlace rígido que apunte a *src* llamado *dst*.

Esta función puede admitir la especificación de *src_dir_fd* o *dst_dir_fd* para proporcionar *rutas relativas a los descriptores de directorio*, y *no sigue enlaces simbólicos*.

Lanza un *evento de auditoría* `os.link` con argumentos *src*, *dst*, *src_dir_fd*, *dst_dir_fd*.

Disponibilidad: Unix, Windows.

Distinto en la versión 3.2: Se agregó soporte para Windows.

Nuevo en la versión 3.3: Se agregaron los argumentos *src_dir_fd*, *dst_dir_fd* y *follow_symlinks*.

Distinto en la versión 3.6: Acepta un *path-like object* para *src* y *dst*.

os.listdir(path='.')

Return a list containing the names of the entries in the directory given by *path*. The list is in arbitrary order, and does not include the special entries `'.'` and `'..'` even if they are present in the directory. If a file is removed from or added to the directory during the call of this function, whether a name for that file be included is unspecified.

path puede ser un *path-like object*. Si *path* es de tipo `bytes` (directa o indirectamente a través de la interfaz [PathLike](#)), los nombres de archivo retornados también serán de tipo `bytes`; en todas las demás circunstancias, serán del tipo `str`.

Esta función también puede admitir *especificando un descriptor de archivo*; el descriptor de archivo debe hacer referencia a un directorio.

Lanza un *evento de auditoría* `os.listdir` con el argumento *ruta*.

Nota: Para codificar los nombres de archivo `str` en `bytes`, use `f.encode()`.

Ver también:

La función `scandir()` retorna entradas de directorio junto con información de atributos de archivo, lo que proporciona un mejor rendimiento para muchos casos de uso comunes.

Distinto en la versión 3.2: El parámetro `path` se convirtió en opcional.

Nuevo en la versión 3.3: Se agregó soporte para especificar `path` como un descriptor de archivo abierto.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

os.**lstat** (`path`, *, `dir_fd=None`)

Realice el equivalente de una llamada al sistema `lstat()` en la ruta dada. Similar a `stat()`, pero no sigue enlaces simbólicos. Retorna un objeto `stat_result`.

En plataformas que no admiten enlaces simbólicos, este es un alias para `stat()`.

A partir de Python 3.3, esto es equivalente a `os.stat(path, dir_fd=dir_fd, follow_symlinks=False)`.

Esta función también puede admitir *rutas relativas a descriptors de directorio*.

Ver también:

La función `stat()`.

Distinto en la versión 3.2: Se agregó soporte para enlaces simbólicos de Windows 6.0 (Vista).

Distinto en la versión 3.3: Se agregó el parámetro `dir_fd`.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

Distinto en la versión 3.8: En Windows, ahora abre puntos de análisis que representan otra ruta (nombres sustitutos), incluidos enlaces simbólicos y uniones de directorio. El sistema operativo resuelve otros tipos de puntos de análisis como `stat()`.

os.**mkdir** (`path`, `mode=0o777`, *, `dir_fd=None`)

Cree un directorio llamado `path` con modo numérico `mode`.

Si el directorio ya existe, se excita `FileExistsError`.

En algunos sistemas, `mode` se ignora. Donde se usa, el valor actual de `umask` se enmascara primero. Si se establecen bits distintos de los últimos 9 (es decir, los últimos 3 dígitos de la representación octal del `mode`), su significado depende de la plataforma. En algunas plataformas, se ignoran y debe llamar a `chmod()` explícitamente para configurarlos.

On Windows, a `mode` of `0o700` is specifically handled to apply access control to the new directory such that only the current user and administrators have access. Other values of `mode` are ignored.

Esta función también puede admitir *rutas relativas a descriptors de directorio*.

También es posible crear directorios temporales; vea la función `tempfile` del módulo `tempfile`. `mkdtemp()`.

Lanza un *evento de auditoría* `os.mkdir` con argumentos `ruta`, `modo`, `dir_fd`.

Nuevo en la versión 3.3: El argumento `dir_fd`.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

Distinto en la versión 3.8.20: Windows now handles a `mode` of `0o700`.

os.**makedirs** (`name`, `mode=0o777`, `exist_ok=False`)

Función de creación de directorio recursiva. Como `mkdir()`, pero hace que todos los directorios de nivel intermedio sean necesarios para contener el directorio hoja.

El parámetro `mode` se pasa a `mkdir()` para crear el directorio hoja; ver *la descripción de mkdir()* por cómo se interpreta. Para configurar los bits de permiso de archivo de cualquier directorio padre recién creado,

puede configurar la `umask` antes de invocar `makedirs()`. Los bits de permiso de archivo de los directorios principales existentes no se modifican.

Si `exist_ok` es `False` (el valor predeterminado), se genera un `FileExistsError` si el directorio de destino ya existe.

Nota: `makedirs()` se confundirá si los elementos de ruta a crear incluyen `pardir` (por ejemplo, «..» en sistemas UNIX).

Esta función maneja las rutas UNC correctamente.

Lanza un *evento de auditoría* `os.mkdir` con argumentos `ruta`, `modo`, `dir_fd`.

Nuevo en la versión 3.2: El parámetro `exist_ok`.

Distinto en la versión 3.4.1: Antes de Python 3.4.1, si `exist_ok` era `True` y el directorio existía, `makedirs()` aún generaría un error si `mode` no coincidía con el modo del directorio existente. Como este comportamiento era imposible de implementar de forma segura, se eliminó en Python 3.4.1. Ver [bpo-21082](#).

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

Distinto en la versión 3.7: El argumento `mode` ya no afecta los bits de permiso de archivo de los directorios de nivel intermedio recién creados.

`os.mkfifo` (*path*, *mode=0o666*, *, *dir_fd=None*)

Cree una FIFO (una tubería con nombre) llamada *path* con modo numérico *modo*. El valor actual de `umask` se enmascara primero del modo.

Esta función también puede admitir *rutas relativas a descriptores de directorio*.

Los FIFO son tuberías a las que se puede acceder como archivos normales. Los FIFO existen hasta que se eliminan (por ejemplo con `os.unlink()`). En general, los FIFO se utilizan como punto de encuentro entre los procesos de tipo «cliente» y «servidor»: el servidor abre el FIFO para leer y el cliente lo abre para escribir. Tenga en cuenta que `mkfifo()` no abre el FIFO — solo crea el punto de encuentro.

Disponibilidad: Unix.

Nuevo en la versión 3.3: El argumento `dir_fd`.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

`os.mknod` (*path*, *mode=0o600*, *device=0*, *, *dir_fd=None*)

Cree un nodo del sistema de archivos (archivo, archivo especial del dispositivo o canalización con nombre) llamado *path*. *mode* especifica tanto los permisos para usar como el tipo de nodo que se creará, combinándose (OR bit a bit) con uno de `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK`, y `stat.S_IFIFO` (esas constantes están disponibles en `stat`). Para `stat.S_IFCHR` y `stat.S_IFBLK`, *device* define el archivo especial del dispositivo recién creado (probablemente usando `os.makedev()`), de lo contrario se ignora.

Esta función también puede admitir *rutas relativas a descriptores de directorio*.

Disponibilidad: Unix.

Nuevo en la versión 3.3: El argumento `dir_fd`.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

`os.major` (*device*)

Extrae el número principal del dispositivo de un número de dispositivo sin formato (generalmente el campo `st_dev` o `st_rdev` de `stat`).

`os.minor` (*device*)

Extrae el número menor del dispositivo de un número de dispositivo sin formato (generalmente el campo `st_dev` o `st_rdev` de `stat`).

`os.makedev` (*major*, *minor*)

Compone un número de dispositivo sin procesar a partir de los números de dispositivo mayor y menor.

os.pathconf (*path*, *name*)

Retorna información de configuración del sistema relevante para un archivo con nombre. *name* especifica el valor de configuración para recuperar; puede ser una cadena que es el nombre de un valor de sistema definido; Estos nombres se especifican en varios estándares (POSIX.1, Unix 95, Unix 98 y otros). Algunas plataformas también definen nombres adicionales. Los nombres conocidos por el sistema operativo host se dan en el diccionario `pathconf_names`. Para las variables de configuración no incluidas en esa asignación, también se acepta pasar un número entero para *name*.

Si *name* es una cadena y no se conoce, se lanza un `ValueError`. Si el sistema anfitrión no admite un valor específico para *name*, incluso si está incluido en `pathconf_names`, se genera un `OSError` con `errno.EINVAL` para el número de error.

Esta función puede soportar *especificando un descriptor de archivo*.

Disponibilidad: Unix.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

os.pathconf_names

Nombres de mapeo de diccionario aceptados por `pathconf()` y `fpathconf()` a los valores enteros definidos para esos nombres por el sistema operativo host. Esto se puede usar para determinar el conjunto de nombres conocidos por el sistema.

Disponibilidad: Unix.

os.readlink (*path*, *, *dir_fd=None*)

Retorna una cadena que representa la ruta a la que apunta el enlace simbólico. El resultado puede ser un nombre de ruta absoluto o relativo; si es relativo, se puede convertir a un nombre de ruta absoluto usando `os.path.join(os.path.dirname(path), result)`.

Si la *path* es un objeto de cadena (directa o indirectamente a través de una interfaz *PathLike*), el resultado también será un objeto de cadena y la llamada puede generar un `UnicodeDecodeError`. Si la *path* es un objeto de bytes (directa o indirectamente), el resultado será un objeto de bytes.

Esta función también puede admitir *rutas relativas a descriptors de directorio*.

Cuando intente resolver una ruta que puede contener enlaces, use `realpath()` para manejar adecuadamente la recurrencia y las diferencias de plataforma.

Disponibilidad: Unix, Windows.

Distinto en la versión 3.2: Se agregó soporte para enlaces simbólicos de Windows 6.0 (Vista).

Nuevo en la versión 3.3: El argumento *dir_fd*.

Distinto en la versión 3.6: Acepta un *path-like object* en Unix.

Distinto en la versión 3.8: Acepta un *path-like object* y un objeto de bytes en Windows.

Distinto en la versión 3.8: Se agregó soporte para uniones de directorio y se modificó para retornar la ruta de sustitución (que generalmente incluye el prefijo `\\?\\`) En lugar del campo opcional «nombre de impresión» que se retornó anteriormente.

os.remove (*path*, *, *dir_fd=None*)

Remove (delete) the file *path*. If *path* is a directory, an `IsADirectoryError` is raised. Use `rmdir()` to remove directories. If the file does not exist, a `FileNotFoundError` is raised.

Esta función puede admitir *rutas relativas a descriptors de directorio*.

En Windows, intentar eliminar un archivo que está en uso provoca una excepción; en Unix, la entrada del directorio se elimina pero el almacenamiento asignado al archivo no está disponible hasta que el archivo original ya no esté en uso.

Esta función es semánticamente idéntica a `unlink()`.

Lanza un *evento de auditoría* `os.remove` con argumentos *ruta*, *dir_fd*.

Nuevo en la versión 3.3: El argumento *dir_fd*.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

os.removedirs (*name*)

Eliminar directorios de forma recursiva. Funciona como `rmdir()` excepto que, si el directorio hoja se elimina con éxito, `removedirs()` intenta eliminar sucesivamente cada directorio principal mencionado en *path* hasta que se genere un error (que se ignora, porque generalmente significa que un directorio padre no está vacío). Por ejemplo, `os.removedirs('foo/bar/baz')` primero eliminará el directorio `'foo/bar/baz'`, y luego eliminará `'foo/bar'` y `'foo'` si están vacíos. Genera `OSError` si el directorio hoja no se pudo eliminar con éxito.

Lanza un *evento de auditoría* `os.remove` con argumentos *ruta*, *dir_fd*.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

os.rename (*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

Cambia el nombre del archivo o directorio *src* a *dst*. Si *dst* existe, la operación fallará con una subclase `OSError` en varios casos:

En Windows, si *dst* existe a `FileExistsError` siempre se genera.

En Unix, si *src* es un archivo y *dst* es un directorio o viceversa, se generará un `IsADirectoryError` o un `NotADirectoryError` respectivamente. Si ambos son directorios y *dst* está vacío, *dst* será reemplazado silenciosamente. Si *dst* es un directorio no vacío, se genera un `OSError`. Si ambos son archivos, *dst* se reemplazará en silencio si el usuario tiene permiso. La operación puede fallar en algunos sabores de Unix si *src* y *dst* están en diferentes sistemas de archivos. Si tiene éxito, el cambio de nombre será una operación atómica (este es un requisito POSIX).

Esta función puede admitir la especificación de *src_dir_fd* o *dst_dir_fd* para proporcionar *rutas relativas a los descriptores de directorio*.

Si desea sobrescribir multiplataforma del destino, use `replace()`.

Lanza un *evento de auditoría* `os.rename` con argumentos *src*, *dst*, *src_dir_fd*, *dst_dir_fd*.

Nuevo en la versión 3.3: Los argumentos *src_dir_fd* y *dst_dir_fd*.

Distinto en la versión 3.6: Acepta un *path-like object* para *src* y *dst*.

os.renames (*old*, *new*)

Directorio recursivo o función de cambio de nombre de archivo. Funciona como `rename()`, excepto que primero se intenta crear cualquier directorio intermedio necesario para que el nuevo nombre de ruta sea bueno. Después del cambio de nombre, los directorios correspondientes a los segmentos de ruta más a la derecha del nombre anterior se eliminarán usando `removeirs()`.

Nota: Esta función puede fallar con la nueva estructura de directorios realizada si carece de los permisos necesarios para eliminar el directorio o archivo hoja.

Lanza un *evento de auditoría* `os.rename` con argumentos *src*, *dst*, *src_dir_fd*, *dst_dir_fd*.

Distinto en la versión 3.6: Acepta un *path-like object* para *old* y *new*.

os.replace (*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

Cambia el nombre del archivo o directorio *src* a *dst*. Si *dst* es un directorio, se generará `OSError`. Si *dst* existe y es un archivo, será reemplazado silenciosamente si el usuario tiene permiso. La operación puede fallar si *src* y *dst* están en sistemas de archivos diferentes. Si tiene éxito, el cambio de nombre será una operación atómica (este es un requisito POSIX).

Esta función puede admitir la especificación de *src_dir_fd* o *dst_dir_fd* para proporcionar *rutas relativas a los descriptores de directorio*.

Lanza un *evento de auditoría* `os.rename` con argumentos *src*, *dst*, *src_dir_fd*, *dst_dir_fd*.

Nuevo en la versión 3.3.

Distinto en la versión 3.6: Acepta un *path-like object* para *src* y *dst*.

os.rmdir (*path*, *, *dir_fd=None*)

Elimina (*delete*) el directorio *path*. Si el directorio no existe o no está vacío, se genera un

`FileNotFoundError` o un `OSError` respectivamente. Para eliminar árboles de directorios completos, se puede usar `shutil.rmtree()`.

Esta función puede admitir *rutas relativas a descriptores de directorio* .

Lanza un *evento de auditoría* `os.rmdir` con argumentos `ruta`, `dir_fd`.

Nuevo en la versión 3.3: El parámetro `dir_fd`.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta* .

`os.scandir (path='.')`

Return an iterator of `os.DirEntry` objects corresponding to the entries in the directory given by `path`. The entries are yielded in arbitrary order, and the special entries `'.'` and `'..'` are not included. If a file is removed from or added to the directory after creating the iterator, whether an entry for that file be included is unspecified.

El uso de `scandir()` en lugar de `listdir()` puede aumentar significativamente el rendimiento del código que también necesita información de tipo de archivo o atributo de archivo, porque: los objetos `os.DirEntry` exponen esta información si el sistema operativo proporciona cuando escanea un directorio. Todos los métodos `os.DirEntry` pueden realizar una llamada al sistema, pero `is_dir()` y `is_file()` generalmente solo requieren una llamada al sistema para enlaces simbólicos; `os.DirEntry.stat()` siempre requiere una llamada al sistema en Unix, pero solo requiere una para enlaces simbólicos en Windows.

`path` puede ser un *path-like object* . Si `path` es de tipo `bytes` (directa o indirectamente a través de la interfaz *PathLike*), el tipo de `name` y los atributos `path` de cada `os.DirEntry` serán `bytes`; en todas las demás circunstancias, serán del tipo `str`.

Esta función también puede admitir *especificando un descriptor de archivo* ; el descriptor de archivo debe hacer referencia a un directorio.

Lanza un *evento de auditoría* `os.scandir` con argumento `ruta`.

El iterador `scandir()` admite el protocolo *context manager* y tiene el siguiente método:

`scandir.close()`

Cierre el iterador y libere los recursos adquiridos.

Esto se llama automáticamente cuando el iterador se agota o se recolecta basura, o cuando ocurre un error durante la iteración. Sin embargo, es aconsejable llamarlo explícitamente o utilizar la palabra clave `with`.

Nuevo en la versión 3.6.

El siguiente ejemplo muestra un uso simple de `scandir()` para mostrar todos los archivos (excepto los directorios) en la `path` dada que no comienzan con `'.'`. La llamada `entry.is_file()` generalmente no realizará una llamada adicional al sistema:

```
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and entry.is_file():
            print(entry.name)
```

Nota: En sistemas basados en Unix, `scandir()` usa el `opendir()` del sistema y `readdir()` funciones. En Windows, utiliza el Win32 `FindFirstFileW` y `FindNextFileW` funciones.

Nuevo en la versión 3.5.

Nuevo en la versión 3.6: Se agregó soporte para el protocolo *context manager* y el método `close()`. Si un iterador `scandir()` no está agotado ni cerrado explícitamente, se emitirá a `ResourceWarning` en su destructor.

La función acepta un *path-like object* .

Distinto en la versión 3.7: Soporte agregado para *descriptores de archivo* en Unix.

class `os.DirEntry`

Objeto generado por `scandir()` para exponer la ruta del archivo y otros atributos de archivo de una entrada de directorio.

`scandir()` proporcionará tanta información como sea posible sin hacer llamadas adicionales al sistema. Cuando se realiza una llamada al sistema `stat()` o `lstat()`, el objeto `os.DirEntry` almacenará en caché el resultado.

Las instancias `os.DirEntry` no están destinadas a ser almacenadas en estructuras de datos de larga duración; si sabe que los metadatos del archivo han cambiado o si ha transcurrido mucho tiempo desde la llamada `scandir()`, llame a `os.stat(entry.path)` para obtener información actualizada.

Debido a que los métodos `os.DirEntry` pueden hacer llamadas al sistema operativo, también pueden generar `OSError`. Si necesita un control muy preciso sobre los errores, puede detectar `OSError` cuando llame a uno de los métodos `os.DirEntry` y maneje según corresponda.

Para ser directamente utilizable como *path-like object*, `os.DirEntry` implementa la interfaz `PathLike`.

Los atributos y métodos en una instancia de `os.DirEntry` son los siguientes:

name

El nombre de archivo base de la entrada, relativo al argumento `scandir()` *path*.

El atributo `name` será `bytes` si el argumento `scandir()` *path* es de tipo `bytes` y `str` de lo contrario. Utilice `fsdecode()` para decodificar los nombres de archivo de bytes.

path

El nombre completo de la ruta de entrada: equivalente a `os.path.join(scandir_path, entry.name)` donde `scandir_path` es el argumento `scandir()` *path*. La ruta solo es absoluta si el argumento `scandir()` *path* fue absoluto. Si el argumento `scandir()` *path* era un *descriptor de archivo*, el atributo `path` es el mismo que el atributo `name`.

El atributo `path` será `bytes` si el argumento `scandir()` *path* es de tipo `bytes` y `str` de lo contrario. Utilice `fsdecode()` para decodificar los nombres de archivo de bytes.

inode()

Retorna el número de inodo de la entrada.

El resultado se almacena en caché en el objeto `os.DirEntry`. Use `os.stat(entry.path, follow_symlinks=False).st_ino` para obtener información actualizada.

En la primera llamada no almacenada en caché, se requiere una llamada del sistema en Windows pero no en Unix.

is_dir(*, follow_symlinks=True)

Retorna `True` si esta entrada es un directorio o un enlace simbólico que apunta a un directorio; retorna `False` si la entrada es o apunta a cualquier otro tipo de archivo, o si ya no existe.

Si `follow_symlinks` es `False`, retorna `True` solo si esta entrada es un directorio (sin seguir los enlaces simbólicos); retorna `False` si la entrada es cualquier otro tipo de archivo o si ya no existe.

El resultado se almacena en caché en el objeto `os.DirEntry`, con un caché separado para `follow_symlinks` `True` y `False`. Llame a `os.stat()` junto con `stat.S_ISDIR()` para obtener información actualizada.

En la primera llamada no almacenada en caché, no se requiere ninguna llamada al sistema en la mayoría de los casos. Específicamente, para los enlaces no simbólicos, ni Windows ni Unix requieren una llamada al sistema, excepto en ciertos sistemas de archivos Unix, como los sistemas de archivos de red, que retornan `dirent.d_type == DT_UNKNOWN`. Si la entrada es un enlace simbólico, se requerirá una llamada al sistema para seguir el enlace simbólico a menos que `follow_symlinks` sea `False`.

Este método puede generar `OSError`, como `PermissionError`, pero `FileNotFoundError` se captura y no se genera.

is_file(*, follow_symlinks=True)

Retorna `True` si esta entrada es un archivo o un enlace simbólico que apunta a un archivo; retorna `False` si la entrada es o apunta a un directorio u otra entrada que no sea de archivo, o si ya no existe.

Si `follow_symlinks` es `False`, retorna `True` solo si esta entrada es un archivo (sin los siguientes enlaces simbólicos); retorna `False` si la entrada es un directorio u otra entrada que no sea de archivo, o si ya no existe.

El resultado se almacena en caché en el objeto `os.DirEntry`. El almacenamiento en caché, las llamadas realizadas al sistema y las excepciones generadas son las siguientes `is_dir()`.

`is_symlink()`

Retorna `True` si esta entrada es un enlace simbólico (incluso si está roto); retorna `False` si la entrada apunta a un directorio o cualquier tipo de archivo, o si ya no existe.

El resultado se almacena en caché en el objeto `os.DirEntry`. Llame a `os.path.islink()` para obtener información actualizada.

En la primera llamada no almacenada en caché, no se requiere ninguna llamada al sistema en la mayoría de los casos. Específicamente, ni Windows ni Unix requieren una llamada al sistema, excepto en ciertos sistemas de archivos Unix, como los sistemas de archivos de red, que retornan `dirent.d_type == DT_UNKNOWN`.

Este método puede generar `OSError`, como `PermissionError`, pero `FileNotFoundError` se captura y no se genera.

`stat(*, follow_symlinks=True)`

Retorna un objeto a `stat_result` para esta entrada. Este método sigue enlaces simbólicos por defecto; para crear un enlace simbólico agregue el argumento `follow_symlinks=False`.

En Unix, este método siempre requiere una llamada al sistema. En Windows, solo requiere una llamada al sistema si `follow_symlinks` es `True` y la entrada es un punto de análisis (por ejemplo, un enlace simbólico o una unión de directorio).

En Windows, los atributos `st_ino`, `st_dev` y `st_nlink` de `stat_result` siempre se establecen en cero. Llame a `os.stat()` para obtener estos atributos.

El resultado se almacena en caché en el objeto `os.DirEntry`, con un caché separado para `follow_symlinks` `True` y `False`. Llame a `os.stat()` para obtener información actualizada.

Tenga en cuenta que existe una buena correspondencia entre varios atributos y métodos de `os.DirEntry` y de `pathlib.Path`. En particular, el atributo `name` tiene el mismo significado, al igual que los métodos `is_dir()`, `is_file()`, `is_symlink()` y `stat()`.

Nuevo en la versión 3.5.

Distinto en la versión 3.6: Se agregó soporte para la interfaz `PathLike`. Se agregó soporte para rutas de `bytes` en Windows.

`os.stat(path, *, dir_fd=None, follow_symlinks=True)`

Obtener el estado de un archivo o un descriptor de archivo. Realice el equivalente de a llamada del sistema `stat()` en la ruta dada. `path` puede especificarse como una cadena o bytes, directa o indirectamente a través de la interfaz `PathLike`, o como un descriptor de archivo abierto. Retorna un objeto `stat_result`.

Esta función normalmente sigue enlaces simbólicos; para crear un enlace simbólico agregue el argumento `follow_symlinks=False`, o use `lstat()`.

Esta función puede soportar *especificando un descriptor de archivo y no siguen enlaces simbólicos*.

En Windows, pasar `follow_symlinks=False` deshabilitará el seguimiento de todos los puntos de análisis sustitutos de nombre, que incluyen enlaces simbólicos y uniones de directorio. Se abrirán directamente otros tipos de puntos de análisis que no se parecen a los enlaces o que el sistema operativo no puede seguir. Al seguir una cadena de enlaces múltiples, esto puede dar como resultado que se retorna el enlace original en lugar del no enlace que impidió el recorrido completo. Para obtener resultados estadísticos para la ruta final en este caso, use la función `os.path.realpath()` para resolver el nombre de la ruta lo más posible y llame a `lstat()` en el resultado. Esto no se aplica a enlaces simbólicos o puntos de unión colgantes, lo que generará las excepciones habituales.

El diseño de todos los módulos incorporados de Python dependientes del sistema operativo es tal que, mientras funcionalidad esté disponible, usará la misma interfaz; por ejemplo, la función `os.stat(path)` retorna estadísticas sobre la ruta (`path`) en el mismo formato (lo que sucede originalmente con la interfaz POSIX).


```

>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264

```

Ver también:

`fstat()` y funciones `lstat()`.

Nuevo en la versión 3.3: Se agregaron los argumentos `dir_fd` y `follow_symlinks`, especificando un descriptor de archivo en lugar de una ruta.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

Distinto en la versión 3.8: En Windows, ahora se siguen todos los puntos de análisis que el sistema operativo puede resolver, y pasar `follow_symlinks=False` desactiva los siguientes puntos de análisis sustitutos de nombre. Si el sistema operativo alcanza un punto de análisis que no puede seguir, `stat` ahora retorna la información de la ruta original como si se hubiera especificado `follow_symlinks=False` en lugar de generar un error.

class os.stat_result

Objeto cuyos atributos corresponden aproximadamente a los miembros de la estructura `stat`. Se utiliza para el resultado de `os.stat()`, `os.fstat()` y `os.lstat()`.

Atributos:

st_mode

Modo de archivo: tipo de archivo y bits de modo de archivo (permisos).

st_ino

Dependiendo de la plataforma, pero si no es cero, identifica de forma exclusiva el archivo para un valor dado de `st_dev`. Típicamente:

- el número de inodo en Unix,
- el índice del archivo <<https://msdn.microsoft.com/en-us/library/aa363788>> en Windows

st_dev

Identificador del dispositivo en el que reside este archivo.

st_nlink

Número de enlaces duros.

st_uid

Identificador de usuario del propietario del archivo.

st_gid

Identificador de grupo del propietario del archivo.

st_size

Tamaño del archivo en bytes, si es un archivo normal o un enlace simbólico. El tamaño de un enlace simbólico es la longitud del nombre de ruta que contiene, sin un byte nulo de terminación.

Marcas de tiempo:

st_atime

Tiempo de acceso más reciente expresado en segundos.

st_mtime

Tiempo de modificación de contenido más reciente expresado en segundos.

st_ctime

Depende de la plataforma:

- el momento del cambio de metadatos más reciente en Unix,
- El tiempo de creación en Windows, expresado en segundos.

st_atime_ns

Tiempo de acceso más reciente expresado en nanosegundos como un entero.

st_mtime_ns

Hora de la modificación de contenido más reciente expresada en nanosegundos como un entero.

st_ctime_ns

Depende de la plataforma:

- el momento del cambio de metadatos más reciente en Unix,
- El tiempo de creación en Windows, expresado en nanosegundos como un entero.

Nota: El significado exacto y la resolución de los atributos `st_atime`, `st_mtime` y `st_ctime` dependen del sistema operativo y del sistema de archivos. Por ejemplo, en sistemas Windows que utilizan los sistemas de archivos FAT o FAT32, `st_mtime` tiene una resolución de 2 segundos y `st_atime` tiene una resolución de solo 1 día. Consulte la documentación de su sistema operativo para más detalles.

De manera similar, aunque `st_atime_ns`, `st_mtime_ns` y `st_ctime_ns` siempre se expresan en nanosegundos, muchos sistemas no proporcionan precisión en nanosegundos. En los sistemas que proporcionan precisión en nanosegundos, el objeto de punto flotante utilizado para almacenar `st_atime`, `st_mtime`, y `st_ctime` no puede preservarlo todo, y como tal será ligeramente inexacto. Si necesita las marcas de tiempo exactas, siempre debe usar `st_atime_ns`, `st_mtime_ns` y `st_ctime_ns`.

En algunos sistemas Unix (como Linux), los siguientes atributos también pueden estar disponibles:

st_blocks

Número de bloques de 512 bytes asignados para el archivo. Esto puede ser más pequeño que `st_size` / 512 cuando el archivo tiene agujeros.

st_blksize

Tamaño de bloque «preferido» para una eficiente E/S del sistema de archivos. Escribir en un archivo en fragmentos más pequeños puede causar una lectura-modificación-reescritura ineficiente.

st_rdev

Tipo de dispositivo si es un dispositivo inode.

st_flags

Indicadores definidos por el usuario para el archivo.

En otros sistemas Unix (como FreeBSD), los siguientes atributos pueden estar disponibles (pero solo se pueden completar si la raíz intenta usarlos):

st_gen

Número de generación de archivos.

st_birthtime

Hora de creación del archivo.

En Solaris y derivados, los siguientes atributos también pueden estar disponibles:

st_fstype

Cadena que identifica de forma exclusiva el tipo de sistema de archivos que contiene el archivo.

En los sistemas Mac OS, los siguientes atributos también pueden estar disponibles:

st_rsize

Tamaño real del archivo.

st_creator

Creador del archivo.

st_type

Tipo de archivo.

En los sistemas Windows, los siguientes atributos también están disponibles:

st_file_attributes

Atributos del archivo de Windows: miembro `dwFileAttributes` de la estructura `BY_HANDLE_FILE_INFORMATION` retornado por `GetFileInformationByHandle()`.
Vea las constantes `FILE_ATTRIBUTE_*` en el módulo `stat`.

st_reparse_tag

Cuando `st_file_attributes` tiene el conjunto `FILE_ATTRIBUTE_REPARSE_POINT`, este campo contiene la etiqueta que identifica el tipo de punto de análisis. Vea las constantes `IO_REPARSE_TAG_*` en el módulo `stat`.

El módulo estándar `stat` define funciones y constantes que son útiles para extraer información de la estructura `stat`. (En Windows, algunos elementos están llenos de valores ficticios).

Para compatibilidad con versiones anteriores, una instancia de `stat_result` también es accesible como una tupla de al menos 10 enteros que dan los miembros más importantes (y portátiles) de la estructura `stat`, en el orden `st_mode`, `st_ino`, `st_dev`, `st_nlink`, `st_uid`, `st_gid`, `st_size`, `st_atime`, `st_mtime`, `st_ctime`. Algunas implementaciones pueden agregar más elementos al final. Para compatibilidad con versiones anteriores de Python, acceder a `stat_result` como una tupla siempre retorna enteros.

Nuevo en la versión 3.3: Se agregaron los miembros `st_atime_ns`, `st_mtime_ns` y `st_ctime_ns`.

Nuevo en la versión 3.5: Se agregó el miembro `st_file_attributes` en Windows.

Distinto en la versión 3.5: Windows ahora retorna el índice del archivo como `st_ino` cuando está disponible.

Nuevo en la versión 3.7: Se agregó el miembro `st_fstype` a Solaris/derivados.

Nuevo en la versión 3.8: Se agregó el miembro `st_reparse_tag` en Windows.

Distinto en la versión 3.8: En Windows, el miembro `st_mode` ahora identifica archivos especiales como `S_IFCHR`, `S_IFIFO` o `S_IFBLK` según corresponda.

os.statvfs(path)

Realice una llamada al sistema a `statvfs()` en la ruta dada. El valor de retorno es un objeto cuyos atributos describen el sistema de archivos en la ruta dada y corresponden a los miembros de la estructura `statvfs`, a saber: `f_bsize`, `f_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`, `f_fsid`.

Se definen dos constantes de nivel de módulo para: indicadores de bit del atributo `f_flag`: si `ST_RDONLY` está configurado, el sistema de archivos está montado de solo lectura, y si `ST_NOSUID` está configurado, el la semántica de los bits `setuid/setgid` está deshabilitada o no es compatible.

Se definen constantes de nivel de módulo adicionales para sistemas basados en GNU / glibc. Estos son `ST_NODEV` (no permitir el acceso a archivos especiales del dispositivo), `ST_NOEXEC` (no permitir la ejecución del programa), `ST_SYNCHRONOUS` (las escrituras se sincronizan a la vez), `ST_MANDLOCK` (permitir bloqueos obligatorios en un FS), `ST_WRITE` (escribir en el archivo/directorio/enlace simbólico), `ST_APPEND` (archivo de solo agregado), `ST_IMMUTABLE` (archivo inmutable), `ST_NOATIME` (no actualiza los tiempos de acceso), `ST_NODIRATIME` (no actualiza los tiempos de acceso al directorio), `ST_RELATIME` (tiempo de actualización relativo a `mtime/ctime`).

Esta función puede soportar *especificando un descriptor de archivo*.

Disponibilidad: Unix.

Distinto en la versión 3.2: Se agregaron las constantes `ST_RDONLY` y `ST_NOSUID`.

Nuevo en la versión 3.3: Se agregó soporte para especificar *path* como un descriptor de archivo abierto.

Distinto en la versión 3.4: El `ST_NODEV`, `ST_NOEXEC`, `ST_SYNCHRONOUS`, `ST_MANDLOCK`, `ST_WRITE`, `ST_APPEND`, `ST_IMMUTABLE`, `ST_NOATIME`, `ST_NODIRATIME`, y `ST_RELATIME` se agregaron constantes.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

Nuevo en la versión 3.7: Agregado `f_fsid`.

`os.supports_dir_fd`

A objeto *set* que indica qué funciones en el módulo `os` aceptan un descriptor de archivo abierto para su parámetro `dir_fd`. Las diferentes plataformas proporcionan características diferentes, y la funcionalidad subyacente que Python usa para implementar el parámetro `dir_fd` no está disponible en todas las plataformas que admite Python. En aras de la coherencia, las funciones que pueden admitir `dir_fd` siempre permiten especificar el parámetro, pero generarán una excepción si la funcionalidad se utiliza cuando no está disponible localmente. (Especificar `None` para `dir_fd` siempre es compatible con todas las plataformas).

Para verificar si una función particular acepta un descriptor de archivo abierto para su parámetro `dir_fd`, use el operador `in` en `supports_dir_fd`. Como ejemplo, esta expresión se evalúa como `True` si `os.stat()` acepta descriptores de archivos abiertos para `dir_fd` en la plataforma local:

```
os.stat in os.supports_dir_fd
```

Actualmente, los parámetros `dir_fd` solo funcionan en plataformas Unix; ninguno de ellos funciona en Windows.

Nuevo en la versión 3.3.

`os.supports_effective_ids`

Un objeto *set* que indica si `os.access()` permite especificar `True` para su parámetro `effective_ids` en la plataforma local. (Especificar `False` para `effective_id` siempre es compatible con todas las plataformas). Si la plataforma local lo admite, la colección contendrá `os.access()`; de lo contrario estará vacío.

Esta expresión se evalúa como `True` si `os.access()` admite `effective_id=True` en la plataforma local:

```
os.access in os.supports_effective_ids
```

Actualmente, `effective_ids` solo es compatible con plataformas Unix; No funciona en Windows.

Nuevo en la versión 3.3.

`os.supports_fd`

A objeto *set* que indica qué funciones en el módulo `os` permiten especificar su parámetro `path` como un descriptor de archivo abierto en la plataforma local. Las diferentes plataformas proporcionan características diferentes, y la funcionalidad subyacente que Python utiliza para aceptar descriptores de archivos abiertos como argumentos `path` no está disponible en todas las plataformas que admite Python.

Para determinar si una función en particular permite especificar un descriptor de archivo abierto para su parámetro `path`, use el operador `in` en `supports_fd`. Como ejemplo, esta expresión se evalúa como `True` si `os.chdir()` acepta descriptores de archivo abiertos para `path` en su plataforma local:

```
os.chdir in os.supports_fd
```

Nuevo en la versión 3.3.

`os.supports_follow_symlinks`

Un objeto *set* que indica qué funciones en el módulo `os` aceptan `False` para su parámetro `follow_symlinks` en la plataforma local. Las diferentes plataformas proporcionan características diferentes, y la funcionalidad subyacente que Python usa para implementar `follow_symlinks` no está disponible en todas las plataformas que admite Python. En aras de la coherencia, las funciones que pueden admitir `follow_symlinks` siempre permiten especificar el parámetro, pero arrojarán una excepción si la funcionalidad se utiliza cuando no está disponible localmente. (Especificar `True` para `follow_symlinks` siempre se admite en todas las plataformas).

Para verificar si una función particular acepta `False` para su parámetro `follow_symlinks`, use el operador `in` en `supports_follow_symlinks`. Como ejemplo, esta expresión se evalúa como `True` si puede especificar `follow_symlinks=False` al llamar a `os.stat()` en la plataforma local:

```
os.stat in os.supports_follow_symlinks
```

Nuevo en la versión 3.3.

`os.symlink (src, dst, target_is_directory=False, *, dir_fd=None)`

Cree un enlace simbólico que apunte a *src* llamado *dst*.

En Windows, un enlace simbólico representa un archivo o un directorio, y no se transforma dinámicamente en el destino. Si el objetivo está presente, el tipo de enlace simbólico se creará para que coincida. De lo contrario, el enlace simbólico se creará como un directorio si *target_is_directory* es `True` o un enlace simbólico de archivo (el valor predeterminado) de lo contrario. En plataformas que no son de Windows, *target_is_directory* se ignora.

Esta función puede admitir *rutas relativas a descriptores de directorio*.

Nota: En las versiones más recientes de Windows 10, las cuentas sin privilegios pueden crear enlaces simbólicos si el Modo desarrollador está habilitado. Cuando el Modo desarrollador no está disponible / habilitado, se requiere el privilegio *SeCreateSymbolicLinkPrivilege*, o el proceso debe ejecutarse como administrador.

OSError se lanza cuando un usuario sin privilegios llama a la función.

Lanza un *evento de auditoría* `os.symlink` con argumentos *src*, *dst*, *dir_fd*.

Disponibilidad: Unix, Windows.

Distinto en la versión 3.2: Se agregó soporte para enlaces simbólicos de Windows 6.0 (Vista).

Nuevo en la versión 3.3: Se agregó el argumento *dir_fd* y ahora permite *target_is_directory* en plataformas que no son de Windows.

Distinto en la versión 3.6: Acepta un *path-like object* para *src* y *dst*.

Distinto en la versión 3.8: Se agregó soporte para enlaces simbólicos sin elevar en Windows con el modo de desarrollador.

`os.sync ()`

Forzar la escritura de todo en el disco.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

`os.truncate (path, length)`

Trunca el archivo correspondiente a *path*, para que tenga como máximo *length* bytes de tamaño.

Esta función puede soportar *especificando un descriptor de archivo*.

Lanza un *evento de auditoría* `os.truncate` con argumentos *path*, *length*.

Disponibilidad: Unix, Windows.

Nuevo en la versión 3.3.

Distinto en la versión 3.5: Se agregó soporte para Windows

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

`os.unlink (path, *, dir_fd=None)`

Elimina (elimine) el archivo *path*. Esta función es semánticamente idéntica a `remove ()`; El nombre `unlink` es su nombre tradicional de Unix. Consulte la documentación de `remove ()` para obtener más información.

Lanza un *evento de auditoría* `os.remove` con argumentos *ruta*, *dir_fd*.

Nuevo en la versión 3.3: El parámetro *dir_fd*.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

`os.utime (path, times=None, *[, ns], dir_fd=None, follow_symlinks=True)`

Establece el acceso y los tiempos modificados del archivo especificado por *path*.

`utime ()` toma dos parámetros opcionales, *times* y *ns*. Estos especifican los tiempos establecidos en *path* y se utilizan de la siguiente manera:

- Si se especifica *ns*, debe ser una tupla de 2 de la forma (*atime_ns*, *mtime_ns*) donde cada miembro es un int que expresa nanosegundos.
- Si *times* no es None, debe ser una 2-tupla de la forma (*atime*, *mtime*) donde cada miembro es un int o flotante que expresa segundos.
- Si *times* es None y *ns* no está especificado, esto es equivalente a especificar *ns*=(*atime_ns*, *mtime_ns*) donde ambas horas son la hora actual.

Es un error especificar tuplas para *times* y *ns*.

Tenga en cuenta que las horas exactas que establezca aquí pueden no ser retornadas por una llamada posterior *stat()*, dependiendo de la resolución con la que su sistema operativo registre los tiempos de acceso y modificación; ver *stat()*. La mejor manera de preservar los tiempos exactos es usar los campos *st_atime_ns* y *st_mtime_ns* del objeto de resultado *os.stat()* con el parámetro *ns* para 'utime'.

Esta función puede soportar *especificando un descriptor de archivo, rutas relativas a los descriptores de directorio y no seguir enlaces simbólicos*.

Lanza un *evento de auditoría* *os.utime* con argumentos *path*, *times*, *ns*, *dir_fd*.

Nuevo en la versión 3.3: Se agregó soporte para especificar *path* como un descriptor de archivo abierto, y los parámetros *dir_fd*, *follow_symlinks* y *ns*.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

os.walk (*top*, *topdown*=True, *onerror*=None, *followlinks*=False)

Genere los nombres de archivo en un árbol de directorios recorriendo el árbol de arriba hacia abajo o de abajo hacia arriba. Para cada directorio en el árbol enraizado en el directorio *top* (incluido *top*), produce una tupla de 3 tuplas (*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)*. Whether or not the lists are sorted depends on the file system. If a file is removed from or added to the *dirpath* directory during generating the lists, whether a name for that file be included is unspecified.

Si el argumento opcional *topdown* es True o no se especifica, el triple para un directorio se genera antes de triplicarse para cualquiera de sus subdirectorios (los directorios se generan de arriba hacia abajo). Si *topdown* es False, el triple para un directorio se genera después de los triples para todos sus subdirectorios (los directorios se generan de abajo hacia arriba). No importa el valor de *topdown*, la lista de subdirectorios se recupera antes de que se generen las tuplas para el directorio y sus subdirectorios.

Cuando *topdown* es True, la persona que llama puede modificar la lista *dirnames* en su lugar (quizás usando *del* o asignación de corte) y *walk()* solo se repetirá en los subdirectorios cuyos nombres permanecen en *dirnames*; Esto se puede utilizar para podar la búsqueda, imponer un orden específico de visitas o incluso para informar *walk()* sobre los directorios que la persona que llama crea o renombra antes de que se reanude *walk()* nuevamente. La modificación de *dirnames* cuando *topdown* es False no tiene ningún efecto en el comportamiento de la caminata, porque en el modo ascendente los directorios en *dirnames* se generan antes de que se genere *dirpath*.

Por defecto, los errores de la llamada *scandir()* se ignoran. Si se especifica el argumento opcional *onerror*, debería ser una función; se llamará con un argumento, una instancia *OSError*. Puede informar el error para continuar con la caminata, o generar la excepción para abortar la caminata. Tenga en cuenta que el nombre de archivo está disponible como el atributo *filename* del objeto de excepción.

Por defecto, *walk()* no entrará en enlaces simbólicos que se resuelven en directorios. Establece *followlinks* en True para visitar los directorios señalados por los enlaces simbólicos, en los sistemas que los admiten.

Nota: Tenga en cuenta que establecer *followlinks* en True puede conducir a una recursión infinita si un enlace apunta a un directorio padre de sí mismo. *walk()* no realiza un seguimiento de los directorios que ya visitó.

Nota: Si pasa un nombre de ruta relativo, no cambie el directorio de trabajo actual entre las reanudaciones de `walk()`. `walk()` nunca cambia el directorio actual, y supone que la persona que llama tampoco.

Este ejemplo muestra el número de bytes que toman los archivos que no son de directorio en cada directorio bajo el directorio inicial, excepto que no se ve en ningún subdirectorio CVS:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS') # don't visit CVS directories
```

En el siguiente ejemplo (implementación simple de `shutil.rmtree()`), recorrer el árbol de abajo hacia arriba es esencial, `rmdir()` no permite eliminar un directorio antes de que el directorio esté vacío:

```
# 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))
```

Distinto en la versión 3.5: Esta función ahora llama `os.scandir()` en lugar de `os.listdir()`, lo que lo hace más rápido al reducir el número de llamadas a `os.stat()`.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

`os.fwalk(top='.', topdown=True, onerror=None, *, follow_symlinks=False, dir_fd=None)`

Esto se comporta exactamente como `walk()`, excepto que produce 4 tuplas (`dirpath`, `dirnames`, `filenames`, `dirfd`), y admite `dir_fd`.

`dirpath`, `dirnames` y `filenames` son idénticos a `walk()` output, y `dirfd` es un descriptor de archivo que se refiere al directorio `dirpath`.

Esta función siempre admite *rutas relativas a descriptors de directorio* y *no siguen enlaces simbólicos*. Sin embargo, tenga en cuenta que, a diferencia de otras funciones, el valor predeterminado `fwalk()` para `follow_symlinks` es `False`.

Nota: Dado que `fwalk()` produce descriptors de archivo, estos solo son válidos hasta el siguiente paso de iteración, por lo que debe duplicarlos (por ejemplo, con `dup()`) si desea mantenerlos más tiempo.

Este ejemplo muestra el número de bytes que toman los archivos que no son de directorio en cada directorio bajo el directorio inicial, excepto que no se ve en ningún subdirectorio CVS:

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/email'):
    print(root, "consumes", end=" ")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]),
          end=" ")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS') # don't visit CVS directories
```


En el siguiente ejemplo, recorrer el árbol de abajo hacia arriba es esencial: `rmdir()` no permite eliminar un directorio antes de que el directorio esté vacío:

```
# 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, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

Disponibilidad: Unix.

Nuevo en la versión 3.3.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

Distinto en la versión 3.7: Se agregó soporte para rutas de acceso *bytes*.

`os.memfd_create(name[, flags=os.MFD_CLOEXEC])`

Cree un archivo anónimo y retorna un descriptor de archivo que se refiera a él. *flags* debe ser una de las constantes `os.MFD_*` disponibles en el sistema (o una combinación ORed bit a bit de ellas). Por defecto, el nuevo descriptor de archivo es *no heredable*.

El nombre proporcionado en *name* se utiliza como nombre de archivo y se mostrará como el destino del enlace simbólico correspondiente en el directorio `/proc/self/fd/`. El nombre que se muestra siempre tiene el prefijo `memfd:` y solo sirve para fines de depuración. Los nombres no afectan el comportamiento del descriptor de archivo y, como tal, varios archivos pueden tener el mismo nombre sin efectos secundarios.

Disponibilidad: Linux 3.17 o posterior con glibc 2.27 o posterior.

Nuevo en la versión 3.8.

`os.MFD_CLOEXEC`
`os.MFD_ALLOW_SEALING`
`os.MFD_HUGETLB`
`os.MFD_HUGE_SHIFT`
`os.MFD_HUGE_MASK`
`os.MFD_HUGE_64KB`
`os.MFD_HUGE_512KB`
`os.MFD_HUGE_1MB`
`os.MFD_HUGE_2MB`
`os.MFD_HUGE_8MB`
`os.MFD_HUGE_16MB`
`os.MFD_HUGE_32MB`
`os.MFD_HUGE_256MB`
`os.MFD_HUGE_512MB`
`os.MFD_HUGE_1GB`
`os.MFD_HUGE_2GB`
`os.MFD_HUGE_16GB`

Estas flags se pueden pasar a `memfd_create()`.

Disponibilidad: Linux 3.17 o posterior con glibc 2.27 o posterior. Los indicadores `MFD_HUGE*` solo están disponibles desde Linux 4.14.

Nuevo en la versión 3.8.

Atributos extendidos de Linux

Nuevo en la versión 3.3.

Estas funciones están disponibles solo en Linux.

os.**getxattr** (*path*, *attribute*, *, *follow_symlinks=True*)

Retorna el valor del atributo del sistema de archivos extendido *attribute* para *path*. *attribute* puede ser bytes o str (directa o indirectamente a través de la interfaz *PathLike*). Si es str, se codifica con la codificación del sistema de archivos.

Esta función puede soportar *especificando un descriptor de archivo y no siguen enlaces simbólicos*.

Lanza un *evento de auditoría* os.getxattr con argumentos *path*, *atributo*.

Distinto en la versión 3.6: Acepta un *path-like object* para *path**y **attribute*.

os.**listxattr** (*path=None*, *, *follow_symlinks=True*)

Retorna una lista de los atributos del sistema de archivos extendido en *path*. Los atributos en la lista se representan como cadenas decodificadas con la codificación del sistema de archivos. Si *path* es None, *listxattr()* examinará el directorio actual.

Esta función puede soportar *especificando un descriptor de archivo y no siguen enlaces simbólicos*.

Lanza un *evento de auditoría* os.listxattr con el argumento *path*.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

os.**removexattr** (*path*, *attribute*, *, *follow_symlinks=True*)

Elimina el atributo del sistema de archivos extendido *attribute* de *path*. *attribute* debe ser bytes o str (directa o indirectamente a través de la interfaz *PathLike*). Si es una cadena, se codifica con la codificación del sistema de archivos.

Esta función puede soportar *especificando un descriptor de archivo y no siguen enlaces simbólicos*.

Lanza un *evento de auditoría* os.removexattr con argumentos *path*, *attribute*.

Distinto en la versión 3.6: Acepta un *path-like object* para *path**y **attribute*.

os.**setxattr** (*path*, *attribute*, *value*, *flags=0*, *, *follow_symlinks=True*)

Set the extended filesystem attribute *attribute* on *path* to *value*. *attribute* must be a bytes or str with no embedded NULs (directly or indirectly through the *PathLike* interface). If it is a str, it is encoded with the filesystem encoding. *flags* may be *XATTR_REPLACE* or *XATTR_CREATE*. If *XATTR_REPLACE* is given and the attribute does not exist, ENODATA will be raised. If *XATTR_CREATE* is given and the attribute already exists, the attribute will not be created and EEXISTS will be raised.

Esta función puede soportar *especificando un descriptor de archivo y no siguen enlaces simbólicos*.

Nota: Un error en las versiones de kernel de Linux anteriores a 2.6.39 hizo que el argumento de las flags se ignorara en algunos sistemas de archivos.

Lanza un *evento de auditoría* os.setxattr con argumentos *path*, *attribute*, *value*, *flags*.

Distinto en la versión 3.6: Acepta un *path-like object* para *path**y **attribute*.

os.**XATTR_SIZE_MAX**

El tamaño máximo que puede tener el valor de un atributo extendido. Actualmente, esto es 64 KiB en Linux.

os.**XATTR_CREATE**

Este es un valor posible para el argumento flags en *setxattr()*. Indica que la operación debe crear un atributo.

os.**XATTR_REPLACE**

Este es un valor posible para el argumento flags en *setxattr()*. Indica que la operación debe reemplazar un atributo existente.

16.1.6 Gestión de proceso

Estas funciones pueden usarse para crear y administrar procesos.

Las varias funciones `exec *` toman una lista de argumentos para el nuevo programa cargado en el proceso. En cada caso, el primero de estos argumentos se pasa al nuevo programa como su propio nombre en lugar de como un argumento que un usuario puede haber escrito en una línea de comando. Para el programador C, este es el `argv[0]` pasado a un programa `main()`. Por ejemplo, `os.execv('/bin/echo', ['foo', 'bar'])` solo imprimirá `bar` en la salida estándar; `foo` parecerá ignorado.

`os.abort()`

Genere una señal `SIGABRT` para el proceso actual. En Unix, el comportamiento predeterminado es producir un volcado de núcleo; en Windows, el proceso retorna inmediatamente un código de salida de 3. Tenga en cuenta que llamar a esta función no llamará al controlador de señal Python registrado para `SIGABRT` con `signal.signal()`.

`os.add_dll_directory(path)`

Agregue una ruta a la ruta de búsqueda de DLL.

Esta ruta de búsqueda se utiliza al resolver dependencias para módulos de extensión importados (el módulo en sí se resuelve a través de `sys.path`), y también mediante `ctypes`.

Elimina el directorio llamando a `close()` en el objeto retornado o utilizándolo en una `with` instrucción.

Consulte la [documentación de Microsoft](#) para obtener más información sobre cómo se cargan las DLL.

Lanza un [evento de auditoría](#) `os.add_dll_directory` con el argumento `path`.

Disponibilidad: Windows.

Nuevo en la versión 3.8: Las versiones anteriores de CPython resolverían las DLL utilizando el comportamiento predeterminado para el proceso actual. Esto condujo a inconsistencias, como solo a veces buscar `PATH` o el directorio de trabajo actual, y las funciones del sistema operativo como `AddDllDirectory` no tienen ningún efecto.

En 3.8, las dos formas principales en que se cargan las DLL ahora anulan explícitamente el comportamiento de todo el proceso para garantizar la coherencia. Ver el [notas de portabilidad](#) para obtener información sobre la actualización de bibliotecas.

`os.execl(path, arg0, arg1, ...)`

`os.execlp(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)`

Todas estas funciones ejecutan un nuevo programa, reemplazando el proceso actual; No vuelven. En Unix, el nuevo ejecutable se carga en el proceso actual y tendrá la misma identificación de proceso que la persona que llama. Los errores se informarán como excepciones `OSError`.

El proceso actual se reemplaza inmediatamente. Los objetos de archivo abierto y los descriptores no se vacían, por lo que si puede haber datos almacenados en estos archivos abiertos, debe limpiarlos usando `sys.stdout.flush()` o `os.fsync()` antes de llamar a `exec*` función.

Las variantes «l» y «v» de las funciones `exec*` difieren en cómo se pasan los argumentos de la línea de comandos. Las variantes «l» son quizás las más fáciles de trabajar si el número de parámetros se fija cuando se escribe el código; los parámetros individuales simplemente se convierten en parámetros adicionales a las funciones `execl*()`. Las variantes «v» son buenas cuando el número de parámetros es variable, y los argumentos se pasan en una lista o tupla como parámetro `args`. En cualquier caso, los argumentos del proceso secundario deben comenzar con el nombre del comando que se ejecuta, pero esto no se aplica.

Las variantes que incluyen una «p» cerca del final (`execlp()`, `execlpe()`, `execvp()`, y `execvpe()`) usarán `PATH` variable de entorno para ubicar el programa `file`. Cuando se reemplaza el entorno (utilizando uno de los siguientes variantes `exec*e` variantes, discutidas en el siguiente párrafo), el nuevo entorno se utiliza

como fuente de la variable `PATH`. Las otras variantes, `execl()`, `execle()`, `execv()`, y `execve()`, no utilizarán la variable `PATH` para localizar el ejecutable; `path` debe contener una ruta absoluta o relativa apropiada.

Para `execle()`, `execlpe()`, `execve()` y `execvpe()` (tenga en cuenta que todo esto termina en «e»), el parámetro `env` debe ser un mapeo que se utiliza para definir las variables de entorno para el nuevo proceso (se utilizan en lugar del entorno del proceso actual); las funciones `execl()`, `execlp()`, `execv()` y `execvp()` hacen que el nuevo proceso herede el entorno del proceso actual.

Para `execve()` en algunas plataformas, `path` también puede especificarse como un descriptor de archivo abierto. Es posible que esta funcionalidad no sea compatible con su plataforma; puede verificar si está disponible o no usando `os.supports_fd`. Si no está disponible, su uso generará un `NotImplementedError`.

Lanza un *evento de auditoría* `os.exec` con argumentos `ruta`, `args`, `env`.

Disponibilidad: Unix, Windows.

Nuevo en la versión 3.3: Se agregó soporte para especificar `path` como un descriptor de archivo abierto para `execve()`.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

`os._exit(n)`

Salga del proceso con el estado `n`, sin llamar a los controladores de limpieza, vaciar los buffers stdio, etc.

Nota: La forma estándar de salir es `sys.exit(n)`. `_exit()` normalmente solo debe usarse en el proceso secundario después de `fork()`.

Los siguientes códigos de salida están definidos y se pueden usar con `_exit()`, aunque no son obligatorios. Por lo general, se usan para programas del sistema escritos en Python, como el programa de entrega de comandos externos de un servidor de correo.

Nota: Es posible que algunos de estos no estén disponibles en todas las plataformas Unix, ya que hay alguna variación. Estas constantes se definen donde están definidas por la plataforma subyacente.

`os.EX_OK`

Código de salida que significa que no se produjo ningún error.

Disponibilidad: Unix.

`os.EX_USAGE`

Código de salida que significa que el comando se usó incorrectamente, como cuando se da un número incorrecto de argumentos.

Disponibilidad: Unix.

`os.EX_DATAERR`

Código de salida que significa que los datos de entrada eran incorrectos.

Disponibilidad: Unix.

`os.EX_NOINPUT`

Código de salida que significa que no existía un archivo de entrada o que no era legible.

Disponibilidad: Unix.

`os.EX_NOUSER`

Código de salida que significa que un usuario especificado no existía.

Disponibilidad: Unix.

`os.EX_NOHOST`

Código de salida que significa que no existía un host especificado.

Disponibilidad: Unix.

os.EX_UNAVAILABLE

Código de salida que significa que un servicio requerido no está disponible.

Disponibilidad: Unix.

os.EX_SOFTWARE

Código de salida que significa que se detectó un error interno de software.

Disponibilidad: Unix.

os.EX_OSERR

Código de salida que significa que se detectó un error del sistema operativo, como la imposibilidad de bifurcar o crear una tubería.

Disponibilidad: Unix.

os.EX_OSFILE

Código de salida que significa que algunos archivos del sistema no existían, no podían abrirse o tenían algún otro tipo de error.

Disponibilidad: Unix.

os.EX_CANTCREAT

Código de salida que significa que no se pudo crear un archivo de salida especificado por el usuario.

Disponibilidad: Unix.

os.EX_IOERR

Código de salida que significa que se produjo un error al realizar E / S en algún archivo.

Disponibilidad: Unix.

os.EX_TEMPFAIL

Código de salida que significa que ocurrió una falla temporal. Esto indica algo que puede no ser realmente un error, como una conexión de red que no se pudo realizar durante una operación recuperable.

Disponibilidad: Unix.

os.EX_PROTOCOL

Código de salida que significa que un intercambio de protocolo fue ilegal, inválido o no se entendió.

Disponibilidad: Unix.

os.EX_NOPERM

Código de salida que significa que no había permisos suficientes para realizar la operación (pero no para problemas del sistema de archivos).

Disponibilidad: Unix.

os.EX_CONFIG

Código de salida que significa que se produjo algún tipo de error de configuración.

Disponibilidad: Unix.

os.EX_NOTFOUND

Código de salida que significa algo así como «no se encontró una entrada».

Disponibilidad: Unix.

os.fork()

Bifurcar un proceso hijo. Retorna 0 en el niño y la identificación del proceso del niño en el padre. Si se produce un error se genera *OSError*.

Tenga en cuenta que algunas plataformas que incluyen FreeBSD <= 6.3 y Cygwin tienen problemas conocidos al usar `fork()` desde un hilo.

Lanza un *evento de auditoría* `os.fork` sin argumentos.

Distinto en la versión 3.8: Llamar a `fork()` en un subinterpretador ya no es compatible (*RuntimeError* está activado).

Advertencia: Ver `ssl` para aplicaciones que usan el módulo SSL con `fork()`.

Disponibilidad: Unix.

os.forkpty()

Bifurca un proceso hijo, usando un nuevo pseudo-terminal como terminal de control del niño. Retorna un par de `(pid, fd)`, donde `pid` es 0 en el elemento secundario, la identificación del proceso del elemento secundario nuevo en el elemento primario y `fd` es el descriptor de archivo del final maestro de El pseudo-terminal. Para un enfoque más portátil, use el módulo `pty`. Si se produce un error se genera `OSError`.

Lanza un *evento de auditoría* `os.forkpty` sin argumentos.

Distinto en la versión 3.8: Llamar a `forkpty()` en un subinterpretador ya no es compatible (`RuntimeError` está activado).

Disponibilidad: algunos sistemas tipo Unix.

os.kill(pid, sig)

Enviar señal `sig` al proceso `pid`. Las constantes para las señales específicas disponibles en la plataforma host se definen en el módulo `signal`.

Windows: Las señales `signal.CTRL_C_EVENT` y `signal.CTRL_BREAK_EVENT` son señales especiales que solo pueden enviarse a procesos de consola que comparten una ventana de consola común, por ejemplo, algunos subprocesos. Cualquier otro valor para `sig` hará que la API `TerminateProcess` elimine el proceso incondicionalmente, y el código de salida se establecerá en `sig`. La versión de Windows de `kill()` también requiere que los identificadores de proceso sean eliminados.

Ver también `signal.pthread_kill()`.

Lanza un *evento de auditoría* `os.kill` con argumentos `pid, sig`.

Nuevo en la versión 3.2: Soporte de Windows.

os.killpg(pgid, sig)

Envíe la señal `sig` al grupo de procesos `pgid`.

Lanza un *evento de auditoría* `os.killpg` con argumentos `pgid, sig`.

Disponibilidad: Unix.

os.nice(increment)

Agregue `increment` a la «simpatía» del proceso. Retorna la nueva amabilidad.

Disponibilidad: Unix.

os.plock(op)

Bloquee segmentos del programa en la memoria. El valor de `op` (definido en `<sys/lock.h>`) determina qué segmentos están bloqueados.

Disponibilidad: Unix.

os.popen(cmd, mode='r', buffering=-1)

Abra una tubería hacia o desde el comando `cmd`. El valor de retorno es un objeto de archivo abierto conectado a la tubería, que puede leerse o escribirse dependiendo de si `mode` es `'r'` (predeterminado) o `'w'`. El argumento `buffering` tiene el mismo significado que el argumento correspondiente a la función incorporada `open()`. El objeto de archivo retornado lee o escribe cadenas de texto en lugar de bytes.

El método `close` retorna `None` si el subproceso salió correctamente, o el código de retorno del subproceso si hubo un error. En los sistemas POSIX, si el código de retorno es positivo, representa el valor de retorno del proceso desplazado a la izquierda en un byte. Si el código de retorno es negativo, el proceso fue terminado por la señal dada por el valor negado del código de retorno. (Por ejemplo, el valor de retorno podría ser `-signal.SIGKILL` si se eliminó el subproceso). En los sistemas Windows, el valor de retorno contiene el código de retorno entero firmado del proceso secundario.

Esto se implementa usando `subprocess.Popen`; consulte la documentación de esa clase para obtener formas más potentes de administrar y comunicarse con subprocesos.

`os.posix_spawn(path, argv, env, *, file_actions=None, setpgroup=None, resetids=False, setsid=False, setsigmask=(), setsigdef=(), scheduler=None)`

Envuelve la API de la biblioteca C `posix_spawn()` para usar desde Python.

La mayoría de los usuarios deberían usar `subprocess.run()` en lugar de `posix_spawn()`.

Los argumentos de solo posición `path`, `args` y `env` son similares a `execve()`.

The `path` parameter is the path to the executable file. The `path` should contain a directory. Use `posix_spawnnp()` to pass an executable file without directory.

El argumento `file_actions` puede ser una secuencia de tuplas que describen acciones para tomar descriptores de archivo específicos en el proceso secundario entre los pasos de implementación de la biblioteca C `fork()` y `exec()`. El primer elemento de cada tupla debe ser uno de los tres indicadores de tipo que se enumeran a continuación y que describen los elementos de tupla restantes:

`os.POSIX_SPAWN_OPEN`

`(os.POSIX_SPAWN_OPEN, fd, path, flags, mode)`

Realiza `os.dup2(os.open(path, flags, mode), fd)`.

`os.POSIX_SPAWN_CLOSE`

`(os.POSIX_SPAWN_CLOSE, fd)`

Realiza `os.close(fd)`.

`os.POSIX_SPAWN_DUP2`

`(os.POSIX_SPAWN_DUP2, fd, new_fd)`

Realiza `os.dup2(fd, new_fd)`.

Estas tuplas corresponden a la biblioteca C `posix_spawn_file_actions_addopen()`, `posix_spawn_file_actions_addclose()`, y `posix_spawn_file_actions_adddup2()`. Llamadas API utilizadas para prepararse para `posix_spawn()` se llame a sí mismo.

El argumento `setpgroup` establecerá el grupo de proceso del elemento secundario en el valor especificado. Si el valor especificado es 0, la ID del grupo de procesos del niño se hará igual que su ID de proceso. Si el valor de `setpgroup` no está establecido, el elemento secundario heredará la ID del grupo de proceso del elemento primario. Este argumento corresponde al flag `POSIX_SPAWN_SETPGROUP` de la biblioteca de C.

Si el argumento `resetids` es `True`, restablecerá el UID y el GID efectivos del niño al UID y GID reales del proceso padre. Si el argumento es `False`, el niño conserva el UID y el GID efectivos del padre. En cualquier caso, si los bits de permiso set-user-ID y set-group-ID están habilitados en el archivo ejecutable, su efecto anulará la configuración del UID y GID efectivos. Este argumento corresponde a la flag de la biblioteca C `POSIX_SPAWN_RESETIDS`.

Si el argumento `setsid` es `True`, creará una nueva ID de sesión para `posix_spawn`. `setsid` requiere `POSIX_SPAWN_SETSID` o flag `POSIX_SPAWN_SETSID_NP`. De lo contrario, se excita `NotImplementedError`.

El argumento `setsigmask` establecerá la máscara de señal en el conjunto de señal especificado. Si no se usa el parámetro, el niño hereda la máscara de señal del padre. Este argumento corresponde al flag `POSIX_SPAWN_SETSIGMASK` de la biblioteca en C.

El argumento `sigdef` restablecerá la disposición de todas las señales en el conjunto especificado. Este argumento corresponde al flag `POSIX_SPAWN_SETSIGDEF` de la biblioteca de C.

El argumento `scheduler` debe ser una tupla que contenga la política del planificador (opcional) y una instancia de `sched_param` con los parámetros del planificador. Un valor de `None` en el lugar de la política del planificador indica que no se proporciona. Este argumento es una combinación de la biblioteca C `POSIX_SPAWN_SETSCHEDPARAM` y flags `POSIX_SPAWN_SETSCHEDULER`.

Lanza un *evento de auditoría* `os.posix_spawn` con argumentos `ruta`, `argv`, `env`.

Nuevo en la versión 3.8.

Disponibilidad: Unix.

`os.posix_spawn` (*path, argv, env, *, file_actions=None, setpgroup=None, resetids=False, setid=False, setsigmask=(), setsigdef=(), scheduler=None*)

Envuelve la API de la biblioteca `posix_spawn()` C para usar desde Python.

Similar a `posix_spawn()` excepto que el sistema busca el archivo *executable* en la lista de directorios especificada por la variable de entorno `PATH` (de la misma manera que para `execvp(3)`).

Lanza un *evento de auditoría* `os.posix_spawn` con argumentos *ruta, argv, env*.

Nuevo en la versión 3.8.

Disponibilidad: Ver documentación `posix_spawn()`.

`os.register_at_fork` (**, before=None, after_in_parent=None, after_in_child=None*)

Registra los invocables que se ejecutarán cuando se bifurca un nuevo proceso secundario utilizando `os.fork()` o API de clonación de procesos similares. Los parámetros son opcionales y solo de palabras clave. Cada uno especifica un punto de llamada diferente.

- *before* es una función llamada antes de bifurcar un proceso hijo.
- *after_in_parent* es una función llamada desde el proceso padre después de bifurcar un proceso hijo.
- *after_in_child* es una función llamada desde el proceso hijo.

Estas llamadas solo se realizan si se espera que el control regrese al intérprete de Python. Un lanzamiento típico *subprocess* no los activará ya que el niño no va a volver a ingresar al intérprete.

Las funciones registradas para su ejecución antes de la bifurcación se invocan en orden de registro inverso. Las funciones registradas para la ejecución después de la bifurcación (ya sea en el padre o en el hijo) se invocan en orden de registro.

Tenga en cuenta que las llamadas `fork()` realizadas por código C de terceros no pueden llamar a esas funciones, a menos que explícitamente llame a `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` y `PyOS_AfterFork_Child()`.

No hay forma de cancelar el registro de una función.

Disponibilidad: Unix.

Nuevo en la versión 3.7.

`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*)

Ejecute el programa *path* en un nuevo proceso.

(Tenga en cuenta que el módulo *subprocess* proporciona funciones más potentes para generar nuevos procesos y recuperar sus resultados; es preferible usar ese módulo que usar estas funciones. Compruebe especialmente la sección *Cómo reemplazar anteriores funciones con el módulo subprocess*.)

Si *mode* es `P_NOWAIT`, esta función retorna la identificación del proceso del nuevo proceso; if *mode* is `P_WAIT`, retorna el código de salida del proceso si sale normalmente, o `-signal`, donde *signal* es la señal que mató el proceso. En Windows, la identificación del proceso en realidad será el identificador del proceso, por lo que se puede usar con la función `waitpid()`.

Nota sobre VxWorks, esta función no retorna `-signal` cuando se cierra el nuevo proceso. En su lugar, genera una excepción `OSError`.

Las variantes «l» y «v» de las funciones *spawn** difieren en cómo se pasan los argumentos de la línea de comandos. Las variantes «l» son quizás las más fáciles de trabajar si el número de parámetros se fija cuando se escribe el código; los parámetros individuales simplemente se convierten en parámetros adicionales a las funciones `spawnl*` (). Las variantes «v» son buenas cuando el número de parámetros es variable, y los

argumentos se pasan en una lista o tupla como parámetro *args*. En cualquier caso, los argumentos del proceso secundario deben comenzar con el nombre del comando que se está ejecutando.

Las variantes que incluyen una segunda «p» cerca del final (*spawnlp()*, *spawnlpe()*, *spawnvp()*, y *spawnvpe()*) usarán *PATH* variable de entorno para ubicar el programa *file*. Cuando se reemplaza el entorno (usando uno de los siguientes *spawn*e* variantes, discutidas en el siguiente párrafo), el nuevo entorno se utiliza como fuente de la variable *PATH*. Las otras variantes, *spawnl()*, *spawnle()*, *spawnv()*, y *spawnve()*, no utilizarán la variable *PATH* para localizar el ejecutable; *path* debe contener una ruta absoluta o relativa apropiada.

Para *spawnle()*, *spawnlpe()*, *spawnve()*, y *spawnvpe()* (tenga en cuenta que todo esto termina en «e»), el parámetro *env* debe ser un mapeo que se utiliza para definir las variables de entorno para el nuevo proceso (se utilizan en lugar del entorno del proceso actual); las funciones *spawnl()*, *spawnlp()*, *spawnv()* y *spawnvp()* hacen que el nuevo proceso herede el entorno del proceso actual. Tenga en cuenta que las claves y los valores en el diccionario *env* deben ser cadenas; Las teclas o valores no válidos harán que la función falle, con un valor de retorno de 127.

Como ejemplo, las siguientes llamadas a *spawnlp()* y *spawnvpe()* son equivalentes:

```
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)
```

Lanza un *evento de auditoría* *os.spawn* con argumentos *mode*, *path*, *args*, *env*.

Disponibilidad: Unix, Windows. *spawnlp()*, *spawnlpe()*, *spawnvp()* y *spawnvpe()* no están disponibles en Windows. *spawnle()* y *spawnve()* no son seguros para subprocessos en Windows; le recomendamos que utilice el módulo *subprocess* en su lugar.

Distinto en la versión 3.6: Acepta un *objeto tipo ruta*.

os.P_NOWAIT

os.P_NOWAITO

Valores posibles para el parámetro *mode* para *spawn** familia de funciones. Si se da alguno de estos valores, las funciones *spawn*()* volverán tan pronto como se haya creado el nuevo proceso, con la identificación del proceso como valor de retorno.

Disponibilidad: Unix, Windows.

os.P_WAIT

Posible valor para el parámetro *mode* para *spawn** familia de funciones. Si esto se da como *mode*, las funciones *spawn*()* no volverán hasta que el nuevo proceso se haya completado y retornará el código de salida del proceso, la ejecución es exitosa, o *-signal* si una señal mata el proceso.

Disponibilidad: Unix, Windows.

os.P_DETACH

os.P_OVERLAY

Valores posibles para el parámetro *mode* para *spawn** familia de funciones. Estos son menos portátiles que los enumerados anteriormente. *P_DETACH* es similar a *P_NOWAIT*, pero el nuevo proceso se desconecta de la consola del proceso de llamada. Si se usa *P_OVERLAY*, el proceso actual será reemplazado; la función *spawn** no volverá.

Disponibilidad: Windows.

os.startfile(path[, operation])

Inicie un archivo con su aplicación asociada.

Cuando *operation* no se especifica o 'abre', esto actúa como hacer doble clic en el archivo en el Explorador de Windows, o dar el nombre del archivo como argumento para el comando **start** desde el shell de comandos interactivo: el archivo se abre con cualquier aplicación (si la hay) a la que está asociada su extensión.

Cuando se da otra *operation*, debe ser un «verbo de comando» que especifica qué se debe hacer con el archivo. Los verbos comunes documentados por Microsoft son 'print' y 'edit' (para usar en archivos), así como 'explore' y 'find' (para usar en directorios).

`startfile()` vuelve tan pronto como se inicia la aplicación asociada. No hay opción de esperar a que la aplicación se cierre y no hay forma de recuperar el estado de salida de la aplicación. El parámetro *path* es relativo al directorio actual. Si desea utilizar una ruta absoluta, asegúrese de que el primer carácter no sea una barra inclinada ('/'); la función subyacente `Win32 ShellExecute()` no funciona si lo es. Use la función `os.path.normpath()` para asegurarse de que la ruta esté codificada correctamente para Win32.

Para reducir la sobrecarga de inicio del intérprete, la función `Win32 ShellExecute()` no se resuelve hasta que esta función se llama por primera vez. Si la función no se puede resolver, se generará `NotImplementedError`.

Lanza un *evento de auditoría* `os.startfile` con argumentos *path*, *operation*.

Disponibilidad: Windows.

`os.system(command)`

Ejecute el comando (una cadena) en una subshell. Esto se implementa llamando a la función Estándar C `system()`, y tiene las mismas limitaciones. Los cambios en `sys.stdin`, etc. no se reflejan en el entorno del comando ejecutado. Si *command* genera alguna salida, se enviará al flujo de salida estándar del intérprete.

En Unix, el valor de retorno es el estado de salida del proceso codificado en el formato especificado para `wait()`. Tenga en cuenta que POSIX no especifica el significado del valor de retorno de la función C `system()`, por lo que el valor de retorno de la función Python depende del sistema.

En Windows, el valor de retorno es el que retorna el shell del sistema después de ejecutar *command*. El shell viene dado por la variable de entorno de Windows `COMSPEC`: generalmente es `cmd.exe`, que retorna el estado de salida de la ejecución del comando; En sistemas que utilizan un shell no nativo, consulte la documentación del shell.

El módulo `subprocess` proporciona instalaciones más potentes para generar nuevos procesos y recuperar sus resultados; usar ese módulo es preferible a usar esta función. Consulte la sección *Cómo reemplazar anteriores funciones con el módulo subprocess* en la documentación de `subprocess` para obtener algunas recetas útiles.

Lanza un *evento de auditoría* `os.system` con argumento *command*.

Disponibilidad: Unix, Windows.

`os.times()`

Retorna los tiempos de proceso globales actuales. El valor de retorno es un objeto con cinco atributos:

- `user` - tiempo de usuario
- `os` — Interfaces misceláneas del sistema operativo
- `children_user` - tiempo de usuario de todos los procesos secundarios
- `children_system` - hora del sistema de todos los procesos secundarios
- `elapsed` - tiempo real transcurrido desde un punto fijo en el pasado

Por compatibilidad con versiones anteriores, este objeto también se comporta como una tupla que contiene `user`, `system`, `children_user`, `children_system`, y `elapsed` en ese orden

Consulte la página de manual de Unix `times(2)` y `times(3)` página de manual en Unix o [MSDN de GetProcessTimes](#) en Windows. En Windows, solo se conocen `user` y `system`; Los otros atributos son cero.

Disponibilidad: Unix, Windows.

Distinto en la versión 3.3: El tipo de objeto retornado cambió de una tupla a un objeto tipo tupla con atributos con nombre.

`os.wait()`

Espere a que se complete un proceso secundario y retorna una tupla que contenga su indicación de estado pid y de salida: un número de 16 bits, cuyo byte bajo es el número de señal que mató el proceso y cuyo byte alto es el estado de salida (si la señal el número es cero); el bit alto del byte bajo se establece si se produjo un archivo central.

Disponibilidad: Unix.

os.**waitid** (*idtype*, *id*, *options*)

Espere la finalización de uno o más procesos secundarios. *idtype* puede ser *P_PID*, *P_PGID* o *P_ALL*. *id* especifica el pid para esperar. *options* se construye a partir de OR de uno o más de *WEXITED*, *WSTOPPED* o *WCONTINUED* y adicionalmente se puede ORing con *WNOHANG* o *WNOWAIT*. El valor de retorno es un objeto que representa los datos contenidos en la estructura *siginfo_t*, a saber: *si_pid*, *si_uid*, *si_signo*, *si_status*, *si_code* o *None* if *WNOHANG* está especificado y no hay hijos en un estado de espera.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

os.**P_PID**

os.**P_PGID**

os.**P_ALL**

Estos son los valores posibles para *idtype* en *waitid()*. Afectan cómo se interpreta *id*.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

os.**WEXITED**

os.**WSTOPPED**

os.**WNOWAIT**

Indicadores que se pueden usar en *options* en *waitid()* que especifican qué señal secundaria esperar.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

os.**CLD_EXITED**

os.**CLD_DUMPED**

os.**CLD_TRAPPED**

os.**CLD_CONTINUED**

Estos son los valores posibles para *si_code* en el resultado retornado por *waitid()*.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

os.**waitpid** (*pid*, *options*)

Los detalles de esta función difieren en Unix y Windows.

En Unix: espere a que se complete un proceso secundario dado por la identificación del proceso *pid*, y retorna una tupla que contenga su identificación del proceso y la indicación del estado de salida (codificado como para *wait()*). La semántica de la llamada se ve afectada por el valor del número entero *options*, que debe ser 0 para el funcionamiento normal.

Si *pid* es mayor que 0, *waitpid()* solicita información de estado para ese proceso específico. Si *pid* es 0, la solicitud es para el estado de cualquier hijo en el grupo de procesos del proceso actual. Si *pid* es -1, la solicitud corresponde a cualquier elemento secundario del proceso actual. Si *pid* es menor que -1, se solicita el estado de cualquier proceso en el grupo de procesos -*pid* (el valor absoluto de *pid*).

Un *OSError* se lanza con el valor de *errno* cuando *syscall* retorna -1.

En Windows: espere a que se complete un proceso dado por el identificador de proceso *pid*, y retorna una tupla que contiene *pid*, y su estado de salida se desplazó a la izquierda en 8 bits (el desplazamiento facilita el uso de la función en la plataforma). A *pid* menor o igual que 0 no tiene un significado especial en Windows y genera una excepción. El valor de entero *options* no tiene ningún efecto. *pid* puede referirse a cualquier proceso cuya identificación sea conocida, no necesariamente un proceso hijo. Las funciones *spawn** llamadas con *P_NOWAIT* retornan manejadores de proceso adecuados.

Distinto en la versión 3.5: Si la llamada al sistema se interrumpe y el controlador de señal no genera una excepción, la función vuelve a intentar la llamada del sistema en lugar de generar una excepción *InterruptedError* (ver **PEP 475** para la justificación).

os.wait3 (options)

Similar a `waitpid()`, excepto que no se proporciona ningún argumento de identificación de proceso y se retorna una tupla de 3 elementos que contiene la identificación de proceso del niño, la indicación del estado de salida y la información de uso de recursos. Consulte `resource.getrusage()` para obtener detalles sobre la información de uso de recursos. El argumento de la opción es el mismo que se proporciona a `waitpid()` y `wait4()`.

Disponibilidad: Unix.

os.wait4 (pid, options)

Similar a `waitpid()`, excepto una tupla de 3 elementos, que contiene la identificación del proceso del niño, la indicación del estado de salida y la información de uso de recursos. Consulte `resource.getrusage()` para obtener detalles sobre la información de uso de recursos. Los argumentos para `wait4()` son los mismos que los proporcionados para `waitpid()`.

Disponibilidad: Unix.

os.WNOHANG

La opción para `waitpid()` para regresar inmediatamente si no hay un estado de proceso secundario disponible de inmediato. La función retorna `(0, 0)` en este caso.

Disponibilidad: Unix.

os.WCONTINUED

Esta opción hace que se informen los procesos secundarios si se han continuado desde una parada de control de trabajo desde la última vez que se informó su estado.

Disponibilidad: algunos sistemas Unix.

os.WUNTRACED

Esta opción hace que se informen los procesos secundarios si se han detenido pero su estado actual no se ha informado desde que se detuvieron.

Disponibilidad: Unix.

Las siguientes funciones toman un código de estado del proceso retornado por `system()`, `wait()`, o `waitpid()` como parámetro. Pueden usarse para determinar la disposición de un proceso.

os.WCOREDUMP (status)

Retorna `True` si se generó un volcado de núcleo para el proceso; de lo contrario, retorna `False`.

Esta función debe emplearse solo si `WIFSIGNALED()` es verdadero.

Disponibilidad: Unix.

os.WIFCONTINUED (status)

retorna `True` si un niño detenido se ha reanudado mediante la entrega de `SIGCONT` (si el proceso se ha continuado desde una parada de control de trabajo), de lo contrario, retorna `False`.

Ver opción `WCONTINUED`.

Disponibilidad: Unix.

os.WIFSTOPPED (status)

retorna `True` si el proceso se detuvo mediante la entrega de una señal; de lo contrario, retorna `False`.

`WIFSTOPPED()` solo retorna `True` si la llamada `waitpid()` se realizó utilizando la opción `WUNTRACED` o cuando se rastrea el proceso (consulte `ptrace(2)`)

Disponibilidad: Unix.

os.WIFSIGNALED (status)

retorna `True` si el proceso finalizó con una señal; de lo contrario, retorna `False`.

Disponibilidad: Unix.

os.WIFEXITED (status)

retorna `True` si el proceso finalizó normalmente, es decir, llamando a `exit()` o `_exit()`, o volviendo de `main()`; de lo contrario, retorna `False`.

Disponibilidad: Unix.

os.**WEXITSTATUS** (*status*)

Retorna el estado de salida del proceso.

Esta función debe emplearse solo si *WIFEXITED()* es verdadero.

Disponibilidad: Unix.

os.**WSTOPSIG** (*status*)

Retorna la señal que hizo que el proceso se detuviera.

Esta función debe emplearse solo si *WIFSTOPPED()* es verdadero.

Disponibilidad: Unix.

os.**WTERMSIG** (*status*)

Retorna el número de la señal que provocó la finalización del proceso.

Esta función debe emplearse solo si *WIFSIGNALED()* es verdadero.

Disponibilidad: Unix.

16.1.7 Interfaz al planificador

Estas funciones controlan cómo el sistema operativo asigna el tiempo de CPU a un proceso. Solo están disponibles en algunas plataformas Unix. Para obtener información más detallada, consulte las páginas de manual de Unix.

Nuevo en la versión 3.3.

Las siguientes políticas de programación están expuestas si son compatibles con el sistema operativo.

os.**SCHED_OTHER**

La política de programación predeterminada.

os.**SCHED_BATCH**

Política de programación para procesos intensivos en CPU que intenta preservar la interactividad en el resto de la computadora.

os.**SCHED_IDLE**

Política de programación para tareas en segundo plano de prioridad extremadamente baja.

os.**SCHED_SPORADIC**

Política de programación para programas de servidor esporádicos.

os.**SCHED_FIFO**

Una política de programación *First In First Out*.

os.**SCHED_RR**

Una política de programación round-robin.

os.**SCHED_RESET_ON_FORK**

Esta flag se puede OR con cualquier otra política de programación. Cuando un proceso con este indicador establece bifurcaciones, la política de programación y la prioridad de su hijo se restablecen a los valores predeterminados.

class os.**sched_param** (*sched_priority*)

Esta clase representa los parámetros de programación ajustables utilizados en *sched_setparam()*, *sched_setscheduler()* y *sched_getparam()*. Es inmutable.

Por el momento, solo hay un parámetro posible:

sched_priority

La prioridad de programación para una política de programación.

os.**sched_get_priority_min** (*policy*)

Obtiene el valor de prioridad mínimo para *policy*. *policy* es una de las constantes de política de programación anteriores.

- **`os.sched_get_priority_max`** (*policy*)
Obtiene el valor de prioridad máxima para *policy*. *policy* es una de las constantes de política de programación anteriores.
- **`os.sched_setscheduler`** (*pid*, *policy*, *param*)
Establece la política de programación para el proceso con PID *pid*. Un *pid* de 0 significa el proceso de llamada. *policy* es una de las constantes de política de programación anteriores. *param* es una instancia de `sched_param`.
- **`os.sched_getscheduler`** (*pid*)
Retorna la política de programación para el proceso con PID *pid*. Un *pid* de 0 significa el proceso de llamada. El resultado es una de las constantes de política de programación anteriores.
- **`os.sched_setparam`** (*pid*, *param*)
Establece parámetros de programación para el proceso con PID *pid*. Un *pid* de 0 significa el proceso de llamada. *param* es una instancia de `sched_param`.
- **`os.sched_getparam`** (*pid*)
Retorna los parámetros de programación como una instancia de `sched_param` para el proceso con PID *pid*. Un *pid* de 0 significa el proceso de llamada.
- **`os.sched_rr_get_interval`** (*pid*)
Retorna el round-robin quantum en segundos para el proceso con PID *pid*. Un *pid* de 0 significa el proceso de llamada.
- **`os.sched_yield`** ()
Renunciar voluntariamente a la CPU.
- **`os.sched_setaffinity`** (*pid*, *mask*)
Restringe el proceso con PID *pid* (o el proceso actual si es cero) a un conjunto de CPU. *mask* es un entero iterable que representa el conjunto de CPU a las que se debe restringir el proceso.
- **`os.sched_getaffinity`** (*pid*)
Retorna el conjunto de CPU al proceso con PID *pid* (o el proceso actual si es cero) está restringido.

16.1.8 Información miscelánea del sistema

- **`os.confstr`** (*name*)
Retorna valores de configuración del sistema con valores de cadena. *name* especifica el valor de configuración para recuperar; puede ser una cadena que es el nombre de un valor de sistema definido; estos nombres se especifican en varios estándares (POSIX, Unix 95, Unix 98 y otros). Algunas plataformas también definen nombres adicionales. Los nombres conocidos por el sistema operativo host se dan como las claves del diccionario `confstr_names`. Para las variables de configuración no incluidas en esa asignación, también se acepta pasar un número entero para *name*.

Si el valor de configuración especificado por *name* no está definido, se retorna `None`.

Si *name* es una cadena y no se conoce, se excita `ValueError`. Si el sistema host no admite un valor específico para *name*, incluso si está incluido en `confstr_names`, se lanza un `OSError` con `errno.EINVAL` para el número de error .

Disponibilidad: Unix.
- **`os.confstr_names`**
Nombres de mapeo de diccionario aceptados por `confstr()` a los valores enteros definidos para esos nombres por el sistema operativo host. Esto se puede usar para determinar el conjunto de nombres conocidos por el sistema.

Disponibilidad: Unix.
- **`os.cpu_count`** ()
Retorna el número de CPU en el sistema. Retorna `None` si no está determinado.

Este número no es equivalente al número de CPU que puede utilizar el proceso actual. El número de CPU utilizables se puede obtener con `len(os.sched_getaffinity(0))`

Nuevo en la versión 3.4.

os.getloadavg()

Retorna el número de procesos en la cola de ejecución del sistema promediada durante los últimos 1, 5 y 15 minutos o aumentos *OSError* si el promedio de carga no se pudo obtener.

Disponibilidad: Unix.

os.sysconf(name)

Retorna valores de configuración del sistema con valores enteros. Si el valor de configuración especificado por *name* no está definido, se retorna -1. Los comentarios sobre el parámetro *name* para *confstr()* se aplican aquí también; El diccionario que proporciona información sobre los nombres conocidos viene dado por *sysconf_names*.

Disponibilidad: Unix.

os.sysconf_names

Nombres de asignación de diccionario aceptados por *sysconf()* a los valores enteros definidos para esos nombres por el sistema operativo host. Esto se puede usar para determinar el conjunto de nombres conocidos por el sistema.

Disponibilidad: Unix.

Los siguientes valores de datos se utilizan para admitir operaciones de manipulación de rutas. Estos están definidos para todas las plataformas.

Las operaciones de nivel superior en los nombres de ruta se definen en el módulo *os.path*.

os.curdir

La cadena constante utilizada por el sistema operativo para referirse al directorio actual. Esto es '.' Para Windows y POSIX. También disponible a través de *os.path*.

os.pardir

La cadena constante utilizada por el sistema operativo para hacer referencia al directorio principal. Esto es '..' para Windows y POSIX. También disponible a través de *os.path*.

os.sep

El carácter utilizado por el sistema operativo para separar los componentes del nombre de ruta. Esto es '/' para POSIX y '\\' para Windows. Tenga en cuenta que saber esto no es suficiente para poder analizar o concatenar nombres de ruta — use *os.path.split()* y *os.path.join()* — pero en ocasiones es útil. También disponible a través de *os.path*.

os.altsep

Un carácter alternativo utilizado por el sistema operativo para separar los componentes del nombre de ruta, o None si solo existe un carácter separador. Esto se establece en '/' en los sistemas Windows donde *sep* es una barra invertida. También disponible a través de *os.path*.

os.extsep

El carácter que separa el nombre de archivo base de la extensión; por ejemplo, el '.' en *os.py*. También disponible a través de *os.path*.

os.pathsep

El carácter utilizado convencionalmente por el sistema operativo para separar los componentes de la ruta de búsqueda (como en PATH), como ':' para POSIX o ';' para Windows. También disponible a través de *os.path*.

os.defpath

La ruta de búsqueda predeterminada utilizada por *exec*p** y *spawn*p** si el entorno no tiene una tecla 'RUTA'. También disponible a través de *os.path*.

os.linesep

La cadena utilizada para separar (o, más bien, terminar) líneas en la plataforma actual. Este puede ser un solo carácter, como '\n' para POSIX, o varios caracteres, por ejemplo, '\r\n' para Windows. No utilice *os.linesep* como terminador de línea cuando escriba archivos abiertos en modo texto (el valor predeterminado); use un solo '\n' en su lugar, en todas las plataformas.

os.devnull

La ruta del archivo del dispositivo nulo. Por ejemplo: `'/dev/null'` para POSIX, `'nul'` para Windows. También disponible a través de `os.path`.

os.RTLD_LAZY**os.RTLD_NOW****os.RTLD_GLOBAL****os.RTLD_LOCAL****os.RTLD_NODELETE****os.RTLD_NOLOAD****os.RTLD_DEEPCBIND**

Flags para usar con las funciones `setdlopenflags()` y `getdlopenflags()`. Consulte la página del manual de Unix `dlopen(3)` para saber qué significan las diferentes flags.

Nuevo en la versión 3.3.

16.1.9 Números al azar

os.getrandom(size, flags=0)

Obtiene hasta `size` bytes aleatorios. La función puede retornar menos bytes que los solicitados.

Estos bytes se pueden usar para generar generadores de números aleatorios en el espacio del usuario o para fines criptográficos.

`getrandom()` se basa en la entropía obtenida de los controladores de dispositivos y otras fuentes de ruido ambiental. La lectura innecesaria de grandes cantidades de datos tendrá un impacto negativo en otros usuarios de los dispositivos `/dev/random` y `/dev/urandom`.

El argumento de las flags es una máscara de bits que puede contener cero o más de los siguientes valores OR juntos:: `py os.GRND_RANDOM` y `GRND_NONBLOCK`.

Consulte también la página del manual [Linux getrandom\(\)](#).

Disponibilidad: Linux 3.17 y más reciente.

Nuevo en la versión 3.6.

os.urandom(size)

Retorna una cadena de `size` bytes aleatorios adecuados para uso criptográfico.

Esta función retorna bytes aleatorios de una fuente de aleatoriedad específica del sistema operativo. Los datos retornados deben ser lo suficientemente impredecibles para las aplicaciones criptográficas, aunque su calidad exacta depende de la implementación del sistema operativo.

En Linux, si la llamada al sistema `getrandom()` está disponible, se usa en modo de bloqueo: bloquee hasta que el grupo de entropía `urandom` del sistema se inicialice (el núcleo recopila 128 bits de entropía). Ver el [PEP 524](#) para la justificación. En Linux, la función `getrandom()` puede usarse para obtener bytes aleatorios en modo sin bloqueo (usando el indicador `GRND_NONBLOCK`) o para sondear hasta que el grupo de entropía `urandom` del sistema se inicialice.

En un sistema tipo Unix, los bytes aleatorios se leen desde el dispositivo `/dev/urandom`. Si el dispositivo `/dev/urandom` no está disponible o no es legible, se genera la excepción `NotImplementedError`.

En Windows, usará `CryptGenRandom()`.

Ver también:

El módulo `secrets` proporciona funciones de nivel superior. Para obtener una interfaz fácil de usar con el generador de números aleatorios proporcionado por su plataforma, consulte `random.SystemRandom`.

Distinto en la versión 3.6.0: En Linux, `getrandom()` ahora se usa en modo de bloqueo para aumentar la seguridad.

Distinto en la versión 3.5.2: En Linux, si el syscall `getrandom()` bloquea (el grupo de entropía `urandom` aún no está inicializado), recurra a la lectura `/dev/urandom`.

Distinto en la versión 3.5: En Linux 3.17 y versiones posteriores, la llamada al sistema `getrandom()` ahora se usa cuando está disponible. En OpenBSD 5.6 y posterior, ahora se usa la función C `getentropy()`. Estas funciones evitan el uso de un descriptor de archivo interno.

os.GRND_NONBLOCK

Por defecto, cuando lee desde `/dev/random`, `getrandom()` bloquea si no hay bytes aleatorios disponibles, y cuando lee desde `/dev/urandom`, bloquea si el grupo de entropía no tiene, sin embargo, se ha inicializado.

Si se establece el indicador `GRND_NONBLOCK`, entonces `getrandom()` no se bloquea en estos casos, sino que inmediatamente genera `BlockingIOError`.

Nuevo en la versión 3.6.

os.GRND_RANDOM

Si se establece este bit, los bytes aleatorios se extraen del grupo `/dev/random` en lugar del grupo `/dev/urandom`.

Nuevo en la versión 3.6.

16.2 io — Herramientas principales para trabajar con *streams*

Código fuente: [Lib/io.py](#)

16.2.1 Resumen

El módulo `io` provee las facilidades principales de Python para manejar diferentes tipos de E/S. Hay tres diferentes tipos de E/S: *texto E/S*, *binario E/S* e *E/S sin formato*. Estas son categorías generales y varios respaldos de almacenamiento se pueden usar para cada una de ellas. Un objeto concreto perteneciendo a cualquiera de estas categorías se llama un *file object*. Otros términos comunes son *stream* y *file-like object*.

Independiente de su categoría, cada objeto *stream* también tendrá varias capacidades: puede ser solamente para lectura, solo escritura, or lectura y escritura. También permite arbitrariamente acceso aleatorio (buscando adelante o hacia atrás en cualquier lugar) o solamente acceso secuencial (por ejemplo en el caso de un *socket* o *pipe*).

Todas los *streams* son cuidadosas del tipo de datos que se les provee. Por ejemplo dando un objeto de clase `str` al método `write()` de un *stream* binaria lanzará un `TypeError`. También dándole un objeto de tipo `bytes` al método `write()` de un *stream* de tipo texto.

Distinto en la versión 3.3: Operaciones que lanzarán un `IOError` ahora lanzan `OSError`, ya que `IOError` es un alias de `OSError`.

E/S Texto

E/S de tipo texto espera y produce objetos de clase `str`. Esto significa que cuando el respaldo de almacenamiento está compuesto de forma nativa de `bytes` (como en el caso de un archivo), la codificación y decodificación de datos está hecho de forma transparente tanto como traducción opcional de caracteres de nueva línea específicos de la plataforma.

La manera más fácil de crear un *stream* de tipo texto es con el método `open()`, con la opción de especificar una codificación:

```
f = open("myfile.txt", "r", encoding="utf-8")
```

Streams de texto en memoria también están disponibles como objetos de tipo `StringIO`:

```
f = io.StringIO("some initial text data")
```

El API (interfaz de programación de aplicaciones) de *streams* tipo texto está descrito con detalle en la documentación de `TextIOBase`.

E/S Binaria

E/S binaria (también conocido como *buffered E/S*) espera *objetos tipo bytes* y produce objetos tipo *bytes*. No se hace codificación, decodificación, o traducciones de nueva línea. Esta categoría de *streams* puede ser usada para todos tipos de datos sin texto, y también cuando se desea control manual sobre el manejo de dato textual.

La manera más fácil para crear un *stream* binario es con el método `open()` con `'b'` en el modo de la cadena de caracteres:

```
f = open("myfile.jpg", "rb")
```

Los *streams* binarios en memoria también están disponibles como objetos tipo *BytesIO*:

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

El API de *stream* binario está descrito con detalle en la documentación de *BufferedIOBase*.

Otros módulos bibliotecarios pueden proveer maneras alternativas para crear *streams* de tipo texto o binario. Ver `socket.socket.makefile()` como ejemplo.

E/S sin formato

E/S sin formato (también conocido como *unbuffered E/S*) es generalmente usado como un fundamento de nivel bajo para *streams* binario y tipo texto; es raramente útil para manipular directamente *streams* sin formatos del código de usuario. Sin embargo puedes crear un *stream* sin formato abriendo un archivo en modo binario con el búfer apagado:

```
f = open("myfile.jpg", "rb", buffering=0)
```

El API de *streams* sin formato está descrito con detalle en la documentación de *RawIOBase*.

16.2.2 Interfaz de módulo de alto nivel

`io.DEFAULT_BUFFER_SIZE`

Un *int* que contiene el búfer de tamaño predeterminado usado por las clases de tipo E/S. `open()` utiliza el *blksize* del archivo (obtenido por `os.stat()`) si es posible.

`io.open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, opener=None)`

Esto es un alias para la función incorporada `open()`.

Lanza un *auditing event* `open` con los argumentos “path”, `mode`, `flags`.

`io.open_code(path)`

Abre el archivo dado con el modo `'rb'`. Esta función debe ser usado cuando la intención es tratar el contenido como código ejecutable.

`path` debe ser un *str* y una ruta absoluta.

Se puede anular el comportamiento de esta función haciendo un pedido anterior a `PyFile_SetOpenCodeHook()`. Sin embargo, asumiendo que `path` es un *str* y una ruta absoluta, `open_code(path)` debería manejarse al igual que `open(path, 'rb')`. El propósito de anular el comportamiento existe para validación adicional o para el preprocesamiento del archivo.

Nuevo en la versión 3.8.

exception `io.BlockingIOError`

Esto es un alias de compatibilidad para la incorporada excepción *BlockingIOError*.

exception `io.UnsupportedOperation`

Una excepción heredando *OSError* y *ValueError* que es generado cuando se llama a una operación no admitida en un *stream*.

Ver también:

`sys` contiene los *streams* estándar de IO `sys.stdin`, `sys.stdout`, y `sys.stderr`.

16.2.3 Jerarquía de clases

La implementación de *streams* E/S está organizada como una jerarquía de clases. Primero *abstract base classes* (ABC), que son usados para especificar las varias categorías de *streams*, luego las clases concretas proveen un *stream* estándar de implementaciones.

Nota: Las clases abstractas base también proveen implementaciones predeterminadas de algunos métodos para ayudar implementar clases de *streams* concretos. Por ejemplo, `BufferedIOBase` proporciona implementaciones no optimizadas de `readinto()` y `readline()`.

En la parte superior de la jerarquía E/S está la clase abstracta base `IOBase`. Define la interfaz básica del *stream*. Tenga en cuenta que no hay separación entre *streams* de lectura y escritura; implementaciones están permitidos lanzar `UnsupportedOperation` si no apoyan la operación.

La clase `RawIOBase` extiende `IOBase`. Maneja la lectura y escritura de bytes a un *stream*. `FileIO` subclasifica `RawIOBase` para proveer una interfaz a los archivos en el sistema de archivos de la máquina.

La `BufferedIOBase` ABC maneja el búfer en un *stream* de bytes sin formato (`RawIOBase`). Sus subclasificaciones, `BufferedWriter`, `BufferedReader`, y `BufferedRWPair` *streams* de búfer que son legible, grabable, y ambos legible y grabable. `BufferedRandom` provee un interfaz búfer a *streams* de acceso aleatorio. Otra subclasificación `BufferedIOBase`, `BytesIO`, es un *stream* de bytes en memoria.

El `TextIOBase` ABC, otra subclasificación de `IOBase`, trata con los *streams* cuyos bytes representan texto, y maneja la codificación y decodificación para cadenas de caracteres y de estos mismos. `TextIOWrapper`, que extiende a este, es un interfaz textual almacenado un *stream* sin formato amortiguado (`BufferedIOBase`). Finalmente, `StringIO` es una *stream* en memoria para texto.

Los nombres de los argumentos no son parte de la especificación, y solo los argumentos de `open()` están destinados a ser utilizados como argumentos de palabras clave.

La siguiente tabla resume los ABC proporcionado por el módulo `io` module:

ABC	Hereda	Métodos de trozos (<i>Stub</i>)	Métodos Mixin y propiedades
<code>IOBase</code>		<code>fileno</code> , <code>seek</code> , and <code>truncate</code>	<code>close</code> , <code>closed</code> , <code>__enter__</code> , <code>__exit__</code> , <code>flush</code> , <code>isatty</code> , <code>__iter__</code> , <code>__next__</code> , <code>readable</code> , <code>readline</code> , <code>readlines</code> , <code>seekable</code> , <code>tell</code> , <code>writable</code> , and <code>writelines</code>
<code>RawIOBase</code>	<code>IOBase</code>	<code>readinto</code> and <code>write</code>	Heredada <code>IOBase</code> methods, <code>read</code> , and <code>readall</code>
<code>BufferedIOBase</code>	<code>IOBase</code>	<code>detach</code> , <code>read</code> , <code>read1</code> , and <code>write</code>	Heredada <code>IOBase</code> methods, <code>readinto</code> , and <code>readinto1</code>
<code>TextIOBase</code>	<code>IOBase</code>	<code>detach</code> , <code>read</code> , <code>readline</code> , and <code>write</code>	Heredada <code>IOBase</code> methods, <code>encoding</code> , <code>errors</code> , and <code>newlines</code>

Clases base E/S

`class io.IOBase`

La clase base abstracta para todas las clases de tipo E/S, actuando sobre *streams* de *bytes*. No hay constructor público.

Esta clase provee implementaciones abstractas vacías para muchos métodos que clases que derivadas pueden anular selectivamente; la implementación predeterminada representa un archivo que no se puede leer, grabar o ser buscado.

Aunque *IOBase* no declara el método `read()` o `write()` porque sus firmas varían, implementaciones y clientes deberían considerar usar métodos como parte de la interfaz. Las implementaciones también podrían lanzar un *ValueError* (o *UnsupportedOperation*) cuando operaciones que estos no apoyan son usados.

El tipo básico usado para leer datos binarios o grabar un archivo es *bytes*. Otros *bytes-like objects* son aceptados como argumentos para métodos también. Clases de tipo E/S funcionan usando datos de tipo *str*.

Tenga en cuenta que llamando cualquier método (incluso indagaciones) en un *stream* cerrada es indefinido. En este caso implementaciones podrían lanzar un error *ValueError*.

IOBase (y sus subcalsificaciones) apoyan el protocolo iterador, significando que un objeto de clase *IOBase* puede ser iterado sobre el rendimiento de las líneas en un *stream* de datos. Líneas son definidas un poco diferente dependiendo si el *stream* es de tipo binario (produciendo *bytes*), o un *stream* de texto (produciendo cadenas de caracteres). Ver `readline()` abajo.

IOBase es también un gestor de contexto y por ende apoya la declaración `with`. En este ejemplo, *file* es cerrado después de que la declaración `with` termina—incluso si alguna excepción ocurre:

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

IOBase provee los siguientes atributos y métodos:

`close()`

Cierra el *stream*. Este método no tiene efecto si el archivo ya está cerrado. Cuando está cerrado, cualquier operación que se le haga al archivo (ej. leer or grabar) lanzará el error *ValueError*.

Como conveniencia, se permite llamar este método más que una vez. Sin embargo, solamente el primer llamado tenderá efecto.

`closed`

True si está cerrada el *stream*.

`fileno()`

Retorna el descriptor de archivo subyacente (un número de tipo entero) de el *stream* si existe. Un *OSError* se lanza si el objeto IO no tiene un archivo descriptor.

`flush()`

Vacía los buffers de grabación del *stream* si corresponde. Esto no hace nada para *streams* que son solamente de lectura o *streams* sin bloqueo.

`isatty()`

Retorna True si el *stream* es interactiva (ej., si está conectado a un terminal o dispositivo tty).

`readable()`

Retorna True si el *stream* puede ser leída. Si es False, el método `read()` lanzará un *OSError*.

`readline(size=-1)`

Lee y retorna una línea del *stream*. Si *size* (tamaño) es especificado, se capturará un máximo de ése mismo tamaño especificado en *bytes*.

El terminador de la línea siempre es `b'\n'` para archivos de tipo binario; para archivos de tipo texto el argumento *newline* para la función `open()` pueden ser usados para seleccionar las líneas terminadoras reconocidas.

readlines (*hint=-1*)

Lee y retorna una lista de líneas del *stream*. *hint* puede ser especificado para controlar el número de líneas que se lee: no se leerán más líneas si el tamaño total (en *bytes* / caracteres) de todas las líneas excede *hint*.

Tenga en cuenta que ya es posible iterar sobre objetos de archivo usando `for line in file: ...` sin llamar `file.readlines()`.

seek (*offset, whence=SEEK_SET*)

Cambiar la posición del *stream* al dado *byte offset*. *offset* se interpreta en relación con la posición indicada por *whence*. El valor dado para *whence* es `SEEK_SET`. Valores para *whence* son:

- `SEEK_SET` o 0 – inicio del *stream* (el dado); *offset* debería ser cero o positivo
- `SEEK_CUR` o 1 – posición actual del *stream*; *offset* puede ser negativo
- `SEEK_END` o 2 – fin del *stream*; *offset* is usualmente negativo

Retorna la nueva posición absoluta.

Nuevo en la versión 3.1: Los constantes “`SEEK_*`”.

Nuevo en la versión 3.3: Algunos sistemas operativos pueden apoyar valores adicionales, como `os.SEEK_HOLE` o `os.SEEK_DATA`. Los valores válidos para un archivo podrían depender de que esté abierto en modo texto o binario.

seekable ()

Retorna `True` si el *stream* apoya acceso aleatorio. Si retorna `False`, `seek()`, `tell()` y `truncate()` lanzarán `OSError`.

tell ()

Retorna la posición actual del *stream*.

truncate (*size=None*)

Cambia el tamaño del *stream* al *size* dado en *bytes* (o la posición actual si no se especifica *size*). La posición actual del *stream* no se cambia. Este cambio de tamaño puede incrementar o reducir el tamaño actual del archivo. En caso de extensión, los contenidos del área del nuevo archivo depende de la plataforma (en la mayoría de los sistemas *bytes* adicionales son llenos de cero). Se retorna el nuevo tamaño del archivo.

Distinto en la versión 3.5: *Windows* llenará los archivos con cero cuando extienda.

writable ()

Retorna `True` si el *stream* apoya grabación. Si retorna `False`, `write()` y `truncate()` lanzarán `OSError`.

writelines (*lines*)

Escribir una lista de líneas al *stream*. No se agrega separadores de líneas, así que es usual que las líneas tengan separador al final.

__del__ ()

Prepara para la destrucción de un objeto. *IOBase* proporciona una implementación dada de este método que ejecuta las instancias del método `close()`.

class io.RawIOBase

Clase base para binarios de tipo E/S sin formato. Hereda la clase *IOBase*. No hay constructor público.

E/S binario sin formato proporciona acceso de nivel bajo a algún dispositivo del sistema operativo o API, y no intenta de encapsular primitivos de alto nivel (esta función se le deja a E/S de tipo búfer y E/S de tipo texto; esto se describe más adelante).

Además de los atributos y métodos de *IOBase*, la clase *RawIOBase* proporciona los siguientes métodos:

read (*size=-1*)

Lee hasta el *size* de los *bytes* del objeto y los retorna. Como conveniencia si no se especifica *size* o es -1, se retornan todos los *bytes* hasta que se retorne el fin del archivo. Sino, se hace solo un llamado al sistema. Se pueden retornar menos de *size bytes* si la llamada del sistema operativo retorna menos de *size bytes*.

Si se retorna 0 *bytes* y el *size* no era 0, esto indica que es el fin del archivo. Si el objeto está en modo sin bloqueo y no hay *bytes* disponibles, se retorna `None`.

La implementación dada difiera al método `readall()` y `readinto()`.

readall()

Lee y retorna todos los *bytes* del *stream* hasta llegar al fin del archivo, usando, si es necesario, varias llamadas al *stream*.

readinto(b)

Lee *bytes* en objeto pre-asignado y grabable *bytes-like object* *b*, y retorna el número de *bytes* leído. Por ejemplo, *b* puede ser una clase de tipo `bytearray`. Si el objeto está en modo sin bloquear y no hay *bytes* disponibles, se retorna `None`.

write(b)

Escribe *bytes-like object* dado, *b*, al *stream* subyacente y retorna la cantidad de *bytes* grabadas. Esto puede ser menos que la longitud de *b* en *bytes*, dependiendo de la especificaciones del *stream* subyacente, especialmente si no está en modo no-bloqueo. `None` se retorna si el *stream* sin formato está configurado para no bloquear y ningún *byte* puede ser rápidamente grabada. El llamador puede deshacer o mutar *b* después que retorne este método, así que la implementación solo debería acceder *b* durante la ejecución al método.

class io.BufferedReader

Clase base para *streams* binarios que apoyan algún tipo de búfer. Hereda `IOBase`. No hay constructor público.

La diferencia principal de `RawIOBase` es que los métodos `read()`, `readinto()` y `write()` intentarán (respectivamente) leer la cantidad de información solicitada o consumir toda la salida dada, a expensas de hacer más de una llamada al sistema.

Adicionalmente, esos métodos pueden lanzar un `BlockingIOError` si el *stream* sin formato subyacente está en modo no bloqueo y no puede obtener o dar más datos; a diferencia de sus contrapartes `RawIOBase`, estos nunca retornarán `None`.

Además, el método `read()` no tiene una implementación dada que difiere al método `readinto()`.

Una implementación típica de `BufferedReader` no debería heredar una implementación de `RawIOBase`, es más, debería envolver como uno, así como hacen las clases `BufferedWriter` y `BufferedReader`.

`BufferedReader` provee o anula estos métodos y atributos en adición a los de `IOBase`:

raw

El *stream* sin formato subyacente (una instancia `RawIOBase`) que `BufferedReader` maneja. Esto no es parte de la API `BufferedReader` y posiblemente no exista en algunas implementaciones.

detach()

Separa el *stream* subyacente del búfer y lo retorna.

Luego que el *stream* sin formato ha sido separado, el búfer está en un estado inutilizable.

Algunos búfer, como `BytesIO`, no tienen el concepto de un *stream* sin formato singular para retornar de este método. Lanza un `UnsupportedOperation`.

Nuevo en la versión 3.1.

read(size=-1)

Lee y retorna hasta *size* en *bytes*. Si el argumento está omitido, `None`, o es negativo, los datos son leídos y retornados hasta que se alcance el fin del archivo. Un objeto *bytes* vacío se retorna si el *stream* está al final del archivo.

Si el argumento es positivo, y el *stream* subyacente no es interactiva, varias lecturas sin formato pueden ser otorgadas para satisfacer la cantidad de *byte* (al menos que primero se llegue al fin del archivo). Pero para los *streams* sin formato interactivas, a lo sumo una lectura sin formato será emitida y un resultado corto no implica que se haya llegado al fin del archivo.

Un `BlockingIOError` se lanza si el *stream* subyacente está en modo no bloqueo y no tiene datos al momento.

read1([size])

Lee y retorna hasta *size* en *bytes* con al menos una llamada al método `read()` (o `readinto()`) del

stream subyacente. Esto puede ser útil si estás implementando tu propio búfer por encima de un objeto *BufferedIOBase*.

Si *size* es `-1` (el valor dado) se retorna un monto arbitrario de *bytes* (más que cero al menos que se haya llegado al fin del archivo).

readinto(*b*)

Lee *bytes* a un objeto predeterminado y grabable *bytes-like object b* y retorna el número de *bytes* leídos. Por ejemplo, *b* puede ser un *bytearray*.

Como *read()*, varias lecturas pueden ser otorgadas al *stream* sin formato subyacente al menos que esto último sea interactivo.

Un *BlockingIOError* se lanza si el *stream* subyacente está en modo no bloqueo y no tiene datos al momento.

readinto1(*b*)

Leer *bytes* a un objeto predeterminado y grabable *bytes-like object b* usando por lo menos una llamada al método *read()* (o *readinto()*) del *stream* subyacente. Retorna la cantidad de *bytes* leídas.

Un *BlockingIOError* se lanza si el *stream* subyacente está en modo no bloqueo y no tiene datos al momento.

Nuevo en la versión 3.5.

write(*b*)

Escribe el *bytes-like object* dado, *b*, y retorna el número de bytes grabados (siempre el equivalente en longitud de *b* en bytes, ya que si falla la grabación se lanza un *OSError*). Dependiendo en la implementación actual estos bytes pueden ser grabados rápidamente al *stream* subyacente o mantenido en un búfer por razones de rendimiento y latencia.

Cuando estás en modo no bloqueo, se lanza un *BlockingIOError* si los datos tenían que ser grabadas al *stream* sin formato pero no pudo aceptar todos los datos sin bloquear.

El llamador puede otorgar o mutar *b* después que este método retorne algo, entonces la implementación debería acceder solamente a *b* durante la llamada al método.

Archivo sin formato E/S

class `io.FileIO` (*name*, *mode*=`'r'`, *closefd*=`True`, *opener*=`None`)

FileIO representa un archivo de nivel OS conteniendo datos en *bytes*. Implementa la interfaz *RawIOBase* (y por ende también la interfaz *IOBase*).

El *name* puede ser una de dos cosas:

- una cadena de caracteres u objeto de tipo *bytes* representando la ruta del archivo en la que fue abierto. En este caso *closefd* es `True` (el valor dado) de otra manera un error será dada.
- un *integer* representando el número de descriptores de archivos de nivel OS que resultan dando acceso a través del objeto *FileIO*. Cuando el objeto *FileIO* está cerrado este fd cerrará también a no ser que *closefd* esté configurado a `False`.

El *mode* puede ser `'r'`, `'w'`, `'x'` o `"a"` para lectura (el valor dado), grabación, creación exclusiva o anejando. Si no existe el archivo se creará cuando se abra para grabar o anejar; se truncará cuando se abra para grabar. Se lanzará un error *FileExistsError* si ya existe cuando se abra para crear. Abriendo un archivo para crear implica grabar entonces este modo se comporta similarmente a `'w'`. Agrega un `'+'` al modo para permitir lectura y grabación simultáneas.

Los métodos *read()* (cuando se llama con un argumento positivo), *readinto()* y *write()* en esta clase harán solo una llamada al sistema.

Un abridor personalizado puede ser usado pasando un llamador como *opener*. El descriptor de archivo subyacente es obtenido llamando *opener* con (*name*, *flags*). *opener* debe retornar un descriptor de archivo abierto (pasando *os.open* como *opener* resulta con funcionamiento similar a pasando `None`).

El archivo recién creado es *non-inheritable*.

Ver la función incorporada `open()` para ejemplos usando el parámetro `opener` .

Distinto en la versión 3.3: El parámetro `opener` fue agregado. El modo `'x'` fue agregado.

Distinto en la versión 3.4: El archivo ahora no es heredable.

Además de los atributos y métodos de las clases `IOBase` y `RawIOBase`, `FileIO`, estos proveen los siguientes atributos:

mode

El modo dado en el constructor.

name

El nombre del archivo. Este es el descriptor del archivo cuando no se proporciona ningún nombre en el constructor.

Streams almacenados (búfer)

Streams E/S almacenadas (búfer) proveen una interfaz de más alto nivel a un dispositivo E/S que a un E/S sin formato.

class `io.BytesIO([initial_bytes])`

Una implementación de *stream* usando búferes de *bytes* en memoria. Hereda `BufferedIOBase`. El búfer está descartado cuando se llama al método `close()`.

El argumento opcional `initial_bytes` es un *bytes-like object* que contiene datos iniciales.

`BytesIO` provee o anula estos métodos además de los de `BufferedIOBase` y `IOBase`:

getbuffer()

Retorna una vista legible y grabable acerca de los contenidos del búfer sin copiarlos. Además mutando la vista actualizará de forma transparente los contenidos del búfer:

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

Nota: Mientras exista la vista el objeto `BytesIO` no se le puede cambiar el tamaño o cerrado.

Nuevo en la versión 3.2.

getvalue()

Retorna *bytes* que contiene los contenidos enteros del búfer.

read1([size])

En la clase `BytesIO` esto es lo mismo que `read()`.

Distinto en la versión 3.7: Ahora es opcional el argumento `size`.

readinto1(b)

En la clase `BytesIO` esto es lo mismo que `readinto()`.

Nuevo en la versión 3.5.

class `io.BufferedReader(raw, buffer_size=DEFAULT_BUFFER_SIZE)`

Un búfer dando un nivel de alto acceso a un objeto `RawIOBase` legible y secuencial. Hereda `BufferedIOBase`. Al leer datos de este objeto se puede solicitar una mayor cantidad de datos del *stream* sin formato subyacente, y mantener el búfer interno. Los datos que han pasado por el proceso de búfer pueden ser retornados directamente en lecturas posteriores.

El constructor crea un `BufferedReader` para el *stream* legible sin formato `raw` y `buffer_size`. Si se omite `buffer_size` se usa `DEFAULT_BUFFER_SIZE`.

BufferedReader provee o anula los métodos en adición a los de *BufferedIOBase* y *IOBase*:

peek (*[size]*)

Retorna *bytes* del *stream* sin avanzar la posición. Al menos una lectura se hace al *stream* sin formato para satisfacer el llamado. El número de bytes retornados puede ser menor o mayor al solicitado.

read (*[size]*)

Lee y retorna *size bytes* o si no se da *size*, o es negativo, hasta el fin del archivo o si la llamada leída podría bloquear in modo no bloquear.

read1 (*[size]*)

Lee y retorna hasta el tamaño *size* en *bytes* con solo un llamado al *stream*. Si al menos un *byte* pasa por el proceso de búfer, solo se retornan *buffered bytes*. De lo contrario se realiza un llamado de lectura de un *stream* sin formato.

Distinto en la versión 3.7: Ahora es opcional el argumento *size*.

class `io.BufferedReader` (*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

Un búfer que proporciona un nivel alto de acceso a un objeto *RawIOBase* grabable y secuencial. Hereda *BufferedIOBase*. Al escribir a este objeto, los datos normalmente se colocan en un búfer interno. El búfer se escribirá en el objeto *RawIOBase* subyacente bajo varias condiciones, incluyendo:

- cuando el búfer se vuelve demasiado pequeño para todos los datos pendientes;
- cuando se llama *flush()*;
- cuando se pide un método *seek()* (para *BufferedRandom* objects);
- cuando el objeto *BufferedReader* is cerrado o anulado.

El constructor crea un *BufferedReader* para el *stream* grabable *raw*. Si no es dado el *buffer_size*, recurre el valor *DEFAULT_BUFFER_SIZE*.

BufferedReader provee o anula estos métodos además de los de *BufferedIOBase* y *IOBase*:

flush ()

Forzar bytes retenidos en el búfer al *stream* sin formato. Un *BlockingIOError* debería ser lanzado si el *stream* sin formato bloquea.

write (*b*)

Escribe el *bytes-like object*, *b*, y retorna el número de bytes grabados. Cuando estás en modo no-bloqueo, se lanza un *BlockingIOError* si el búfer tiene que ser escrito pero el *stream* sin formato bloquea.

class `io.BufferedReader` (*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

Una interfaz búfer para *streams* de acceso aleatorio. Hereda *BufferedReader* y *BufferedWriter*.

El constructor crea un lector y una grabación para un *stream* sin formato buscable, dado en el primer argumento. Si se omite el *buffer_size* este recae sobre el valor predeterminado *DEFAULT_BUFFER_SIZE*.

BufferedReader es capaz de todo lo que puede hacer *BufferedReader* o *BufferedWriter*. Adicionalmente, se garantiza implementar *seek()* y *tell()*.

class `io.BufferedReader` (*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*)

Un objeto búfer E/S combinando dos objetos *RawIOBase* unidireccionales – uno legible y el otro escribible – a un punto final singular bidireccional. Hereda *BufferedIOBase*.

reader y *writer* son objetos *RawIOBase* que son respectivamente legibles y escribibles. Si se omite *buffer_size* este se recae sobre el valor predeterminado *DEFAULT_BUFFER_SIZE*.

BufferedReader implementa todos los métodos de *BufferedIOBase* excepto por *detach()*, que lanza un *UnsupportedOperation*.

Advertencia: *BufferedReader* no intenta sincronizar accesos al *stream* sin formato subyacente. No debes pasar el mismo objeto como legible y escribible; usa *BufferedReader* en su lugar.

E/S Texto

class `io.TextIOBase`

Clase base para los *streams* de tipo texto. Esta clase proporciona una interfaz basada en caracteres y líneas para transmitir E/S. Hereda `IOBase`. No hay constructor público.

`TextIOBase` provee o anula estos atributos y métodos de datos además de los de `IOBase`:

encoding

El nombre de la codificación utilizada para decodificar los *bytes* del *stream* a cadenas de caracteres y para codificar cadenas de caracteres en bytes.

errors

La configuración de error del decodificador o codificador.

newlines

Una cadena de caracteres, una tupla de cadena de caracteres, o `None`, indicando las nuevas líneas traducidas hasta ese momento. Dependiendo de la implementación y los indicadores iniciales del constructor, esto puede no estar disponible.

buffer

El búfer binario subyacente (una instancia `BufferedIOBase`) que maneja `TextIOBase`. Esto no es parte del API de `TextIOBase` y puede no existir en algunas implementaciones.

detach()

Separa el búfer binario subyacente de `TextIOBase` y lo retorna.

Una vez que se ha separado el búfer subyacente, la `TextIOBase` está en un estado inutilizable.

Algunas implementaciones de `TextIOBase`, como `StringIO`, puede no tener el concepto de un búfer subyacente y llamar a este método se lanzará `UnsupportedOperation`.

Nuevo en la versión 3.1.

read (*size=-1*)

Lee y retorna como máximo *size* caracteres del *stream* como un `str` singular. Si *size* es negativo o `None`, lee hasta llegar al fin del archivo.

readline (*size=-1*)

Leer hasta la nueva línea o fin del archivo y retorna un `str` singular. Si el *stream* está al fin del archivo una cadena de caracteres se retorna.

Si se especifica *size* como máximo *size* de caracteres será leído.

seek (*offset, whence=SEEK_SET*)

Cambia la posición del *stream* dada *offset*. El comportamiento depende del parámetro *whence*. El valor dado de *whence* es `SEEK_SET`.

- `SEEK_SET` o 0: buscar el inicio del *stream* (el dado); *offset* debería ser un número dado por `TextIOBase.tell()`, o cero. Cualquier otro valor *offset* produce comportamiento indefinido.
- `SEEK_CUR` o 1: buscar la posición actual; *offset* debería ser cero, que es una operación no (no se apoya ningún otro valor).
- `SEEK_END` o 2: buscar el fin del *stream*; *offset* debería ser cero (cualquier otro valor no es apoyado).

Retorna la nueva posición absoluta como un número opaco.

Nuevo en la versión 3.1: Los constantes "`SEEK_*`".

tell()

Retorna la posición actual de la secuencia como un número opaco. El número no suele representar una cantidad de bytes en el almacenamiento binario subyacente.

write (*s*)

Escribe la cadena de caracteres *s* al *stream* y retorna el número de caracteres grabadas.

class `io.TextIOWrapper` (*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)

Una secuencia de texto almacenado sobre un *stream* binario *BufferedIOBase*. Hereda *TextIOBase*.

encoding da el nombre de la codificación con que el *stream* será codificada o decodificado. Se da al valor predeterminado `locale.getpreferredencoding(False)`.

errors es una cadena de caracteres opcional que especifica cómo se manejan los errores codificados y decodificados. Pasa 'strict' para lanzar una excepción *ValueError* si hay un error codificado (el valor predeterminado *None* tiene el mismo efecto), o pasa 'ignore' para ignorar errores. (Tenga en cuenta que ignorar errores puede llevar a perder datos). 'replace' causa un marcador de reemplazo (como '?') para ser insertado donde haya datos mal formados. 'backslashreplace' reemplaza datos mal formados con una secuencia de escape de tipo barra invertida. Cuando se escribe, 'xmlcharrefreplace' (reemplazar con la referencia apropiada de XML) o 'namereplace' (reemplazar con secuencias de caracteres `\N{...}`) pueden ser usadas. Cualquier otro nombre de manejador de errores que hayan sido registrados con `codecs.register_error()` también son válidas.

newline controla cómo finalizar las terminaciones de líneas. Pueden ser *None*, ' ', '\n', '\r', and '\r\n'. Funciona de la siguiente manera:

- Al leer la entrada del *stream*, si *newline* es *None*, el modo *universal newlines* está habilitada. Líneas en la entrada pueden terminar en '\n', '\r', o '\r\n', y estos son traducidos a '\n' antes de ser retornados al llamador. Si es ' ', se habilita el modo universal de nuevas líneas, pero las terminaciones de líneas son retornados al llamador sin traducir. Si tiene cualquiera de los otros valores legales, líneas de ingreso son terminadas por la cadena de caracteres dada, y la línea final se retorna al llamado sin traducir.
- Al escribir la salida al *stream*, si *newline* es *None*, cualquier carácter '\n' escrito son traducidos a la línea separador *default* del sistema, `os.linesep`. Si *newline* es ' ' o '\n', la traducción no ocurre. Si *newline* es de cualquier de los otros valores legales, cualquier carácter '\n' escrito es traducido a la cadena de caracteres dada.

Si *line_buffering* es *True*, se implica `flush()` cuando una llamada a grabar contiene un carácter de nueva línea o un retorno.

Si *write_through* es *True*, llamadas a `write()` no garantizan ser pasados por el proceso de búfer: cualquier dato grabado en el objeto *TextIOWrapper* es inmediatamente manejado por el *buffer* binario subyacente.

Distinto en la versión 3.3: Se ha agregado el argumento *write_through*.

Distinto en la versión 3.3: La codificación (*encoding*) por defecto es ahora `locale.getpreferredencoding(False)` en vez de `locale.getpreferredencoding()`. No cambie temporalmente la codificación local usando `locale.setlocale()`, use la codificación local actual en vez del preferido del usuario.

TextIOWrapper provee estos miembros además de las de *TextIOBase* y sus padres:

line_buffering

Si el almacenamiento en línea está habilitado.

write_through

Si grabaciones son pasadas inmediatamente al búfer binario subyacente.

Nuevo en la versión 3.7.

reconfigure (**, encoding[, errors[, newline[, line_buffering[, write_through]*])

Reconfigura este *stream* textual usando las nuevas configuraciones de *encoding*, *errors*, *newline*, *line_buffering* y *write_through*.

Los parámetros que no son especificados mantienen las configuraciones actuales, excepto por *errors*='strict' cuando *encoding* se especifica pero *errors* no está especificado.

No es posible cambiar la codificación o nueva línea si algunos datos han sido captados por el *stream*. Sin embargo, cambiando la codificación después de grabar es posible.

Este método hace una nivelación implícita del *stream* antes de configurar los nuevos parámetros.

Nuevo en la versión 3.7.

class `io.StringIO` (*initial_value*="", *newline*='\n')

Un *stream* en memoria para E/S de tipo texto. El búfer de tipo texto es descartado cuando se llama al método `close()`.

El valor inicial del búfer puede ser configurado dando *initial_value*. Si la traducción de la nueva línea es habilitado, nuevas líneas serán codificado como si fuera por `write()`. El *stream* está posicionado al inicio del búfer.

El argumento *newline* funciona como `TextIOWrapper`. El valor dado es considerado solamente como caracteres `\n` al final de las líneas y no hacen ninguna traducción de nuevas líneas. Si *newline* está configurado con `None`, nuevas líneas son escritas como `\n` en todas las plataformas, pero descodificación universal de las nuevas líneas todavía se realiza cuando se lee.

`StringIO` provee este método además de los de `TextIOBase` y sus padres:

getvalue()

Retorna un `str` que contiene el contenido entero de los búfer. Nuevas líneas son descodificados como si fuera `read()`, aunque la posición de la transmisión no haya cambiado.

Ejemplos de uso:

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# 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()
```

class `io.IncrementalNewlineDecoder`

Un códec auxiliar que descodifica nuevas líneas para el modo *universal newlines*. Hereda `codecs.IncrementalDecoder`.

16.2.4 Rendimiento

Esta sección discute el rendimiento de las implementaciones concretas de E/S proporcionadas.

E/S Binaria

Leyendo y grabando solamente grandes porciones de datos incluso cuando el usuario pide para solo un byte, E/S de tipo búfer esconde toda ineficiencia llamando y ejecutando las rutinas E/S del sistema operativo que no ha pasado por el proceso de búfer. La ganancia depende en el OS y el tipo de E/S que se ejecuta. Por ejemplo, en algunos sistemas operativos modernos como Linux, un disco E/S sin búfer puede ser rápido como un E/S búfer. Al final, sin embargo, es que el E/S búfer ofrece rendimiento predecible independientemente de la plataforma y el dispositivo de respaldo. Entonces es siempre preferible user E/S búfer que E/S sin búfer para datos binarios.

E/S Texto

E/S de tipo texto por sobre un almacenamiento binario (como un archivo) es más lento que un E/S binario sobre el mismo almacenamiento porque requiere conversiones entre unicode y datos binarios usando un códec de caracteres. Esto puede ser notable al manejar datos enormes de texto como archivos de registro. También `TextIOWrapper.tell()` y `TextIOWrapper.seek()` son bastante lentos debido al uso del algoritmo de reconstrucción.

`StringIO`, sin embargo, es un contenedor unicode nativo en memoria y exhibirá una velocidad similar a `BytesIO`.

Multihilo

objetos `FileIO` son seguros para subprocessos en la medida en que las llamadas al sistema operativo (como `read(2)` en Unix) que envuelven también son seguros para subprocessos.

Objetos binarios búfer (instancias de `BufferedReader`, `BufferedWriter`, `BufferedRandom` y `BufferedRWPair`) protegen sus estructuras internas usando un bloqueo; es seguro, entonces, llamarlos de varios hilos a la vez.

objetos `TextIOWrapper` no son seguros para subprocessos.

Reentrada

Objetos binarios búfer (instancias de `BufferedReader`, `BufferedWriter`, `BufferedRandom` y `BufferedRWPair`) no son reentrante. Mientras llamadas reentrantes no ocurren en situaciones normales pueden surgir haciendo E/S en un manejador `signal`. Si un hilo trata de entrar de nuevo a un objeto búfer que se está accediendo actualmente, se lanza un `RuntimeError`. Tenga en cuenta que esto no prohíbe un hilo diferente entrando un objeto búfer.

Lo de arriba de forma implícita extiende a los archivos de tipo texto ya que la función `open()` envolverá un objeto almacenado en un `TextIOWrapper`. Esto incluye `streams` estándar y por ende afecta la función `print()` también.

16.3 time — Tiempo de acceso y conversiones

Este módulo proporciona varias funciones relacionadas con el tiempo. Para la funcionalidad relacionada, consulte también los módulos `datetime` y `calendar`.

Aunque este módulo siempre está disponible, no todas las funciones están disponibles en todas las plataformas. La mayoría de las funciones definidas en este módulo llaman a la plataforma de biblioteca C funciones con el mismo nombre. A veces puede ser útil consultar la documentación de la plataforma, ya que la semántica de estas funciones varía entre plataformas.

Una explicación de algunas terminologías y convenciones está en orden.

- El *epoch* es el punto donde comienza el tiempo, y depende de la plataforma. Para Unix, la época es el 1 de enero de 1970, 00:00:00 (UTC). Para averiguar cuál es la época en una plataforma determinada, mire `time.gmtime(0)`.
- El término *seconds since the epoch* se refiere al número total de segundos transcurridos desde la época, excluyendo típicamente *leap seconds*. Los segundos bisiestos se excluyen de este total en todas las plataformas compatibles con POSIX.
- Es posible que las funciones de este módulo no manejen fechas y horas antes de la época o mucho en el futuro. El punto de corte en el futuro está determinado por la biblioteca C; para sistemas de 32 bits, normalmente es en 2038.
- Función `strptime()` puede analizar años de 2 dígitos cuando se le da el código de formato `%y`. Cuando se analizan los años de 2 dígitos, se convierten de acuerdo con los estándares POSIX e ISO C: los valores 69–99 se asignan a 1969–1999, y los valores 0–68 se asignan a 2000–2068.

- UTC es la hora universal coordinada (anteriormente conocida como hora media de Greenwich, o GMT). El acrónimo UTC no es un error, sino un compromiso entre inglés y francés.
- El horario de verano es el horario de verano, un ajuste de la zona horaria por (generalmente) una hora durante parte del año. Las reglas DST son mágicas (determinadas por la ley local) y pueden cambiar de un año a otro. La biblioteca C tiene una tabla que contiene las reglas locales (a menudo se lee desde un archivo del sistema para la flexibilidad) y es la única fuente de verdadera sabiduría en este sentido.
- La precisión de las diversas funciones en tiempo real puede ser inferior a la sugerida por las unidades en las que se expresa su valor o argumento. P.ej. En la mayoría de los sistemas Unix, el reloj «funciona» solo 50 o 100 veces por segundo.
- Por otro lado, la precisión de `time()` y `sleep()` es mejor que sus equivalentes Unix: los tiempos se expresan como números de coma flotante, `time()` retorna el tiempo más preciso disponible (usando Unix `gettimeofday()` donde esté disponible) y `sleep()` aceptará un tiempo con una fracción distinta de cero (Unix `select()` se usa para implementar esto, donde esté disponible).
- El valor de tiempo retornado por `gmtime()`, `localtime()`, y `strptime()`, y aceptado por `asctime()`, `mktime()` y `strftime()`, es una secuencia de 9 enteros. Los valores de retorno de `gmtime()`, `localtime()`, y `strptime()` también ofrecen nombres de atributos para campos individuales.

Ver `struct_time` para una descripción de estos objetos.

Distinto en la versión 3.3: El tipo `struct_time` se extendió para proporcionar los atributos `tm_gmtoff` y `tm_zone` cuando la plataforma admite los miembros correspondientes de `struct tm`.

Distinto en la versión 3.6: Los atributos `struct_time` `tm_gmtoff` y `tm_zone` ahora están disponibles en todas las plataformas.

- Use las siguientes funciones para convertir entre representaciones de tiempo:

Desde	A	Usar
segundos desde la época	<code>struct_time</code> en UTC	<code>gmtime()</code>
segundos desde la época	<code>struct_time</code> en hora local	<code>localtime()</code>
<code>struct_time</code> en UTC	segundos desde la época	<code>calendar.timegm()</code>
<code>struct_time</code> en hora local	segundos desde la época	<code>mktime()</code>

16.3.1 Las Funciones

`time.asctime([t])`

Convierta una tupla o `struct_time` que represente una hora retornada por `gmtime()` o `localtime()` en una cadena de la siguiente forma: 'Dom 20 de junio 23:21:05 1993'. El campo del día tiene dos caracteres de largo y se rellena con espacio si el día es un solo dígito, por ejemplo: 'Miercoles 9 de Julio 04:26:40 1993'.

Si no se proporciona `t`, se utiliza la hora actual retornada por `localtime()`. La información regional no es utilizada por `asctime()`.

Nota: A diferencia de la función C del mismo nombre, `asctime()` no agrega una nueva línea final.

`time.thread_getcpuclockid(thread_id)`

Retorna el `clk_id` del reloj de tiempo de CPU específico del subproceso para el `thread_id` especificado.

Utilice `threading.get_ident()` o el atributo `ident` de objetos `threading.Thread` para obtener un valor adecuado para `*thread_id*`.

Advertencia: Pasar un `*thread_id*` no válido o caducado puede provocar un comportamiento indefinido, como un error de segmentación.

Disponibilidad: Unix (consulte la página de manual para: manpage: `pthread_getcpuclockid (3)` para más información).

Nuevo en la versión 3.7.

`time.clock_getres (clk_id)`

Retorna la resolución (precisión) del reloj especificado `clk_id`. Consulte *Constantes de ID de reloj* para obtener una lista de los valores aceptados para `clk_id`.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

`time.clock_gettime (clk_id) → float`

Retorna la hora del reloj especificado `clk_id`. Consulte *Constantes de ID de reloj* para obtener una lista de los valores aceptados para `clk_id`.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

`time.clock_gettime_ns (clk_id) → int`

Similar a `clock_gettime ()` pero retorna el tiempo en nanosegundos.

Disponibilidad: Unix.

Nuevo en la versión 3.7.

`time.clock_settime (clk_id, time: float)`

Establece la hora del reloj especificado `clk_id`. Actualmente, `CLOCK_REALTIME` es el único valor aceptado para `clk_id`.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

`time.clock_settime_ns (clk_id, time: int)`

Similar a `clock_settime ()` pero establece el tiempo con nanosegundos.

Disponibilidad: Unix.

Nuevo en la versión 3.7.

`time.ctime ([secs])`

Convierta un tiempo expresado en segundos desde la época en una cadena de una forma: 'Dom 20 de junio 23:21:05 1993' que representa la hora local. El campo del día tiene dos caracteres de largo y se rellena con espacio si el día es de un solo dígito, por ejemplo: 'Miercoles Junio 9 04:26:40 1993'.

Si no se proporciona `secs` o `None`, se utiliza la hora actual retornada por `time ()`. `ctime (secs)` es equivalente a `asctime (localtime (secs))`. La información de configuración regional no la utiliza `ctime ()`.

`time.get_clock_info (name)`

Obtenga información sobre el reloj especificado como un objeto de espacio de nombres. Los nombres de reloj admitidos y las funciones correspondientes para leer su valor son:

- 'monotonic': `time.monotonic ()`
- 'perf_counter': `time.perf_counter ()`
- 'process_time': `time.process_time ()`
- 'thread_time': `time.thread_time ()`
- 'time': `time.time ()`

The result has the following attributes:

- *adjustable*: True si el reloj se puede cambiar automáticamente (por ejemplo, por un demonio NTP) o manualmente por el administrador del sistema, False de lo contrario

- *implementation*: el nombre de la función C subyacente utilizada para obtener el valor del reloj. Consulte *Constantes de ID de reloj* para conocer los posibles valores.
- *monotonic*: “Verdadero” si el reloj no puede retroceder, *False* de lo contrario
- *resolution*: la resolución del reloj en segundos (*float*)

Nuevo en la versión 3.3.

`time.gmtime([secs])`

Convierta un tiempo expresado en segundos desde la época en a *struct_time* en UTC en el que el indicador *dst* siempre es cero. Si no se proporciona *secs* o *None*, se utiliza la hora actual retornada por *time()*. Se ignoran fracciones de segundo. Consulte más arriba para obtener una descripción del objeto *struct_time*. Ver *calendar.timegm()* para el inverso de esta función.

`time.localtime([secs])`

Como *gmtime()* pero se convierte a la hora local. Si no se proporciona *secs* o *None*, se utiliza la hora actual retornada por *time()*. El indicador *dst* se establece en 1 cuando DST se aplica al tiempo dado.

localtime() may raise *OverflowError*, if the timestamp is outside the range of values supported by the platform C *localtime()* or *gmtime()* functions, and *OSError* on *localtime()* or *gmtime()* failure. It's common for this to be restricted to years between 1970 and 2038.

`time.mktime(t)`

Esta es la función inversa de *localtime()*. Su argumento es *struct_time* o una 9-tupla completa (ya que se necesita la bandera *dst*; use -1 como la bandera *dst* si es desconocida) que expresa la hora en hora *local*, no UTC. Retorna un número de coma flotante, por compatibilidad con *time()*. Si el valor de entrada no se puede representar como un tiempo válido, se generará *OverflowError* o *ValueError* (que depende de si Python o las bibliotecas C subyacentes capturan el valor no válido). La fecha más temprana para la que puede generar una hora depende de la plataforma.

`time.monotonic()` → float

Return the value (in fractional seconds) of a monotonic clock, i.e. a clock that cannot go backwards. The clock is not affected by system clock updates. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

Nuevo en la versión 3.3.

Distinto en la versión 3.5: La función ahora está siempre disponible y siempre en todo el sistema.

`time.monotonic_ns()` → int

Similar a *monotonic()*, pero el tiempo de retorno es en nanosegundos.

Nuevo en la versión 3.7.

`time.perf_counter()` → float

Return the value (in fractional seconds) of a performance counter, i.e. a clock with the highest available resolution to measure a short duration. It does include time elapsed during sleep and is system-wide. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

Nuevo en la versión 3.3.

`time.perf_counter_ns()` → int

Similar a *perf_counter()*, pero el tiempo de retorno es en nanosegundos.

Nuevo en la versión 3.7.

`time.process_time()` → float

Return the value (in fractional seconds) of the sum of the system and user CPU time of the current process. It does not include time elapsed during sleep. It is process-wide by definition. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

Nuevo en la versión 3.3.

`time.process_time_ns()` → int

Similar a *process_time()* pero retorna el tiempo en nanosegundos.

Nuevo en la versión 3.7.

`time.sleep(secs)`

Suspende la ejecución del hilo que lo invoca por el número de segundos dado. El argumento puede ser un número de punto flotante para indicar un tiempo de suspensión más preciso. El tiempo de suspensión real puede ser menor que el solicitado porque cualquier señal detectada terminará la función `sleep()` siguiendo la rutina de captura de la señal. El tiempo de suspensión también puede ser más largo que el solicitado por una cantidad arbitraria debido a la programación de otra actividad en el sistema.

Distinto en la versión 3.5: La función ahora duerme al menos * segundos * incluso si el sueño es interrumpido por una señal, excepto si el manejador de la señal genera una excepción (ver [PEP 475](#) para la justificación).

`time.strftime(format[, t])`

Convierta una tupla o `struct_time` que represente un tiempo retornado por `gmtime()` o `localtime()` en una cadena como se especifica mediante el argumento `format`. Si no se proporciona `t`, se utiliza la hora actual retornada por `localtime()`. `format` debe ser una cadena. `ValueError` se genera si algún campo en `t` está fuera del rango permitido.

0 es un argumento legal para cualquier posición en la tupla de tiempo; si normalmente es ilegal, el valor se fuerza a uno correcto.

Las siguientes directivas se pueden incrustar en la cadena `format`. Se muestran sin el ancho de campo opcional y la especificación de precisión, y se reemplazan por los caracteres indicados en el resultado `strftime()`:

Di-rec-ti-va	Sentido	No-tas
%a	Nombre abreviado del día local de la localidad.	
%A	Nombre completo del día laborable de Localidad.	
%b	Nombre abreviado del mes de la localidad.	
%B	Nombre completo del mes de la localidad.	
%c	Representación apropiada de fecha y hora de la localidad.	
%d	Día del mes como número decimal [01,31].	
%H	Hora (reloj de 24 horas) como un número decimal [00,23].	
%I	Hora (reloj de 12 horas) como un número decimal [01,12].	
%j	Día del año como número decimal [001,366].	
%m	Mes como un número decimal [01,12].	
%M	Minuto como un número decimal [00,59].	
%p	El equivalente de la configuración regional de AM o PM.	(1)
%S	Segunda como un número decimal [00,61].	(2)
%U	Número de semana del año (domingo como primer día de la semana) como número decimal [00,53]. Todos los días en un nuevo año anterior al primer domingo se consideran en la semana 0.	(3)
%w	Día de la semana como un número decimal [0 (domingo), 6].	
%W	Número de semana del año (lunes como primer día de la semana) como número decimal [00,53]. Todos los días en un nuevo año anterior al primer lunes se consideran en la semana 0.	(3)
%x	Representación de fecha apropiada de la localidad.	
%X	Representación del tiempo apropiado de la localidad.	
%y	Año sin siglo como número decimal [00,99].	
%Y	Año con siglo como número decimal.	
%z	Desplazamiento de zona horaria que indica una diferencia horaria positiva o negativa con respecto a UTC / GMT de la forma + HHMM o -HHMM, donde H representa dígitos decimales de hora y M representa dígitos decimales de minuto [-23: 59, +23: 59].	
%Z	Nombre de zona horaria (sin caracteres si no existe zona horaria).	
%%	Un carácter literal '% '.	

Notas:

(1) Cuando se usa con la función `strptime()`, la directiva `%p` solo afecta el campo de hora de salida si

se usa la directiva `%I` para analizar la hora.

- (2) El rango realmente es “0” a “61”; el valor “60” es válido en las marcas de tiempo que representan [segundos intercalares](#) y el valor 61 es compatible por razones históricas.
- (3) Cuando se usa con la función `strptime()`, `%U` y `%W` solo se usan en los cálculos cuando se especifica el día de la semana y el año.

Aquí hay un ejemplo, un formato para fechas compatibles con el especificado en estándar de correo electrónico de Internet [RFC 2822](#).¹

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

Es posible que se admitan directivas adicionales en ciertas plataformas, pero solo las que se enumeran aquí tienen un significado estandarizado por ANSI C. Para ver el conjunto completo de códigos de formato admitidos en su plataforma, consulte la documentación de: [manpage: strftime \(3\)](#).

En algunas plataformas, una especificación de precisión y ancho de campo opcional puede seguir inmediatamente el `'%'` inicial de una directiva en el siguiente orden; Esto tampoco es portátil. El ancho del campo es normalmente 2 excepto `%j` donde es 3.

`time.strptime(string[, format])`

Analiza una cadena que representa un tiempo de acuerdo con un formato. El valor de retorno es a `struct_time` como lo retorna `gmtime()` o `localtime()`.

El parámetro `format` utiliza las mismas directivas que las utilizadas por `strftime()`; el valor predeterminado es `"%a %b %d %H:%M:%S %Y"` que coincide con el formato retornado por `ctime()`. Si `string` no se puede analizar de acuerdo con `format`, o si tiene un exceso de datos después del análisis, se excita `ValueError`. Los valores predeterminados utilizados para completar los datos faltantes cuando no se pueden inferir valores más precisos son (1900, 1, 1, 0, 0, 0, 0, 0, 1, -1). Tanto `string` como `format` deben ser strings.

Por ejemplo:

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
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)
```

La compatibilidad con la directiva `%Z` se basa en los valores contenidos en “`tzname`” y en si “`daylight`” es verdadero. Debido a esto, es específico de la plataforma, excepto para reconocer UTC y GMT que siempre se conocen (y se consideran zonas horarias que no son de horario de verano).

Solo se admiten las directivas especificadas en la documentación. Debido a que `strftime()` se implementa por plataforma, a veces puede ofrecer más directivas que las enumeradas. Pero `strptime()` es independiente de cualquier plataforma y, por lo tanto, no necesariamente admite todas las directivas disponibles que no están documentadas como compatibles.

class `time.struct_time`

El tipo de secuencia de valores de tiempo retornado por `gmtime()`, `localtime()`, y `strptime()`. Es un objeto con una interfaz *named tuple*: se puede acceder a los valores por índice y por nombre de atributo. Los siguientes valores están presentes:

¹ El uso de `%Z` ahora está en desuso, pero el escape `%z` que se expande al desplazamiento de hora / minuto preferido no es compatible con todas las bibliotecas ANSI C. Además, una lectura estricta del estándar original de 1982 [RFC 822](#) requiere un año de dos dígitos (`%y` en lugar de `%Y`), pero la práctica se trasladó a años de 4 dígitos mucho antes del año 2000. Después de eso, [RFC 822](#) se volvió obsoleto y el año de 4 dígitos fue recomendado por primera vez por [RFC 1123](#) y luego ordenado por [RFC 2822](#).

Índice	Atributo	Valores
0	tm_year	(por ejemplo, 1993)
1	tm_mon	rango [1, 12]
2	tm_mday	rango [1, 31]
3	tm_hour	rango [0, 23]
4	tm_min	rango [0, 59]
5	tm_sec	rango [0, 61]; ver (2) in <code>strptime()</code> descripción
6	tm_wday	rango [0, 6], Lunes es 0
7	tm_yday	rango [1, 366]
8	tm_isdst	0, 1 ó -1; ver abajo
N/A	tm_zone	abreviatura del nombre de la zona horaria
N/A	tm_gmtoff	desplazamiento al este de UTC en segundos

Tenga en cuenta que, a diferencia de la estructura C, el valor del mes es un rango de [1, 12], no [0, 11].

En llamadas a `mktime()`, `tm_isdst` puede establecerse en 1 cuando el horario de verano está vigente y 0 cuando no lo está. Un valor de -1 indica que esto no se conoce y, por lo general, se completará el estado correcto.

Cuando una tupla con una longitud incorrecta se pasa a una función que espera a `struct_time`, o que tiene elementos del tipo incorrecto, se genera a `TypeError`.

`time.time()` → float

Retorna el tiempo en segundos desde *epoch* como un número de coma flotante. La fecha específica de la época y el manejo de los *leap seconds* depende de la plataforma. En Windows y la mayoría de los sistemas Unix, la época es el 1 de enero de 1970, 00:00:00 (UTC) y los segundos bisiestos no se cuentan para el tiempo en segundos desde la época. Esto se conoce comúnmente como **Tiempo Unix**. Para saber cuál es la época en una plataforma determinada, mire `gmtime(0)`.

Tenga en cuenta que aunque el tiempo siempre se retorna como un número de coma flotante, no todos los sistemas proporcionan tiempo con una precisión superior a 1 segundo. Si bien esta función normalmente retorna valores no decrecientes, puede retornar un valor más bajo que una llamada anterior si el reloj del sistema se ha retrasado entre las dos llamadas.

El número retornado por `time()` se puede convertir a un formato de hora más común (es decir, año, mes, día, hora, etc.) en UTC pasándolo a función `gmtime()` o en hora local pasándola a la función `localtime()`. En ambos casos se retorna un objeto `struct_time`, desde el cual se puede acceder a los componentes de la fecha del calendario como atributos.

`time.thread_time()` → float

Return the value (in fractional seconds) of the sum of the system and user CPU time of the current thread. It does not include time elapsed during sleep. It is thread-specific by definition. The reference point of the returned value is undefined, so that only the difference between the results of two calls in the same thread is valid.

Disponibilidad: Windows, Linux, sistemas Unix que admiten “`CLOCK_THREAD_CPUTIME_ID`”.

Nuevo en la versión 3.7.

`time.thread_time_ns()` → int

Similar a `thread_time()` pero retorna el tiempo en nanosegundos.

Nuevo en la versión 3.7.

`time.time_ns()` → int

Similar a `time()` pero retorna el tiempo como un número entero de nanosegundos desde la *época*.

Nuevo en la versión 3.7.

`time.tzset()`

Restablezca las reglas de conversión de tiempo utilizadas por las rutinas de la biblioteca. La variable de entorno TZ especifica cómo se hace esto. También establecerá las variables `tzname` (de variable de entorno TZ), `timezone` (segundos que no son DST al oeste de UTC), `altzone` (segundos DST al oeste de UTC) y

`daylight` (a 0 si esta zona horaria no tiene ninguna regla de horario de verano, o a cero si hay un horario pasado, presente o futuro cuando se aplica el horario de verano).

Disponibilidad: Unix.

Nota: Aunque en muchos casos, cambiar la variable de entorno TZ puede afectar la salida de funciones como `localtime()` sin llamar a `tzset()`, no se debe confiar en este comportamiento.

La variable de entorno TZ no debe contener espacios en blanco.

El formato estándar de la variable de entorno TZ es (espacio en blanco agregado para mayor claridad):

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

Donde están los componentes:

std` y ``dst Tres o más caracteres alfanuméricos que dan las abreviaturas de zona horaria. Estos se propagarán en `time.tzname`

offset El desplazamiento tiene la forma: $\pm hh[:mm[:ss]]$. Esto indica el valor agregado de la hora local para llegar a UTC. Si está precedido por un “-”, la zona horaria está al este del primer meridiano; de lo contrario, es oeste. Si no hay desplazamiento después de `dst`, se supone que el horario de verano es una hora antes del horario estándar.

start[/time], end[/time] Indica cuándo cambiar hacia y desde DST. El formato de las fechas de inicio y finalización es uno de los siguientes:

Jn El día de Julio * n * ($1 \leq n \leq 365$). Los días bisiestos no se cuentan, por lo que en todos los años el 28 de febrero es el día 59 y el 1 de marzo es el día 60.

n El día julio basado en cero ($0 \leq n \leq 365$). Los días bisiestos se cuentan y es posible referirse al 29 de febrero.

Mm.n.d El día d ($0 \leq d \leq 6$) de la semana n del mes m del año ($1 \leq n \leq 5$, $1 \leq m \leq 12$, donde la semana 5 significa «el último d día del mes m», que puede ocurrir en la cuarta o quinta semana). La semana 1 es la primera semana en la que ocurre el día d. El día cero es un domingo.

`time` tiene el mismo formato que `offset`, excepto que no se permite ningún signo inicial (“-” o “+”). El valor predeterminado, si no se da el tiempo, es 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'
```

En muchos sistemas Unix (incluidos *BSD, Linux, Solaris y Darwin), es más conveniente utilizar la base de datos `zoneinfo` (`tzfile(5)`) del sistema para especificar las reglas de zona horaria. Para hacer esto, configure la variable de entorno TZ en la ruta del archivo de datos de zona horaria requerida, en relación con la raíz de la base de datos de zona horaria “zoneinfo” de los sistemas, generalmente ubicada en `/usr/share/zoneinfo`. Por ejemplo, 'US/Eastern', 'Australia/Melbourne', 'Egypt' o 'Europe/Amsterdam'.

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

16.3.2 Constantes de ID de reloj

Estas constantes se utilizan como parámetros para `clock_getres()` y `clock_gettime()`.

`time.CLOCK_BOOTTIME`

Idéntico a `CLOCK_MONOTONIC`, excepto que también incluye cualquier momento en que el sistema esté suspendido.

Esto permite que las aplicaciones obtengan un reloj monotónico con suspensión sin tener que lidiar con las complicaciones de `CLOCK_REALTIME`, que puede tener discontinuidades si se cambia la hora usando `settimeofday()` o similar.

Disponibilidad: Linux 2.6.39 o posterior.

Nuevo en la versión 3.7.

`time.CLOCK_HIGHRES`

El sistema operativo Solaris tiene un temporizador “`CLOCK_HIGHRES`” que intenta utilizar una fuente de hardware óptima y puede brindar una resolución cercana a los nanosegundos. “`CLOCK_HIGHRES`” es el reloj de alta resolución no ajustable.

Disponibilidad: Solaris.

Nuevo en la versión 3.3.

`time.CLOCK_MONOTONIC`

Reloj que no se puede configurar y representa el tiempo monótono desde algún punto de partida no especificado.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

`time.CLOCK_MONOTONIC_RAW`

Similar a `CLOCK_MONOTONIC`, pero proporciona acceso a un tiempo sin procesar basado en hardware que no está sujeto a ajustes NTP.

Disponibilidad: Linux 2.6.28 y posterior, macOS 10.12 y posterior.

Nuevo en la versión 3.3.

`time.CLOCK_PROCESS_CPUTIME_ID`

Temporizador por proceso de alta resolución desde la CPU.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

`time.CLOCK_PROF`

Temporizador por proceso de alta resolución desde la CPU.

Disponibilidad: FreeBSD, NetBSD 7 o posterior, OpenBSD.

Nuevo en la versión 3.7.

`time.CLOCK_THREAD_CPUTIME_ID`

Reloj de tiempo de CPU específico de subproceso.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

`time.CLOCK_UPTIME`

Tiempo cuyo valor absoluto es el tiempo que el sistema ha estado funcionando y no suspendido, proporcionando una medición precisa del tiempo de actividad, tanto absoluta como de intervalo.

Disponibilidad: FreeBSD, OpenBSD 5.5 o posterior.

Nuevo en la versión 3.7.

`time.CLOCK_UPTIME_RAW`

Reloj que se incrementa monotónicamente, rastreando el tiempo desde un punto arbitrario, no afectado por los ajustes de frecuencia o tiempo y no incrementado mientras el sistema está dormido.

Disponibilidad: macOS 10.12 y posterior.

Nuevo en la versión 3.8.

La siguiente constante es el único parámetro que se puede enviar a `clock_settime()`.

`time.CLOCK_REALTIME`

Reloj en tiempo real de todo el sistema. Configurar este reloj requiere los privilegios apropiados.

Disponibilidad: Unix.

Nuevo en la versión 3.3.

16.3.3 Constantes de zona horaria

`time.altzone`

El desplazamiento de la zona horaria de horario de verano local, en segundos al oeste de UTC, si se define uno. Esto es negativo si la zona horaria local del horario de verano está al este de UTC (como en Europa occidental, incluido el Reino Unido). Solo use esto si la `daylight` no es cero. Vea la nota abajo.

`time.daylight`

No es cero si se define una zona horaria DST. Vea la nota abajo.

`time.timezone`

El desplazamiento de la zona horaria local (no DST), en segundos al oeste de UTC (negativo en la mayoría de Europa occidental, positivo en los EE. UU., Cero en el Reino Unido). Vea la nota abajo.

`time.tzname`

Una tupla de dos cadenas: la primera es el nombre de la zona horaria local no DST, la segunda es el nombre de la zona horaria local DST. Si no se define la zona horaria DST, la segunda cadena no debe usarse. Vea la nota abajo.

Nota: Para las constantes de zona horaria anteriores (`altzone`, `daylight`, `timezone`, y `tzname`), el valor está determinado por las reglas de zona horaria vigentes en el momento de carga del módulo o la última vez se llama a `tzset()` y puede ser incorrecto en el pasado. Se recomienda utilizar `tm_gmtoff` y `tm_zone` resulta de `localtime()` para obtener información sobre la zona horaria.

Ver también:

Modulo `datetime` Más interfaz orientada a objetos para fechas y horas.

Modulo `locale` Servicios de internacionalización. La configuración regional afecta la interpretación de muchos especificadores de formato en `strftime()` y `strptime()`.

Modulo `calendar` Funciones generales relacionadas con el calendario. `timegm()` es el inverso de `gmtime()` de este módulo.

Notas al pie

16.4 argparse — Analizador sintáctico (*Parser*) para las opciones, argumentos y sub-comandos de la línea de comandos

Nuevo en la versión 3.2.

Código fuente: [Lib/argparse.py](#)

Tutorial

Esta página contiene la información de referencia de la API. Para una introducción más amigable al análisis de la línea de comandos de Python, echa un vistazo al `argparse` tutorial.

El módulo `argparse` facilita la escritura de interfaces de línea de comandos amigables. El programa define qué argumentos requiere, y `argparse` averiguará cómo analizar los de `sys.argv`. El módulo `argparse` también genera automáticamente mensajes de ayuda y de uso y muestra errores cuando los usuarios dan parámetros incorrectos al programa.

16.4.1 Ejemplo

El siguiente código es un programa Python que toma una lista de números enteros y obtiene la suma o el máximo:

```
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))
```

Asumiendo que el código Python anterior se guarda en un archivo llamado `prog.py`, se puede ejecutar en la línea de comandos y proporciona mensajes de ayuda útiles:

```
$ 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)
```

Cuando se ejecuta con los parámetros apropiados, muestra la suma o el máximo de los números enteros de la línea de comandos:

```
$ python prog.py 1 2 3 4
4
```

(continué en la próxima página)

(proviene de la página anterior)

```
$ python prog.py 1 2 3 4 --sum
10
```

Si se pasan argumentos incorrectos, se mostrará un error:

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

Las siguientes secciones te guiarán a través de este ejemplo.

Creando un analizador sintáctico (*parser*)

El primer paso para usar *argparse* es crear un objeto *ArgumentParser*

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

El objeto *ArgumentParser* contendrá toda la información necesaria para analizar la línea de comandos con los tipos de datos de Python.

Añadiendo argumentos

Completar un *ArgumentParser* con información sobre los argumentos del programa se hace realizando llamadas al método *add_argument()*. Generalmente, estas llamadas le dicen a *ArgumentParser* cómo capturar las cadenas de caracteres de la línea de comandos y convertirlas en objetos. Esta información se almacena y se usa cuando se llama a *parse_args()*. Por ejemplo:

```
>>> 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)')
```

Más tarde, llamando a *parse_args()* retornará un objeto con dos atributos, *integers* y *accumulate*. El atributo *integers* será una lista de uno o más enteros, y el atributo *accumulate* será la función *sum()*, si se especificó *--sum* en la línea de comandos, o la función *max()* si no.

Analizando argumentos

ArgumentParser analiza los argumentos mediante el método *parse_args()*. Éste inspeccionará la línea de comandos, convertirá cada argumento al tipo apropiado y luego invocará la acción correspondiente. En la mayoría de los casos, esto significa que un simple objeto *Namespace* se construirá a partir de los atributos analizados en la línea de comandos:

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

En un *script*, *parse_args()* será llamado típicamente sin argumentos, y la *ArgumentParser* determinará automáticamente los argumentos de la línea de comandos de *sys.argv*.

16.4.2 Objetos *ArgumentParser*

```
class argparse.ArgumentParser (prog=None, usage=None, description=None, epilog=None,
                                parents=[], formatter_class=argparse.HelpFormatter,
                                prefix_chars='-', fromfile_prefix_chars=None, argu-
                                ment_default=None, conflict_handler='error', add_help=True,
                                allow_abbrev=True)
```

Crea un nuevo objeto *ArgumentParser*. Todos los parámetros deben pasarse como argumentos de palabra clave. Cada parámetro tiene su propia descripción más detallada a continuación, pero en resumen son:

- *prog* - El nombre del programa (default: `sys.argv[0]`)
- *usage* - La cadena de caracteres que describe el uso del programa (por defecto: generado a partir de los argumentos añadidos al analizador)
- *description* - Texto a mostrar antes del argumento ayuda (por defecto: ninguno)
- *epilog* - Texto a mostrar después del argumento ayuda (por defecto: ninguno)
- *parents* - Una lista de objetos *ArgumentParser* cuyos argumentos también deberían ser incluidos
- *formatter_class* - Una clase para personalizar la salida de la ayuda
- *prefix_chars* - El conjunto de caracteres que preceden a los argumentos opcionales (por defecto: '-')
- *fromfile_prefix_chars* - El conjunto de caracteres que preceden a los archivos de los cuales se deberían leer los argumentos adicionales (por defecto: None)
- *argument_default* - El valor global por defecto de los argumentos (por defecto: None)
- *conflict_handler* - La estrategia para resolver los opcionales conflictivos (normalmente es innecesaria)
- *add_help* - Añade una opción `-h/--help` al analizador (por defecto: True)
- *allow_abbrev* - Permite abreviar las opciones largas si la abreviatura es inequívoca. (por defecto: True)

Distinto en la versión 3.5: se añadió el parámetro *allow_abbrev*.

Distinto en la versión 3.8: En versiones anteriores, *allow_abbrev* también deshabilitaba la agrupación de banderas (*flags*) cortas como `-vv` para que sea `-v -v`.

En las siguientes secciones se describe cómo se utiliza cada una de ellas.

prog

Por defecto, los objetos *ArgumentParser* utilizan `sys.argv[0]` para determinar cómo mostrar el nombre del programa en los mensajes de ayuda. Este valor por defecto es casi siempre deseable porque hará que los mensajes de ayuda coincidan con la forma en que el programa fue invocado en la línea de comandos. Por ejemplo, considera un archivo llamado `myprogram.py` con el siguiente código:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

La ayuda para este programa mostrará `myprogram.py` como el nombre del programa (sin importar desde dónde se haya invocado el programa):

```
$ 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   foo help
```

Para cambiar este comportamiento por defecto, se puede proporcionar otro valor usando el argumento “prog=” para *ArgumentParser*:

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
  -h, --help  show this help message and exit
```

Ten en cuenta que el nombre del programa, ya sea determinado a partir de `sys.argv[0]` o del argumento `prog=`, está disponible para los mensajes de ayuda usando el especificador de formato `%(prog)s`.

```
>>> 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
```

uso

Por defecto, *ArgumentParser* determina el mensaje de uso a partir de los argumentos que contiene:

```
>>> 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
```

El mensaje por defecto puede ser sustituido con el argumento de palabra clave `usage=`:

```
>>> 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
```

El especificador de formato % (prog) s está preparado para introducir el nombre del programa en los mensajes de ayuda.

description

La mayoría de las llamadas al constructor `ArgumentParser` usarán el argumento de palabra clave `description=`. Este argumento da una breve descripción de lo que hace el programa y cómo funciona. En los mensajes de ayuda, la descripción se muestra entre la cadena de caracteres de uso (*usage*) de la línea de comandos y los mensajes de ayuda para los distintos argumentos:

```
>>> 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
```

Por defecto, la descripción será ajustada a una línea para que encaje en el espacio dado. Para cambiar este comportamiento, revisa el argumento *formatter_class*.

epilog

A algunos programas les gusta mostrar una descripción adicional del programa después de la descripción de los argumentos. Dicho texto puede ser especificado usando el argumento `epilog=` para `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
```

Al igual que con el argumento *description*, el texto `epilog=` está por defecto ajustado a una línea, pero este comportamiento puede ser modificado con el argumento *formatter_class* para `ArgumentParser`.

parents

A veces, varios analizadores comparten un conjunto de argumentos comunes. En lugar de repetir las definiciones de estos argumentos, se puede usar un único analizador con todos los argumentos compartidos y pasarlo en el argumento `parents=` a `ArgumentParser`. El argumento `parents=` toma una lista de objetos `ArgumentParser`, recoge todas las acciones de posición y de opción de éstos, y añade estas acciones al objeto `ArgumentParser` que se está construyendo:

```
>>> 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

Ten en cuenta que la mayoría de los analizadores padre especificarán `add_help=False`. De lo contrario, el `ArgumentParser` verá dos opciones `-h/--help` (una para el padre y otra para el hijo) y generará un error.

Nota: Debes inicializar completamente los analizadores antes de pasarlos a través de `parents=`. Si cambias los analizadores padre después del analizador hijo, esos cambios no se reflejarán en el hijo.

formatter_class

los objetos `ArgumentParser` permiten personalizar el formato de la ayuda especificando una clase de formato alternativa. Actualmente, hay cuatro clases de este tipo:

```
class argparse.RawDescriptionHelpFormatter
class argparse.RawTextHelpFormatter
class argparse.ArgumentDefaultsHelpFormatter
class argparse.MetavarTypeHelpFormatter
```

`RawDescriptionHelpFormatter` y `RawTextHelpFormatter` dan más control sobre cómo se muestran las descripciones de texto. Por defecto, los objetos `ArgumentParser` ajustan a la línea los textos de *description* y *epilog* en los mensajes de ayuda de la línea de comandos:

```
>>> 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
```

Pasar `RawDescriptionHelpFormatter` como `formatter_class=` indica que *description* y *epilog* ya tienen el formato correcto y no deben ser ajustados a la línea:

```
>>> 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
...     '''))
```

(continué en la próxima página)

(proviene de la página anterior)

```
...         '''))
>>> 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 mantiene espacios en blanco para todo tipo de texto de ayuda, incluyendo descripciones de argumentos. Sin embargo, varias líneas nuevas son reemplazadas por una sola. Si deseas conservar varias líneas en blanco, añade espacios entre las nuevas líneas.

ArgumentDefaultsHelpFormatter añade automáticamente información sobre los valores por defecto a cada uno de los mensajes de ayuda de los argumentos:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> 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)
```

MetavarTypeHelpFormatter utiliza el nombre del parámetro *type* para cada argumento como el nombre a mostrar para sus valores (en lugar de utilizar *dest* como lo hace el formato habitual):

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

optional arguments:
  -h, --help  show this help message and exit
  --foo int
```

prefix_chars

La mayoría de las opciones de la línea de comandos usarán `-` como prefijo, por ejemplo `-f/--foo`. Los analizadores que necesiten soportar caracteres prefijo diferentes o adicionales, por ejemplo, para opciones como `+f` o `/foo`, pueden especificarlos usando el argumento `prefix_chars=` para el constructor *ArgumentParser*:

```
>>> 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')
```

El argumento `prefix_chars=` tiene un valor por defecto de `'-'`. Proporcionar un conjunto de caracteres que no incluya `-` causará que las opciones `-f/--foo` no sean inhabilitadas.

fromfile_prefix_chars

A veces, por ejemplo, cuando se trata de una lista de argumentos particularmente larga, puede tener sentido mantener la lista de argumentos en un archivo en lugar de escribirla en la línea de comandos. Si el argumento `fromfile_prefix_chars=` se da al constructor *ArgumentParser*, entonces los argumentos que empiezan con cualquiera de los caracteres especificados se tratarán como archivos, y serán reemplazados por los argumentos que contienen. Por ejemplo:

```
>>> 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')
```

Los argumentos leídos de un archivo deben ser por defecto uno por línea (pero vea también *convert_arg_line_to_args()*) y se tratan como si estuvieran en el mismo lugar que el argumento de referencia del archivo original en la línea de comandos. Así, en el ejemplo anterior, la expresión `['-f', 'foo', '@args.txt']` se considera equivalente a la expresión `['-f', 'foo', '-f', 'bar']`.

El argumento `fromfile_prefix_chars=` por defecto es `None`, lo que significa que los argumentos nunca serán tratados como referencias de archivos.

argument_default

Generalmente, los valores por defecto de los argumentos se especifican ya sea pasando un valor por defecto a *add_argument()* o llamando a los métodos *set_defaults()* con un conjunto específico de pares nombre-valor. A veces, sin embargo, puede ser útil especificar un único valor por defecto para todos los argumentos del analizador. Esto se puede lograr pasando el argumento de palabra clave `argument_default=aArgumentParser`. Por ejemplo, para suprimir globalmente la creación de atributos en las llamadas a *parse_args()*, proporcionamos el argumento `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()
```

allow_abbrev

Normalmente, cuando pasas una lista de argumentos al método `parse_args()` de un `ArgumentParser`, *reconoce las abreviaturas* de las opciones largas.

Esta característica puede ser desactivada poniendo `allow_abbrev` a `False`:

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

Nuevo en la versión 3.5.

conflict_handler

Los objetos `ArgumentParser` no permiten dos acciones con la misma cadena de caracteres de opción. Por defecto, los objetos `ArgumentParser` lanzan una excepción si se intenta crear un argumento con una cadena de caracteres de opción que ya está en uso:

```
>>> 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
```

A veces (por ejemplo, cuando se utiliza *parents*) puede ser útil anular simplemente cualquier argumento antiguo con la misma cadena de caracteres de opción. Para lograr este comportamiento, se puede suministrar el valor `'resolve'` al argumento `conflict_handler=` de `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
```

Ten en cuenta que los objetos `ArgumentParser` sólo eliminan una acción si todas sus cadenas de caracteres de opción están anuladas. Así, en el ejemplo anterior, la antigua acción `-f/--foo` se mantiene como la acción `"-f"`, porque sólo la cadena de caracteres de opción `--foo` fue anulada.

add_help

Por defecto, los objetos `ArgumentParser` añaden una opción que simplemente muestra el mensaje de ayuda del analizador. Por ejemplo, considera un archivo llamado `myprogram.py` que contiene el siguiente código:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

Si `-h` o `--help` se indica en la línea de comandos, se imprimirá la ayuda de `ArgumentParser`:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   foo help
```

Ocasionalmente, puede ser útil desactivar la inclusión de esta opción de ayuda. Esto se puede lograr pasando `False` como argumento de `add_help=` a `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
```

La opción de ayuda es típicamente `-h/--help`. La excepción a esto es si `prefix_chars=` se especifica y no incluye `-`, en cuyo caso `-h` y `--help` no son opciones válidas. En este caso, el primer carácter en `prefix_chars` se utiliza para preceder a las opciones de ayuda:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

optional arguments:
  +h, ++help  show this help message and exit
```

16.4.3 El método `add_argument()`

`ArgumentParser.add_argument` (*name or flags...* [*, action*] [*, nargs*] [*, const*] [*, default*] [*, type*] [*, choices*] [*, required*] [*, help*] [*, metavar*] [*, dest*])

Define cómo se debe interpretar un determinado argumento de línea de comandos. Cada parámetro tiene su propia descripción más detallada a continuación, pero en resumen son:

- *name or flags* - Ya sea un nombre o una lista de cadena de caracteres de opción, e.g. `foo o -f`, `--foo`.
- *action* - El tipo básico de acción a tomar cuando este argumento se encuentra en la línea de comandos.
- *nargs* - El número de argumentos de la línea de comandos que deben ser consumidos.
- *const* - Un valor fijo requerido por algunas selecciones de *action* y *nargs*.
- *default* - El valor dado si el argumento está ausente de la línea de comandos.
- *type* - El tipo al que debe convertirse el argumento de la línea de comandos.
- *choices* - Un contenedor con los valores permitidos para el argumento.
- *required* - Si se puede omitir o no la opción de la línea de comandos (sólo opcionales).
- *help* - Una breve descripción de lo que hace el argumento.
- *metavar* - Un nombre para el argumento en los mensajes de uso.
- *dest* - El nombre del atributo que será añadido al objeto retornado por `parse_args()`.

En las siguientes secciones se describe cómo se utiliza cada una de ellas.

name or flags

El método `add_argument()` debe saber si se espera un argumento opcional, como `-f` o `--foo`, o un argumento posicional, como una lista de nombres de archivos. Por lo tanto, los primeros argumentos que se pasan a `add_argument()` deben ser una serie de indicadores (*flags*), o un simple nombre de argumento (*name*). Por ejemplo, se puede crear un argumento opcional como:

```
>>> parser.add_argument('-f', '--foo')
```

mientras que un argumento posicional podría ser creado como:

```
>>> parser.add_argument('bar')
```

Cuando se llama a `parse_args()`, los argumentos opcionales serán identificados por el prefijo `-`, y el resto de los argumentos serán asumidos como posicionales:

```
>>> 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: the following arguments are required: bar
```

action

Los objetos `ArgumentParser` asocian los argumentos de la línea de comandos con las acciones. Estas acciones pueden hacer casi cualquier cosa con los argumentos de línea de comandos asociados a ellas, aunque la mayoría de las acciones simplemente añaden un atributo al objeto retornado por `parse_args()`. El argumento de palabra clave `action` especifica cómo deben ser manejados los argumentos de la línea de comandos. Las acciones proporcionadas son:

- `'store'` - Esta sólo almacena el valor del argumento. Esta es la acción por defecto. Por ejemplo:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args('--foo 1'.split())
Namespace(foo='1')
```

- `'store_const'` - Esta almacena el valor especificado por el argumento de palabra clave `const`. La acción `'store_const'` se usa más comúnmente con argumentos opcionales que especifican algún tipo de indicador (*flag*). Por ejemplo:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_const', const=42)
>>> parser.parse_args(['--foo'])
Namespace(foo=42)
```

- `'store_true'` y `'store_false'` - Son casos especiales de `'store_const'` usados para almacenar los valores `True` y `False` respectivamente. Además, crean valores por defecto de `False` y `True` respectivamente. Por ejemplo:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> parser.parse_args('--foo --bar'.split())
Namespace(foo=True, bar=False, baz=True)
```

- 'append' - Esta almacena una lista, y añade cada valor del argumento a la lista. Esto es útil para permitir que una opción sea especificada varias veces. Ejemplo de uso:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- 'append_const' - Esta almacena una lista, y añade el valor especificado por el argumento de palabra clave *const* a la lista. (Nótese que el argumento de palabra clave *const* por defecto es None.) La acción 'append_const' es útil típicamente cuando múltiples argumentos necesitan almacenar constantes a la misma lista. Por ejemplo:

```
>>> 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=[<class 'str'>, <class 'int'>])
```

- 'count' - Esta cuenta el número de veces que un argumento de palabra clave aparece. Por ejemplo, esto es útil para incrementar los niveles de detalle:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count', default=0)
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

Observa, *default* (el valor por defecto) será None a menos que explícitamente se establezca como 0.

- 'help' - Esta imprime un mensaje de ayuda completo para todas las opciones del analizador actual y luego termina. Por defecto, se añade una acción de ayuda automáticamente al analizador. Ver *ArgumentParser* para detalles de cómo se genera la salida.
- 'version' - Esta espera un argumento de palabra clave *version=* en la llamada *add_argument()*, e imprime la información de la versión y finaliza cuando es invocada:

```
>>> 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
```

- 'extend' - Esta almacena una lista, y extiende cada valor del argumento a la lista. Ejemplo de uso:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action="extend", nargs="+", type=str)
>>> parser.parse_args(['--foo', 'f1', '--foo', 'f2', 'f3', 'f4'])
Namespace(foo=['f1', 'f2', 'f3', 'f4'])
```

Nuevo en la versión 3.8.

También puedes especificar una acción arbitraria pasando una subclase *Action* u otro objeto que implemente la misma interfaz. La forma recomendada de hacer esto es extender *Action*, sobrescribiendo el método “__call__” y opcionalmente el método “__init__”.

Un ejemplo de una acción personalizada:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super().__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         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')
```

Para más detalles, ver [Action](#).

nargs

Los objetos *ArgumentParser* suelen asociar un único argumento de línea de comandos con una única acción a realizar. El argumento de palabra clave *nargs* asocia un número diferente de argumentos de línea de comandos con una sola acción. Los valores admitidos son:

- N (un entero). N argumentos de la línea de comandos se agruparán en una lista. Por ejemplo:

```
>>> 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'])
```

Ten en cuenta que *nargs=1* produce una lista de un elemento. Esto es diferente del valor por defecto, en el que el elemento se produce por sí mismo.

- '?'. Un argumento se consumirá desde la línea de comandos si es posible, y se generará como un sólo elemento. Si no hay ningún argumento de línea de comandos, se obtendrá el valor de *default*. Ten en cuenta que para los argumentos opcionales, hay un caso adicional - la cadena de caracteres de opción está presente pero no va seguida de un argumento de línea de comandos. En este caso se obtendrá el valor de *const*. Algunos ejemplos para ilustrar esto:

```
>>> 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')
```

Uno de los usos más comunes de *nargs='?'* es permitir archivos de entrada y salida opcionales:

```
>>> 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'])
```

(continué en la próxima página)

(proviene de la página anterior)

```
Namespace(infile=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
>>> parser.parse_args([])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)
```

- `'*'`. Todos los argumentos presentes en la línea de comandos se recogen en una lista. Ten en cuenta que generalmente no tiene mucho sentido tener más de un argumento posicional con `nargs='*'`, pero es posible tener múltiples argumentos opcionales con `nargs='*'`. Por ejemplo:

```
>>> 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'])
```

- `'+'`. Al igual que `'*'`, todos los argumentos de la línea de comandos se recogen en una lista. Además, se generará un mensaje de error si no había al menos un argumento presente en la línea de comandos. Por ejemplo:

```
>>> 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: the following arguments are required: foo
```

- `argparse.REMAINDER`. Todos los argumentos restantes de la línea de comandos se recogen en una lista. Esto es útil habitualmente para utilidades de línea de comandos que envían a otras utilidades de línea de comandos:

```
>>> 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')
```

Si no se proporciona el argumento de palabra clave `nargs`, el número de argumentos consumidos se determina por [action](#). Generalmente esto significa que se consumirá un único argumento de línea de comandos y se obtendrá un único elemento (no una lista).

const

El argumento `const` de `add_argument()` se usa para mantener valores constantes que no se leen desde la línea de comandos pero que son necesarios para las diversas acciones de `ArgumentParser`. Los dos usos más comunes son:

- Cuando `add_argument()` se llama con `action='store_const'` o `action='append_const'`. Estas acciones añaden el valor `const` a uno de los atributos del objeto retornado por `parse_args()`. Mira la descripción [action](#) para ver ejemplos.
- Cuando `add_argument()` se llama con cadenas de caracteres de opción (como `-f` o `-foo`) y `nargs='?'`. Esto crea un argumento opcional que puede ir seguido de cero o un argumento de línea de comandos. Al analizar la línea de comandos, si la cadena de opciones se encuentra sin ningún argumento de línea de comandos que la siga, asumirá en su lugar el valor de `const`. Mira la descripción [nargs](#) para ejemplos.

Con las acciones `'store_const'` y `'append_const'`, se debe asignar el argumento palabra clave `const`. Para otras acciones, por defecto es `None`.

default

Todos los argumentos opcionales y algunos argumentos posicionales pueden ser omitidos en la línea de comandos. El argumento de palabra clave `default` de `add_argument()`, cuyo valor por defecto es `None`, especifica qué valor debe usarse si el argumento de línea de comandos no está presente. Para los argumentos opcionales, se usa el valor `default` cuando la cadena de caracteres de opción no está presente en la línea de comandos:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

Si el valor `default` es una cadena de caracteres, el analizador procesa el valor como si fuera un argumento de la línea de comandos. En particular, el analizador aplica cualquier argumento de conversión *type*, si se proporciona, antes de establecer el atributo en el valor de retorno *Namespace*. En caso contrario, el analizador utiliza el valor tal y como es:

```
>>> 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)
```

Para argumentos posiciones con *nargs* igual a `?` o `*`, el valor `default` se utiliza cuando no hay ningún argumento de línea de comandos presente:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

Proporcionar `default=argparse.SUPPRESS` causa que no se agregue ningún atributo si el argumento de la línea de comandos no está presente:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

type

Por defecto, los objetos *ArgumentParser* leen los argumentos de la línea de comandos como simples cadenas de caracteres. Sin embargo, muy a menudo la cadena de caracteres de la línea de comandos debe ser interpretada como otro tipo, como una *float* o *int*. El argumento de palabra clave `type` de `add_argument()` permite realizar cualquier comprobación y conversión de tipos que sea necesaria. Los tipos y funciones incorporadas (*built-in*) comunes pueden ser usados directamente como el valor del argumento `type`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.add_argument('bar', type=open)
>>> parser.parse_args('2 temp.txt'.split())
Namespace(bar=<_io.TextIOWrapper name='temp.txt' encoding='UTF-8'>, foo=2)
```

Consulta la sección sobre el argumento de palabra clave *default* para obtener información sobre cuándo se aplica el argumento *type* a los argumentos por defecto.

Para facilitar el uso de varios tipos de archivos, el módulo *argparse* proporciona *FileType* de fábrica que toma los argumentos *mode=*, *bufsize=*, *encoding=* y *errors=* de la función *open()*. Por ejemplo, *FileType('w')* puede usarse para crear un archivo modificable:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<_io.TextIOWrapper name='out.txt' encoding='UTF-8'>)
```

type= puede aceptar cualquier invocable que use un sólo argumento de cadena de caracteres y retorne el valor convertido:

```
>>> 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
```

El argumento de palabra clave *choices* puede ser más conveniente para las verificadoras de tipo que simplemente comparan contra un rango de valores:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=int, choices=range(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)
```

Consulta la sección *choices* para más detalles.

choices

Algunos argumentos de la línea de comandos deberían seleccionarse de un conjunto restringido de valores. Estos pueden ser manejados pasando un objeto contenedor como el argumento de palabra clave *choices* a *add_argument()*. Cuando se analiza la línea de comandos, se comprueban los valores de los argumentos y se muestra un mensaje de error si el argumento no era uno de los valores aceptables:

```
>>> 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')
```

Ten en cuenta que la inclusión en el contenedor *choices* se comprueba después de que se haya realizado cualquier conversión de *type*, por lo que el tipo de los objetos del contenedor *choices* debe coincidir con el *type* especificado:

```
>>> 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)
```

Se puede pasar cualquier contenedor como el valor para *choices*, así que los objetos *list*, *set*, y los contenedores personalizados están todos soportados.

required

En general, el módulo *argparse* asume que los indicadores (*flags*) como *-f* y *--bar* señalan argumentos *opcionales*, que siempre pueden ser omitidos en la línea de comandos. Para hacer una opción *requerida*, se puede especificar “True” para el argumento de palabra clave “required=” en *add_argument()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: [-h] --foo FOO
: error: the following arguments are required: --foo
```

Como muestra el ejemplo, si una opción está marcada como *required*, *parse_args()* informará de un error si esa opción no está presente en la línea de comandos.

Nota: Las opciones requeridas están consideradas generalmente mala práctica porque los usuarios esperan que las opciones sean *opcionales*, y por lo tanto deberían ser evitadas cuando sea posible.

help

El valor *help* es una cadena de caracteres que contiene una breve descripción del argumento. Cuando un usuario solicita ayuda (normalmente usando *-h* o *--help* en la línea de comandos), estas descripciones *help* se mostrarán con cada argumento:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...                     help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='+',
...                     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
```

Las cadenas de texto *help* pueden incluir varios descriptores de formato para evitar la repetición de cosas como el nombre del programa o el argumento *default*. Los descriptores disponibles incluyen el nombre del programa, *% (prog) s* y la mayoría de los argumentos de palabra clave de *add_argument()*, por ejemplo *% (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
```

Como la cadena de caracteres de ayuda soporta el formato-%, si quieres que aparezca un % literal en la ayuda, debes escribirlo como %%.

`argparse` soporta el silenciar la ayuda para ciertas opciones, ajustando el valor `help` a `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

Cuando `ArgumentParser` genera mensajes de ayuda, necesita alguna forma de referirse a cada argumento esperado. Por defecto, los objetos `ArgumentParser` utilizan el valor `dest` como «nombre» de cada objeto. Por defecto, para las acciones de argumento posicional, el valor `dest` se utiliza directamente, y para las acciones de argumento opcional, el valor `dest` está en mayúsculas. Así, un único argumento posicional con `dest='bar'` se denominará `bar`. Un único argumento opcional `--foo` que debería seguirse por un único argumento de línea de comandos se denominará `FOO`. Un ejemplo:

```
>>> 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

positional arguments:
  bar

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO
```

Un nombre alternativo se puede especificar con `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:
```

(continué en la próxima página)

(proviene de la página anterior)

```
XXX

optional arguments:
  -h, --help  show this help message and exit
  --foo YYY
```

Ten en cuenta que `metavar` sólo cambia el nombre *mostrado* - el nombre del atributo en el objeto `parse_args()` sigue estando determinado por el valor *dest*.

Diferentes valores de `nargs` pueden causar que *metavar* sea usado múltiples veces. Proporcionar una tupla a `metavar` especifica una visualización diferente para cada uno de los argumentos:

```
>>> 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

La mayoría de las acciones `ArgumentParser` añaden algún valor como atributo del objeto retornado por `parse_args()`. El nombre de este atributo está determinado por el argumento de palabra clave `dest` de `add_argument()`. Para acciones de argumento posicional, se proporciona `dest` normalmente como primer argumento de `add_argument()`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

Para las acciones de argumentos opcionales, el valor de `dest` se infiere normalmente de las cadenas de caracteres de opción. `ArgumentParser` genera el valor de `dest` tomando la primera cadena de caracteres de opción larga y quitando la cadena inicial `--`. Si no se proporcionaron cadenas de caracteres de opción largas, `dest` se derivará de la primera cadena de caracteres de opción corta quitando el carácter `-`. Cualquier carácter `-` interno se convertirá a caracteres `_` para asegurarse de que la cadena de caracteres es un nombre de atributo válido. Los ejemplos siguientes ilustran este comportamiento:

```
>>> 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` permite que se proporcione un nombre de atributo personalizado:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```


Las clases *Action*

Las clases *Action* implementan la API de *Action*, un invocable que retorna un invocable que procesa los argumentos de la línea de comandos. Cualquier objeto que siga esta API puede ser pasado como el parámetro *action* a `add_argument()`.

```
class argparse.Action(option_strings, dest, nargs=None, const=None, default=None, type=None,
                     choices=None, required=False, help=None, metavar=None)
```

Los objetos *Action* son utilizados por un *ArgumentParser* para representar la información necesaria para analizar un sólo argumento de una o más cadenas de caracteres de la línea de comandos. La clase *Action* debe aceptar los dos argumentos de posición más cualquier argumento de palabra clave pasado a *ArgumentParser.add_argument()* excepto para la propia *action*.

Las instancias de *Action* (o el valor de retorno de cualquier invocable al parámetro *action*) deben tener definidos los atributos *"dest"*, *"option_strings"*, *"default"*, *"type"*, *"required"*, *"help"*, etc. La forma más fácil de asegurar que estos atributos estén definidos es llamar a *Action.__init__*.

Las instancias de *Action* deben ser invocables, por lo que las subclasses deben anular el método `__call__`, que debería aceptar cuatro parámetros:

- *parser* - El objeto *ArgumentParser* que contiene esta acción.
- *namespace* - El objeto *Namespace* que será retornado por *parse_args()*. La mayoría de las acciones añaden un atributo a este objeto usando *setattr()*.
- *values* - Los argumentos de la línea de comandos asociados, con cualquier tipo de conversión aplicada. Las conversiones de tipos se especifican con el argumento de palabra clave *type* a *add_argument()*.
- *option_string* - La cadena de caracteres de opción que se usó para invocar esta acción. El argumento *option_string* es opcional, y estará ausente si la acción está asociada a un argumento de posición.

El método `__call__` puede realizar acciones arbitrarias, pero típicamente establece atributos en *namespace* basados en *dest* y *values*.

16.4.4 El método *parse_args()*

```
ArgumentParser.parse_args(args=None, namespace=None)
```

Convierte las cadenas de caracteres de argumentos en objetos y los asigna como atributos del espacio de nombres (*namespace*). Retorna el espacio de nombres (*namespace*) ocupado.

Las llamadas previas a *add_argument()* determinan exactamente qué objetos se crean y cómo se asignan. Mira la documentación de *add_argument()* para más detalles.

- *args* - Lista de cadenas de caracteres para analizar. El valor por defecto se toma de *sys.argv*.
- *namespace* - Un objeto para obtener los atributos. Por defecto es un nuevo objeto vacío *Namespace*.

Sintaxis del valor de la opción

El método *parse_args()* soporta diversas formas de especificar el valor de una opción (si requiere uno). En el caso más simple, la opción y su valor se pasan como dos argumentos separados:

```
>>> 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)
```

En el caso de opciones largas (opciones con nombres más largos que un sólo carácter), la opción y el valor también se pueden pasar como un sólo argumento de línea de comandos, utilizando `=` para separarlos:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

Para las opciones cortas (opciones de un sólo carácter de largo), la opción y su valor pueden ser concatenados:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

Se pueden unir varias opciones cortas, usando un sólo prefijo -, siempre y cuando sólo la última opción (o ninguna de ellas) requiera un valor:

```
>>> 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')
```

Argumentos no válidos

Mientras analiza la línea de comandos, `parse_args()` comprueba una variedad de errores, incluyendo opciones ambiguas, tipos no válidos, opciones no válidas, número incorrecto de argumentos de posición, etc. Cuando encuentra un error de este tipo, termina y muestra el error junto con un mensaje de uso:

```
>>> 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'

>>> # 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
```

Argumentos conteniendo -

El método `parse_args()` busca generar errores cuando el usuario ha cometido claramente una equivocación, pero algunas situaciones son inherentemente ambiguas. Por ejemplo, el argumento de línea de comandos `-1` podría ser un intento de especificar una opción o un intento de proporcionar un argumento de posición. El método `parse_args()` es cauteloso aquí: los argumentos de posición sólo pueden comenzar con - si se ven como números negativos y no hay opciones en el analizador que se puedan ver como números negativos

```
>>> 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')
```

(continué en la próxima página)

(proviene de la página anterior)

```

>>> # 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

```

Si tienes argumentos de posición que deben comenzar con `-` y no parecen números negativos, puedes insertar el pseudo-argumento `--` que indica a `parse_args()` que todo lo que sigue es un argumento de posición:

```

>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)

```

Abreviaturas de los argumentos (coincidencia de prefijos)

El método `parse_args()` *por defecto* permite abreviar las opciones largas a un prefijo, si la abreviatura es inequívoca (el prefijo coincide con una opción única):

```

>>> 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

```

Se incurre en un error por argumentos que podrían derivar en más de una opción. Esta característica puede ser desactivada poniendo `allow_abbrev` a `False`.

Más allá de `sys.argv`

A veces puede ser útil tener un *ArgumentParser* analizando argumentos que no sean los de `sys.argv`. Esto se puede lograr pasando una lista de cadenas de caracteres a `parse_args()`. Esto es útil para probar en el *prompt* interactivo:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(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])
```

El objeto *Namespace*

class `argparse.Namespace`

Clase simple utilizada por defecto por `parse_args()` para crear un objeto que contenga atributos y retornarlo.

Esta clase es deliberadamente simple, sólo una subclase *object* con una representación de cadena de texto legible. Si prefieres tener una vista en forma de diccionario de los atributos, puedes usar el lenguaje estándar de Python, `vars()`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

También puede ser útil tener un *ArgumentParser* que asigne atributos a un objeto ya existente, en lugar de un nuevo objeto *Namespace*. Esto se puede lograr especificando el argumento de palabra clave `namespace=`:

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

16.4.5 Otras utilidades

Sub-comandos

`ArgumentParser.add_subparsers` (*[title][, description][, prog][, parser_class][, action][, option_string][, dest][, required][, help][, metavar]*)

Muchos programas dividen su funcionalidad en varios sub-comandos, por ejemplo, el programa `svn` puede llamar sub-comandos como `svn checkout`, `svn update`, y `svn commit`. Dividir la funcionalidad de esta forma puede ser una idea particularmente buena cuando un programa realiza varias funciones diferentes que requieren diferentes tipos de argumentos en la línea de comandos. *ArgumentParser* soporta la creación de tales sub-comandos con el método `add_subparsers()`. El método `add_subparsers()`

se llama normalmente sin argumentos y retorna un objeto de acción especial. Este objeto tiene un único método, `add_parser()`, que toma un nombre de comando y cualquier argumento de construcción *ArgumentParser*, y retorna un objeto *ArgumentParser* que puede ser modificado de la forma habitual.

Descripción de los parámetros:

- *title* - título para el grupo del analizador secundario en la salida de la ayuda; por defecto «*subcommands*» si se proporciona la descripción, de lo contrario utiliza el título para los argumentos de posición
- *description* - descripción para el grupo del analizador secundario en la salida de la ayuda, por defecto `None`
- *prog* - información de uso que se mostrará con la ayuda de los sub-comandos, por defecto el nombre del programa y cualquier argumento de posición antes del argumento del analizador secundario
- *parser_class* - clase que se usará para crear instancias de análisis secundario, por defecto la clase del analizador actual (por ejemplo, *ArgumentParser*)
- *action* - el tipo básico de acción a tomar cuando este argumento se encuentre en la línea de comandos
- *dest* - nombre del atributo en el que se almacenará el nombre del sub-comando; por defecto `None` y no se almacena ningún valor
- *required* - Si se debe proporcionar o no un sub-comando, por defecto `False` (añadido en 3.7)
- *help* - ayuda para el grupo de análisis secundario en la salida de la ayuda, por defecto `None`
- *metavar* - cadena de caracteres que presenta los sub-comandos disponibles en la ayuda; por defecto es `None` y presenta los sub-comandos de la forma {cmd1, cmd2, ..}

Algún ejemplo de uso:

```
>>> # 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
>>> 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)
```

Ten en cuenta que el objeto retornado por *parse_args()* sólo contendrá atributos para el analizador principal y el analizador secundario que fue seleccionado por la línea de comandos (y no cualquier otro analizador secundario). Así que en el ejemplo anterior, cuando se especifica el comando `a`, sólo están presentes los atributos `foo` y `bar`, y cuando se especifica el comando `b`, sólo están presentes los atributos `foo` y `baz`.

Del mismo modo, cuando se solicita un mensaje de ayuda de un analizador secundario, sólo se imprimirá la ayuda para ese analizador en particular. El mensaje de ayuda no incluirá mensajes del analizador principal o de analizadores relacionados. (Sin embargo, se puede dar un mensaje de ayuda para cada comando del analizador secundario suministrando el argumento `help=` a *add_parser()* como se ha indicado anteriormente).

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...
```

(continué en la próxima página)

(proviene de la página anterior)

```
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
```

El método `add_subparsers()` también soporta los argumentos de palabra clave “title” y `description`. Cuando cualquiera de los dos esté presente, los comandos del analizador secundario aparecerán en su propio grupo en la salida de la ayuda. Por ejemplo:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                   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
```

Además, `add_parser` soporta un argumento adicional `aliases`, que permite que múltiples cadenas se refieran al mismo analizador secundario. Este ejemplo, algo del estilo “svn”, alias `co` como abreviatura para `checkout`:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

Una forma particularmente efectiva de manejar los sub-comandos es combinar el uso del método `add_subparsers()` con llamadas a `set_defaults()` para que cada analizador secundario sepa qué función de Python debe ejecutar. Por ejemplo:

```

>>> # 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))

```

De esta manera, puedes dejar que `parse_args()` haga el trabajo de llamar a la función apropiada después de que el análisis de los argumentos se haya completado. Asociar funciones con acciones como esta es típicamente la forma más fácil de manejar las diferentes acciones para cada uno de tus analizadores secundarios. Sin embargo, si es necesario comprobar el nombre del analizador secundario que se ha invocado, el argumento de palabra clave `dest` a la llamada `add_subparsers()` hará el trabajo:

```

>>> 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')

```

Distinto en la versión 3.7: Nuevo argumento de palabra clave *required*.

Objetos `FileType`

class `argparse.FileType` (*mode='r', bufsize=-1, encoding=None, errors=None*)

El generador `FileType` crea objetos que pueden ser transferidos al argumento tipo de `ArgumentParser.add_argument()`. Los argumentos que tienen objetos `FileType` como su tipo abrirán los argumentos de líneas de comandos como archivos con los modos, tamaños de búfer, codificaciones y manejo de errores solicitados (véase la función `open()` para más detalles):

```

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))

```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>,
raw=<_io.FileIO name='raw.dat' mode='wb'>)
```

Los objetos *FileType* entienden el pseudo-argumento '-' y lo convierten automáticamente en `sys.stdin` para objetos de lectura *FileType* y `sys.stdout` para objetos de escritura *FileType*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

Nuevo en la versión 3.4: Los argumentos de palabra clave *encodings* y *errors*.

Grupos de argumentos

`ArgumentParser.add_argument_group(title=None, description=None)`

Por defecto, *ArgumentParser* agrupa los argumentos de la línea de comandos en «argumentos de posición» y «argumentos opcionales» al mostrar los mensajes de ayuda. Cuando hay una mejor agrupación conceptual de argumentos que esta predeterminada, se pueden crear grupos apropiados usando el método `add_argument_group()`:

```
>>> 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
```

El método `add_argument_group()` retorna un objeto de grupo de argumentos que tiene un método `add_argument()` igual que un *ArgumentParser* convencional. Cuando se añade un argumento al grupo, el analizador lo trata como un argumento cualquiera, pero presenta el argumento en un grupo aparte para los mensajes de ayuda. El método `add_argument_group()` acepta los argumentos *title* y *description* que pueden ser usados para personalizar esta presentación:

```
>>> 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
```

Ten en cuenta que cualquier argumento que no esté en los grupos definidos por el usuario terminará en las secciones habituales de «argumentos de posición» y «argumentos opcionales».

Exclusión mutua

`ArgumentParser.add_mutually_exclusive_group(required=False)`

Crear un grupo de exclusividad mutua. `argparse` se asegurará de que sólo uno de los argumentos del grupo de exclusividad mutua esté presente en la línea de comandos:

```
>>> 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
```

El método `add_mutually_exclusive_group()` también acepta un argumento `required`, para indicar que se requiere al menos uno de los argumentos mutuamente exclusivos:

```
>>> 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
```

Ten en cuenta que actualmente los grupos de argumentos mutuamente exclusivos no admiten los argumentos `title` y `description` de `add_argument_group()`.

Valores por defecto del analizador

`ArgumentParser.set_defaults(**kwargs)`

La mayoría de las veces, los atributos del objeto retornado por `parse_args()` se determinarán completamente inspeccionando los argumentos de la línea de comandos y las acciones de los argumentos. `set_defaults()` permite que se añadan algunos atributos adicionales que se determinan sin ninguna inspección de la línea de comandos:

```
>>> 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)
```

Ten en cuenta que los valores por defecto a nivel analizador siempre prevalecen sobre los valores por defecto a nivel argumento:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

Los valores por defecto a nivel analizador pueden ser muy útiles cuando se trabaja con varios analizadores. Consulta el método `add_subparsers()` para ver un ejemplo de este tipo.

`ArgumentParser.get_default(dest)`

Obtiene el valor por defecto para un atributo del espacio de nombres (`namespace`), establecido ya sea por `add_argument()` o por `set_defaults()`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

Mostrando la ayuda

En la mayoría de las aplicaciones típicas, `parse_args()` se encargará de dar formato y mostrar cualquier mensaje de uso o de error. Sin embargo, hay varios métodos para dar formato disponibles:

`ArgumentParser.print_usage(file=None)`

Muestra una breve descripción de cómo se debe invocar el `ArgumentParser` en la línea de comandos. Si `file` es `None`, se asume `sys.stdout`.

`ArgumentParser.print_help(file=None)`

Muestra un mensaje de ayuda, incluyendo el uso del programa e información sobre los argumentos registrados en el `ArgumentParser`. Si `file` es `None`, se asume `sys.stdout`.

También hay variantes de estos métodos que simplemente retornan una cadena de caracteres en lugar de mostrarla:

`ArgumentParser.format_usage()`

Retorna una cadena de caracteres que contiene una breve descripción de cómo se debe invocar el `ArgumentParser` en la línea de comandos.

`ArgumentParser.format_help()`

Retorna una cadena de caracteres que contiene un mensaje de ayuda, incluyendo el uso del programa e información sobre los argumentos registrados en el `ArgumentParser`.

Análisis parcial

`ArgumentParser.parse_known_args(args=None, namespace=None)`

A veces una secuencia de comandos (*script*) sólo puede analizar algunos de los argumentos de la línea de comandos, pasando el resto de los argumentos a otra secuencia o programa. En estos casos, el método `parse_known_args()` puede ser útil. Funciona de forma muy parecida a `parse_args()` excepto que no produce un error cuando hay argumentos extra presentes. En lugar de ello, retorna una tupla de dos elementos que contiene el espacio de nombres ocupado y la lista de argumentos de cadena de caracteres restantes.

```
>>> 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'])
```

Advertencia: *Coincidencia de prefijos* las reglas se aplican a `parse_known_args()`. El analizador puede consumir una opción aunque sea sólo un prefijo de una de sus opciones conocidas, en lugar de dejarla en la lista de argumentos restantes.

Personalizando el análisis de archivos

`ArgumentParser.convert_arg_line_to_args(arg_line)`

Los argumentos que se leen de un archivo (mira el argumento de palabra clave *fromfile_prefix_chars* para el constructor *ArgumentParser*) se leen uno por línea. *convert_arg_line_to_args()* puede ser invalidado para una lectura más elegante.

Este método utiliza un sólo argumento *arg_line* que es una cadena de caracteres leída desde el archivo de argumentos. Retorna una lista de argumentos analizados a partir de esta cadena de caracteres. El método se llama una vez por línea leída del fichero de argumentos, en orden.

Una alternativa útil de este método es la que trata cada palabra separada por un espacio como un argumento. El siguiente ejemplo demuestra cómo hacerlo:

```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

Métodos de salida

`ArgumentParser.exit(status=0, message=None)`

Este método finaliza el programa, saliendo con el *status* especificado y, si corresponde, muestra un *message* antes de eso. El usuario puede anular este método para manejar estos pasos de manera diferente:

```
class ErrorCatchingArgumentParser(argparse.ArgumentParser):
    def exit(self, status=0, message=None):
        if status:
            raise Exception(f'Exiting because of an error: {message}')
        exit(status)
```

`ArgumentParser.error(message)`

Este método imprime un mensaje de uso incluyendo el *message* para error estándar y finaliza el programa con código de estado 2.

Análisis entremezclado

`ArgumentParser.parse_intermixed_args(args=None, namespace=None)`

`ArgumentParser.parse_known_intermixed_args(args=None, namespace=None)`

Una serie de comandos *Unix* permiten al usuario mezclar argumentos opcionales con argumentos de posición. Los métodos *parse_intermixed_args()* y *parse_known_intermixed_args()* soportan este modo de análisis.

Estos analizadores no soportan todas las capacidades de *argparse*, y generarán excepciones si se utilizan capacidades no soportadas. En particular, los analizadores secundarios, *argparse.REMAINDER*, y los grupos mutuamente exclusivos que incluyen tanto opcionales como de posición no están soportados.

El siguiente ejemplo muestra la diferencia entre *parse_known_args()* y *parse_intermixed_args()*: el primero retorna `['2', '3']` como argumentos sin procesar, mientras que el segundo recoge todos los de posición en *rest*.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('cmd')
>>> parser.add_argument('rest', nargs='*', type=int)
>>> parser.parse_known_args('doit 1 --foo bar 2 3'.split())
(Namespace(cmd='doit', foo='bar', rest=[1]), ['2', '3'])
>>> parser.parse_intermixed_args('doit 1 --foo bar 2 3'.split())
Namespace(cmd='doit', foo='bar', rest=[1, 2, 3])
```

`parse_known_intermixed_args()` retorna una tupla de dos elementos que contiene el espacio de nombres poblado y la lista de los restantes argumentos de cadenas de caracteres. `parse_intermixed_args()` arroja un error si quedan argumentos de cadenas de caracteres sin procesar.

Nuevo en la versión 3.7.

16.4.6 Actualizar el código de *optparse*

Originalmente, el módulo *argparse* había intentado mantener la compatibilidad con *optparse*. Sin embargo, *optparse* era difícil de extender de forma transparente, particularmente con los cambios necesarios para soportar los nuevos especificadores `nargs=` y los mensajes de uso mejorados. Cuando casi todo en *optparse* había sido copiado y pegado o *monkey-patched*, ya no parecía práctico tratar de mantener la retro-compatibilidad.

El módulo *argparse* mejora la biblioteca estándar del módulo *optparse* de varias maneras, incluyendo:

- Manejando argumentos de posición.
- Soportando sub-comandos.
- Permitiendo prefijos de opción alternativos como `+` y `/`.
- Manejando argumentos de estilo cero o más y uno o más.
- Generando mensajes de uso más informativos.
- Proporcionando una interfaz mucho más simple para `type` y `action` personalizadas.

Una manera de actualizar parcialmente de *optparse* a *argparse*:

- Reemplaza todas las llamadas `optparse.OptionParser.add_option()` con llamadas `ArgumentParser.add_argument()`.
- Reemplaza `(options, args) = parser.parse_args()` con `args = parser.parse_args()` y agrega las llamadas adicionales `ArgumentParser.add_argument()` para los argumentos de posición. Ten en cuenta que lo que antes se llamaba `options`, ahora en el contexto *argparse* se llama `args`.
- Reemplaza `optparse.OptionParser.disable_interspersed_args()` por `parse_intermixed_args()` en lugar de `parse_args()`.
- Reemplaza las acciones de respuesta y los argumentos de palabra clave `callback_*` con argumentos de `type` o `action`.
- Reemplaza los nombres de cadena de caracteres por argumentos de palabra clave `type` con los correspondientes objetos tipo (por ejemplo, *int*, *float*, *complex*, etc).
- Reemplaza `optparse.Values` por *Namespace* y `optparse.OptionError` y `optparse.OptionValueError` por *ArgumentError*.
- Reemplaza las cadenas de caracteres con argumentos implícitos como `%default` o `%prog` por la sintaxis estándar de Python y usa diccionarios para dar formato a cadenas de caracteres, es decir, `%(default)s` y `%(prog)s`.
- Reemplaza el argumento `version` del constructor *OptionParser* por una llamada a `parser.add_argument('--version', action='version', version='<the version>')`.

16.5 getopt — Analizador de estilo C para opciones de línea de comando

Código fuente: `Lib/getopt.py`

Nota: El módulo `getopt` es un analizador de opciones de línea de comando cuya API está diseñada para que los usuarios de la función C `getopt()` estén familiarizados. Los usuarios que no estén familiarizados con la función C `getopt()` o que deseen escribir menos código y obtener mejor ayuda y mensajes de error deberían considerar el uso del módulo `argparse`.

Este módulo ayuda a los scripts a analizar los argumentos de la línea de comandos en `sys.argv`. Admite las mismas convenciones que la función Unix `getopt()` (incluidos los significados especiales de los argumentos de la forma “_” y “--”). También se pueden usar opciones largas similares a las admitidas por el software GNU a través de un tercer argumento opcional.

Este módulo proporciona dos funciones y una excepción:

`getopt.getopt(args, shortopts, longopts=[])`

Analiza las opciones de la línea de comandos y la lista de parámetros. *args* es la lista de argumentos a analizar, sin la referencia principal al programa en ejecución. Por lo general, esto significa `sys.argv[1:]`. *shortopts* es la cadena de letras de opciones que el script quiere reconocer, con opciones que requieren un argumento seguido de dos puntos (':'); es decir, el mismo formato que Unix `getopt()` usa).

Nota: A diferencia de GNU `getopt()`, después de un argumento no-opcional, todos los argumentos adicionales se consideran también no-opcionales. Esto es similar a la forma en que funcionan los sistemas Unix que no son GNU.

longopts, si se especifica, debe ser una lista de cadenas con los nombres de las opciones largas que deben admitirse. Los caracteres principales '-' no deben incluirse en el nombre de la opción. Las opciones largas que requieren un argumento deben ir seguidas de un signo igual ('='). Los argumentos opcionales no son compatibles. Para aceptar solo opciones largas, *shortopts* debe ser una cadena vacía. Las opciones largas en la línea de comando pueden reconocerse siempre que proporcionen un prefijo del nombre de la opción que coincida exactamente con una de las opciones aceptadas. Por ejemplo, si *longopts* es ['foo', 'frob'], la opción `--foo` coincidirá como `--foo`, pero `--f` no coincidirá de forma exclusiva, por lo que se lanzará `GetoptError`.

El valor de retorno consta de dos elementos: el primero es una lista de pares (*option*, *value*); el segundo es la lista de argumentos del programa que quedan después de que se eliminó la lista de opciones (esta es una porción final de *args*). Cada par de opción y valor retornado tiene la opción como su primer elemento, con un guión para las opciones cortas (por ejemplo, '-x') o dos guiones para las opciones largas (por ejemplo, '--long-option'), y el argumento de la opción como su segundo elemento, o una cadena vacía si la opción no tiene argumento. Las opciones aparecen en la lista en el mismo orden en que se encontraron, lo que permite múltiples ocurrencias. Las opciones largas y cortas pueden ser mixtas.

`getopt.gnu_getopt(args, shortopts, longopts=[])`

Esta función funciona como `getopt()`, excepto que el modo de escaneo estilo GNU se usa por defecto. Esto significa que los argumentos opcionales y no opcionales pueden estar mezclados. La función `getopt()` detiene el procesamiento de opciones tan pronto como se encuentra un argumento no-opcionales.

Si el primer carácter de la cadena de opciones es '+', o si la variable de entorno `POSIXLY_CORRECT` está configurada, el procesamiento de opciones se detiene tan pronto como se encuentre un argumento que no sea de opción.

exception `getopt.GetoptError`

Esto se lanza cuando se encuentra una opción no reconocida en la lista de argumentos o cuando una opción que requiere un argumento no recibe ninguna. El argumento de la excepción es una cadena que indica la causa

del error. Para opciones largas, un argumento dado a una opción que no requiere una también provocará que se lance esta excepción. Los atributos `msg` y `opt` dan el mensaje de error y la opción relacionada; si no hay una opción específica con la cual se relaciona la excepción, `opt` es una cadena vacía.

exception `getopt.error`

Alias para `GetoptError`; para compatibilidad con versiones anteriores.

Un ejemplo que usa solo opciones de estilo Unix:

```
>>> 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']
```

Usar nombres largos de opciones es igualmente fácil:

```
>>> 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', '
↪')]
>>> args
['a1', 'a2']
```

En un script, el uso típico es algo como esto:

```
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(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()
```

Tenga en cuenta que se podría generar una interfaz de línea de comando equivalente con menos código y más ayuda informativa y mensajes de error utilizando el módulo `argparse`:

```
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 ..
```

Ver también:

Módulo `argparse` Opción de línea de comando alternativa y biblioteca de análisis de argumentos.

16.6 logging — Logging facility for Python

Source code: `Lib/logging/__init__.py`

Important

This page contains the API reference information. For tutorial information and discussion of more advanced topics, see

- [Basic Tutorial](#)
- [Advanced Tutorial](#)
- [Logging Cookbook](#)

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.

16.6.1 Logger Objects

Loggers have the following attributes and methods. Note that Loggers should *NEVER* be 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`

propagate

If this attribute 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`.

Nota: 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.

setLevel (*level*)

Sets the threshold for this logger to *level*. Logging messages which are less severe than *level* will be ignored; logging messages which have severity *level* or higher will be emitted by whichever handler or handlers service this logger, unless a handler's level has been set to a higher severity level than *level*.

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.

Distinto en la versión 3.2: The *level* parameter now accepts a string representation of the level such as "INFO" as an alternative to the integer constants such as `INFO`. Note, however, that levels are internally stored as integers, and methods such as e.g. `getEffectiveLevel()` and `isEnabledFor()` will return/expect to be passed integers.

isEnabledFor (*level*)

Indicates if a message of severity *level* would be processed by this logger. This method checks first the module-level level set by `logging.disable(level)` and then the logger's effective level as determined by `getEffectiveLevel()`.

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.

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.

Nuevo en la versión 3.2.

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.) No `%` formatting operation is performed on *msg* when no *args* are supplied.

There are four keyword arguments in *kwargs* which are inspected: *exc_info*, *stack_info*, *stacklevel* and *extra*.

If *exc_info* does not evaluate as false, it causes exception information to be added to the logging message. If an exception tuple (in the format returned by `sys.exc_info()`) or an exception instance is provided, it is used; otherwise, `sys.exc_info()` is called to get the exception information.

The second optional keyword argument is *stack_info*, which defaults to `False`. If true, stack information is added to the logging message, including the actual logging call. Note that this is not the same stack information as that displayed through specifying *exc_info*: The former is stack frames from the bottom of the stack up to the logging call in the current thread, whereas the latter is information about stack frames which have been unwound, following an exception, while searching for exception handlers.

You can specify *stack_info* independently of *exc_info*, e.g. to just show how you got to a certain point in your code, even when no exceptions were raised. The stack frames are printed following a header line which says:

```
Stack (most recent call last):
```

This mimics the `Traceback (most recent call last):` which is used when displaying exception frames.

The third optional keyword argument is *stacklevel*, which defaults to 1. If greater than 1, the corresponding number of stack frames are skipped when computing the line number and function name set in the `LogRecord` created for the logging event. This can be used in logging helpers so that the function name, filename and line number recorded are not the information for the helper function/method, but rather its caller. The name of this parameter mirrors the equivalent one in the `warnings` module.

The fourth 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*.

Distinto en la versión 3.2: The *stack_info* parameter was added.

Distinto en la versión 3.5: The *exc_info* parameter can now accept exception instances.

Distinto en la versión 3.8: The *stacklevel* parameter was added.

info (*msg*, **args*, ***kwargs*)

Logs a message with level INFO on this logger. The arguments are interpreted as for *debug()*.

warning (*msg*, **args*, ***kwargs*)

Logs a message with level WARNING on this logger. The arguments are interpreted as for *debug()*.

Nota: There is an obsolete method *warn* which is functionally identical to *warning*. As *warn* is deprecated, please do not use it - use *warning* instead.

error (*msg*, **args*, ***kwargs*)

Logs a message with level ERROR on this logger. The arguments are interpreted as for *debug()*.

critical (*msg*, **args*, ***kwargs*)

Logs a message with level CRITICAL on this logger. The arguments are interpreted as for *debug()*.

log (*level*, *msg*, **args*, ***kwargs*)

Logs a message with integer level *level* on this logger. The other arguments are interpreted as for *debug()*.

exception (*msg*, **args*, ***kwargs*)

Logs a message with level ERROR on this logger. The arguments are interpreted as for *debug()*. Exception info is added to the logging message. This method should only be called from an exception handler.

addFilter (*filter*)

Adds the specified filter *filter* to this logger.

removeFilter (*filter*)

Removes the specified filter *filter* from this logger.

filter (*record*)

Apply this logger’s filters to the record and return *True* 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.

addHandler (*hdlr*)

Adds the specified handler *hdlr* to this logger.

removeHandler (*hdlr*)

Removes the specified handler *hdlr* from this logger.

findCaller (*stack_info=False*, *stacklevel=1*)

Finds the caller’s source filename and line number. Returns the filename, line number, function name and

stack information as a 4-element tuple. The stack information is returned as `None` unless `stack_info` is `True`.

The `stacklevel` parameter is passed from code calling the `debug()` and other APIs. If greater than 1, the excess is used to skip stack frames before determining the values to be returned. This will generally be useful when calling logging APIs from helper/wrapper code, so that the information in the event log refers not to the helper/wrapper code, but to the code that calls it.

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()`.

makeRecord (*name, level, fn, lno, msg, args, exc_info, func=None, extra=None, sinfo=None*)

This is a factory method which can be overridden in subclasses to create specialized `LogRecord` instances.

hasHandlers ()

Checks to see if this logger has any handlers configured. This is done by looking for handlers in this logger and its parents in the logger hierarchy. Returns `True` if a handler was found, else `False`. The method stops searching up the hierarchy whenever a logger with the “propagate” attribute set to false is found - that will be the last logger which is checked for the existence of handlers.

Nuevo en la versión 3.2.

Distinto en la versión 3.7: Loggers can now be pickled and unpickled.

16.6.2 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

16.6.3 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__()`.

class `logging.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.

createLock ()

Initializes a thread lock which can be used to serialize access to underlying I/O functionality which may not be threadsafe.

acquire ()

Acquires the thread lock created with `createLock()`.

release()

Releases the thread lock acquired with *acquire()*.

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.

Distinto en la versión 3.2: The *level* parameter now accepts a string representation of the level such as “INFO” as an alternative to the integer constants such as INFO.

setFormatter(fmt)

Sets the *Formatter* for this handler to *fmt*.

addFilter(filter)

Adds the specified filter *filter* to this handler.

removeFilter(filter)

Removes the specified filter *filter* from this handler.

filter(record)

Apply this handler’s filters to the record and return `True` 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.

flush()

Ensure all logging output has been flushed. This version does nothing and is intended to be implemented by subclasses.

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.

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.

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).

format(record)

Do formatting for a record - if a formatter is set, use it. Otherwise, use the default formatter for the module.

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*.

16.6.4 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 *Formateo de cadenas al estilo *printf** 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, style='%'*)

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, a format is used which is described in the *formatTime()* documentation.

The *style* parameter can be one of `"%"`, `"{"` or `"$"` and determines how the format string will be merged with its data: using one of %-formatting, *str.format()* or *string.Template*. See formatting-styles for more information on using `{-}` and `$-formatting` for log messages.

Distinto en la versión 3.2: The *style* parameter was added.

Distinto en la versión 3.8: The *validate* parameter was added. Incorrect or mismatched style and *fmt* will raise a *ValueError*. For example: `logging.Formatter('%(asctime)s - %(message)s', style='{')`.

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.

If stack information is available, it's appended after the exception information, using *formatStack()* to transform it if necessary.

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 format `"%Y-%m-%d %H:%M:%S,uuu"` is used, where the *uuu* part is a millisecond value and the other letters are as per the *time.strftime()* documentation. An example time in this format is `2003-01-23 00:29:50,411`. 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.

Distinto en la versión 3.3: Previously, the default format was hard-coded as in this example: `2010-09-06 22:38:15,292` where the part before the comma is handled by a *strftime* format

string ('%Y-%m-%d %H:%M:%S'), and the part after the comma is a millisecond value. Because `strptime` does not have a format placeholder for milliseconds, the millisecond value is appended using another format string, '%s,%03d' — and both of these format strings have been hardcoded into this method. With the change, these strings are defined as class-level attributes which can be overridden at the instance level when desired. The names of the attributes are `default_time_format` (for the `strptime` format string) and `default_msec_format` (for appending the millisecond value).

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.

formatStack (*stack_info*)

Formats the specified stack information (a string as returned by `traceback.print_stack()`, but with the last newline removed) as a string. This default implementation just returns the input value.

16.6.5 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.

Distinto en la versión 3.2: You don't need to create specialized `Filter` classes, or use other classes with a `filter` method: you can use a function (or other callable) as a filter. The filtering logic will check to see if the filter object has a `filter` attribute: if it does, it's assumed to be a `Filter` and its `filter()` method is called. Otherwise, it's assumed to be a callable and called with the record as the single parameter. The returned value should conform to that returned by `filter()`.

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`).

16.6.6 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*, *sinfo=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.

Parámetros

- **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*: *levelname* 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.
- **sinfo** – A text string representing stack information from the base of the stack in the current thread, up to the logging call.

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.

Distinto en la versión 3.2: The creation of a *LogRecord* has been made more configurable by providing a factory which is used to create the record. The factory can be set using *getLogRecordFactory()* and *setLogRecordFactory()* (see this for the factory's signature).

This functionality can be used to inject your own values into a *LogRecord* at creation time. You can use the following pattern:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdeadbeef
    return record

logging.setLogRecordFactory(record_factory)
```

With this pattern, multiple factories could be chained, and as long as they don't overwrite each other's attributes or unintentionally overwrite the standard attributes listed above, there should be no surprises.

16.6.7 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.

If you are using {}-formatting (`str.format()`), you can use `{attrname}` as the placeholder in the format string. If you are using \$-formatting (`string.Template`), use the form `${attrname}`. In both cases, of course, replace `attrname` with the actual attribute name you want to use.

In the case of {}-formatting, you can specify formatting flags by placing them after the attribute name, separated from it with a colon. For example: a placeholder of `{msecs:03d}` would format a millisecond value of 4 as 004. Refer to the `str.format()` documentation for full details on the options available to you.

Attribute name	Format	Description
args	You shouldn't need to format this yourself.	The tuple of arguments merged into <code>msg</code> to produce <code>message</code> , or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	<code>%(asctime)s</code>	Human-readable time when the <code>LogRecord</code> 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	<code>%(created)f</code>	Time when the <code>LogRecord</code> was created (as returned by <code>time.time()</code>).
exc_info	You shouldn't need to format this yourself.	Exception tuple (à la <code>sys.exc_info</code>) or, if no exception has occurred, <code>None</code> .
filename	<code>%(filename)s</code>	Filename portion of <code>pathname</code> .
funcName	<code>%(funcName)s</code>	Name of function containing the logging call.
levelname	<code>%(levelname)s</code>	Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL').
levelno	<code>%(levelno)s</code>	Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL).
lineno	<code>%(lineno)d</code>	Source line number where the logging call was issued (if available).
message	<code>%(message)s</code>	The logged message, computed as <code>msg % args</code> . This is set when <code>Formatter.format()</code> is invoked.
module	<code>%(module)s</code>	Module (name portion of <code>filename</code>).
msecs	<code>%(msecs)d</code>	Millisecond portion of the time when the <code>LogRecord</code> was created.
msg	You shouldn't need to format this yourself.	The format string passed in the original logging call. Merged with <code>args</code> to produce <code>message</code> , or an arbitrary object (see arbitrary-object-messages).
name	<code>%(name)s</code>	Name of the logger used to log the call.
pathname	<code>%(pathname)s</code>	Full pathname of the source file where the logging call was issued (if available).
process	<code>%(process)d</code>	Process ID (if available).
processName	<code>%(processName)s</code>	Process name (if available).
relativeCreated	<code>%(relativeCreated)d</code>	Time in milliseconds when the <code>LogRecord</code> was created, relative to the time the logging module was loaded.
stack_info	You shouldn't need to format this yourself.	Stack frame information (where available) from the bottom of the stack in the current thread, up to and including the stack frame of the logging call which resulted in the creation of this record.
thread	<code>%(thread)d</code>	Thread ID (if available).
threadName	<code>%(threadName)s</code>	Thread name (if available).

Distinto en la versión 3.1: `processName` was added.

16.6.8 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.

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()*, *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* and *handlers*. These methods have the same signatures as their counterparts in *Logger*, so you can use the two types of instances interchangeably.

Distinto en la versión 3.2: The *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* and *handlers* methods were added to *LoggerAdapter*. These methods delegate to the underlying logger.

16.6.9 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.

16.6.10 Module-Level Functions

In addition to the classes described above, there are a number of module-level functions.

`logging.getLogger` (*name=None*)

Return a logger with the specified name or, if *name* is *None*, 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.getLogRecordFactory` ()

Return a callable which is used to create a *LogRecord*.

Nuevo en la versión 3.2: This function has been provided, along with *setLogRecordFactory()*, to allow developers more control over how the *LogRecord* representing a logging event is constructed.

See *setLogRecordFactory()* for more information about the how the factory is called.

`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 three 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()`) or an exception instance is provided, it is used; otherwise, `sys.exc_info()` is called to get the exception information.

The second optional keyword argument is *stack_info*, which defaults to `False`. If true, stack information is added to the logging message, including the actual logging call. Note that this is not the same stack information as that displayed through specifying *exc_info*: The former is stack frames from the bottom of the stack up to the logging call in the current thread, whereas the latter is information about stack frames which have been unwound, following an exception, while searching for exception handlers.

You can specify *stack_info* independently of *exc_info*, e.g. to just show how you got to a certain point in your code, even when no exceptions were raised. The stack frames are printed following a header line which says:

```
Stack (most recent call last):
```

This mimics the `Traceback (most recent call last):` which is used when displaying exception frames.

The third 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*.

Distinto en la versión 3.2: The *stack_info* parameter 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()`.

Nota: There is an obsolete function `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

`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()`. Exception info is 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()`.

Nota: 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(level=CRITICAL)`

Provides an overriding level `level` 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 `level` 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.

Note that if you have defined any custom logging level higher than `CRITICAL` (this is not recommended), you won't be able to rely on the default value for the `level` parameter, but will have to explicitly supply a suitable value.

Distinto en la versión 3.7: The `level` parameter was defaulted to level `CRITICAL`. See [bpo-28524](#) for more information about this change.

`logging.addLevelName(level, levelName)`

Associates level `level` 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.

Nota: If you are thinking of defining your own levels, please see the section on custom-levels.

`logging.getLevelName(level)`

Returns the textual or numeric representation of logging level `level`.

If `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 `level` is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned.

The `level` parameter also accepts a string representation of the level such as "INFO". In such cases, this functions returns the corresponding numeric value of the level.

If no matching numeric or string value is passed in, the string "Level %s" % level is returned.

Nota: Levels are internally integers (as they need to be compared in the logging logic). 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*), and vice versa.

Distinto en la versión 3.4: In Python versions earlier than 3.4, this function could also be passed a text level, and would return the corresponding numeric value of the level. This undocumented behaviour was considered a mistake, and was removed in Python 3.4, but reinstated in 3.4.2 due to retain backward compatibility.

`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, unless the keyword argument *force* is set to *True*.

Nota: 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
<i>filename</i>	Specifies that a <i>FileHandler</i> be created, using the specified filename, rather than a <i>StreamHandler</i> .
<i>filemode</i>	If <i>filename</i> is specified, open the file in this <i>mode</i> . Defaults to 'a'.
<i>format</i>	Use the specified format string for the handler. Defaults to attributes <i>levelname</i> , <i>name</i> and <i>message</i> separated by colons.
<i>datefmt</i>	Use the specified date/time format, as accepted by <i>time.strftime()</i> .
<i>style</i>	If <i>format</i> is specified, use this style for the format string. One of '%', '{' or '\$' for <i>printf-style</i> , <i>str.format()</i> or <i>string.Template</i> respectively. Defaults to '%'
<i>level</i>	Set the root logger level to the specified <i>level</i> .
<i>stream</i>	Use the specified stream to initialize the <i>StreamHandler</i> . Note that this argument is incompatible with <i>filename</i> - if both are present, a <i>ValueError</i> is raised.
<i>handlers</i>	If specified, this should be an iterable of already created handlers to add to the root logger. Any handlers which don't already have a formatter set will be assigned the default formatter created in this function. Note that this argument is incompatible with <i>filename</i> or <i>stream</i> - if both are present, a <i>ValueError</i> is raised.
<i>force</i>	If this keyword argument is specified as true, any existing handlers attached to the root logger are removed and closed, before carrying out the configuration as specified by the other arguments.

Distinto en la versión 3.2: The *style* argument was added.

Distinto en la versión 3.3: The *handlers* argument was added. Additional checks were added to catch situations where incompatible arguments are specified (e.g. *handlers* together with *stream* or *filename*, or *stream* together with *filename*).

Distinto en la versión 3.8: The *force* argument was added.

`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.

When the logging module is imported, it registers this function as an exit handler (see *atexit*), so normally there's no need to do that manually.

`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. After this call, as at any other time, do not instantiate loggers directly using the subclass: continue to use the `logging.getLogger()` API to get your loggers.

`logging.setLogRecordFactory(factory)`

Set a callable which is used to create a *LogRecord*.

Parámetros *factory* – The factory callable to be used to instantiate a log record.

Nuevo en la versión 3.2: This function has been provided, along with `getLogRecordFactory()`, to allow developers more control over how the *LogRecord* representing a logging event is constructed.

The factory has the following signature:

```
factory(name, level, fn, lno, msg, args, exc_info, func=None,
        sinfo=None, **kwargs)
```

name The logger name.

level The logging level (numeric).

fn The full pathname of the file where the logging call was made.

lno The line number in the file where the logging call was made.

msg The logging message.

args The arguments for the logging message.

exc_info An exception tuple, or None.

func The name of the function or method which invoked the logging call.

sinfo A stack traceback such as is provided by `traceback.print_stack()`, showing the call hierarchy.

kwargs Additional keyword arguments.

16.6.11 Module-Level Attributes

`logging.lastResort`

A «handler of last resort» is available through this attribute. This is a *StreamHandler* writing to `sys.stderr` with a level of `WARNING`, and is used to handle logging events in the absence of any logging configuration. The end result is to just print the message to `sys.stderr`. This replaces the earlier error message saying that «no handlers could be found for logger XYZ». If you need the earlier behaviour for some reason, `lastResort` can be set to None.

Nuevo en la versión 3.2.

16.6.12 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).

Ver también:

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.

16.7 logging.config — Configuración de registro

Código fuente: `Lib/logging/config.py`

Important

Esta página solo contiene información de referencia. Para tutoriales, por favor consulte

- Tutorial Básico
- Tutorial Avanzado
- Guía de registro *Logging*

Esta sección describe la API para configurar el módulo de registro.

16.7.1 Funciones de configuración

Las siguientes funciones configuran el módulo de registro. Se encuentran en el módulo `logging.config`. Su uso es opcional: puede configurar el módulo de registro utilizando estas funciones o realizando llamadas a la API principal (definida en `logging`) y definiendo los manejadores que se declaran en `logging` o `logging.handlers`.

`logging.config.dictConfig(config)`

Toma la configuración de registro de un diccionario. El contenido de este diccionario se describe en *Esquema del diccionario de configuración* a continuación.

Si se encuentra un error durante la configuración, esta función lanzará un `ValueError`, `TypeError`, `AttributeError` o `ImportError` con un mensaje descriptivo adecuado. La siguiente es una lista (posiblemente incompleta) de condiciones que lanzará un error:

- Un `nivel` que no es una cadena o que es una cadena que no corresponde a un nivel de registro real.
- Un valor de `propagación` que no es booleano.

- Una identificación que no tiene un destino correspondiente.
- Una identificación de manejador inexistente encontrada durante una llamada incremental.
- Un nombre de registrador no válido.
- Incapacidad para resolver un objeto interno o externo.

El análisis se realiza mediante la clase `DictConfigurator`, a cuyo constructor se le pasa el diccionario utilizado para la configuración, y tiene un método `configure()`. El módulo `logging.config` tiene un atributo invocable `dictConfigClass` que se establece inicialmente en `DictConfigurator`. Puede reemplazar el valor de `dictConfigClass` con una implementación adecuada propia.

`dictConfig()` llama `dictConfigClass` pasando el diccionario especificado, y luego llama al método `configure()` en el objeto retornado para que la configuración surta efecto:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

Por ejemplo, una subclase de `DictConfigurator` podría llamar a `DictConfigurator.__init__()` en su mismo `__init__()`, luego configurar prefijos personalizados que serían utilizables en la siguiente llamada `configure()`. `dictConfigClass` estaría vinculado a esta nueva subclase, y luego `dictConfig()` podría llamarse exactamente como en el estado predeterminado, no personalizado.

Nuevo en la versión 3.2.

`logging.config.fileConfig(fname, defaults=None, disable_existing_loggers=True)`

Lee la configuración de registro desde un archivo de formato de `configparser`. El formato del archivo debe ser como se describe en *Formato de archivo de configuración*. Esta función se puede invocar varias veces desde una aplicación, lo que permite al usuario final seleccionar entre varias configuraciones predeterminadas (si el desarrollador proporciona un mecanismo para presentar las opciones y cargar la configuración elegida).

Parámetros

- **fname** – Un nombre de archivo, o un objeto similar a un archivo, o una instancia derivada de `RawConfigParser`. Si se pasa una instancia derivada de `RawConfigParser`, se usa tal cual. De lo contrario, se instancia `ConfigParser`, y la configuración leída desde el objeto pasado en `fname`. Si eso tiene un método `readline()`, se supone que es un objeto similar a un archivo y se lee usando `read_file()`; de lo contrario, se supone que es un nombre de archivo y se pasa a `read()`.
- **defaults** – Los valores predeterminados para pasar al `ConfigParser` se pueden especificar en este argumento.
- **disable_existing_loggers** – Si se especifica como `False`, los registradores que existen cuando se realiza esta llamada se dejan habilitados. El valor predeterminado es `True` porque esto permite un comportamiento antiguo de una manera compatible con versiones anteriores. Este comportamiento consiste en deshabilitar cualquier registrador no root existente a menos que ellos o sus antepasados se mencionen explícitamente en la configuración de registro.

Distinto en la versión 3.4: Una instancia de una subclase de `RawConfigParser` ahora se acepta como un valor para `fname`. Esto facilita:

- Uso de un archivo de configuración donde la configuración de registro es solo parte de la configuración general de la aplicación.
- Uso de una configuración leída de un archivo, y luego modificada por la aplicación que lo usa (por ejemplo, basada en parámetros de línea de comandos u otros aspectos del entorno de ejecución) antes de pasar a `fileConfig`.

`logging.config.listen(port=DEFAULT_LOGGING_CONFIG_PORT, verify=None)`

Inicia un servidor de socket en el puerto especificado y escucha nuevas configuraciones. Si no se especifica ningún puerto, se usa el valor predeterminado del módulo `DEFAULT_LOGGING_CONFIG_PORT`. Las configuraciones de registro se enviarán como un archivo adecuado para su procesamiento por `dictConfig()`

o `fileConfig()`. Retorna una instancia de `Thread` en la que puede llamar `start()` para iniciar el servidor, y que puede `join()` cuando corresponda. Para detener el servidor, llame a `stopListening()`.

El argumento `verificar`, si se especifica, debe ser invocable, lo que debería verificar si los bytes recibidos en el socket son válidos y deben procesarse. Esto podría hacerse encriptando y / o firmando lo que se envía a través del socket, de modo que el `verificar` invocable pueda realizar la verificación o descifrado de la firma. El llamado invocable `verificar` se llama con un solo argumento (los bytes recibidos a través del socket) y debe retornar los bytes a procesar, o `None` para indicar que los bytes deben descartarse. Los bytes retornados podrían ser los mismos que los pasados en bytes (por ejemplo, cuando solo se realiza la verificación), o podrían ser completamente diferentes (tal vez si se realizó el descifrado).

Para enviar una configuración al socket, lea el archivo de configuración y envíelo al socket como una secuencia de bytes precedida por una cadena de longitud de cuatro bytes empaquetada en binario usando `struct.pack('>L', n)`.

Nota: Debido a que partes de la configuración se pasan a través de `eval()`, el uso de esta función puede abrir a sus usuarios a un riesgo de seguridad. Si bien la función solo se une a un socket en `localhost` y, por lo tanto, no acepta conexiones de máquinas remotas, hay escenarios en los que se puede ejecutar código no confiable bajo la cuenta del proceso que llama `listen()`. Específicamente, si el proceso que llama `listen()` se ejecuta en una máquina multiusuario donde los usuarios no pueden confiar el uno en el otro, entonces un usuario malintencionado podría hacer arreglos para ejecutar código esencialmente arbitrario en el proceso del usuario víctima, simplemente conectándose al socket `listen()` de la víctima y enviando una configuración que ejecuta cualquier código que el atacante quiera ejecutar en el proceso de la víctima. Esto es especialmente fácil de hacer si se usa el puerto predeterminado, pero no es difícil incluso si se usa un puerto diferente). Para evitar el riesgo de que esto suceda, use el argumento `verificar` para `listen()` para evitar que se apliquen configuraciones no reconocidas.

Distinto en la versión 3.4: Se agregó el argumento `verificar`.

Nota: Si desea enviar configuraciones al oyente que no deshabiliten los registradores existentes, deberá usar un formato JSON para la configuración, que utilizará `dictConfig()` para la configuración. Este método le permite especificar `disable_existing_loggers` como `False` en la configuración que envía.

`logging.config.stopListening()`

Detiene el servidor de escucha que se creó con una llamada a `listen()`. Esto normalmente se llama antes de llamar `join()` en el valor de retorno de `listen()`.

16.7.2 Esquema del diccionario de configuración

La descripción de una configuración de registro requiere una lista de los diversos objetos para crear y las conexiones entre ellos; por ejemplo, puede crear un manejador llamado “consola” y luego decir que el registrador llamado “inicio” enviará sus mensajes al manejador “consola”. Estos objetos no se limitan a los proporcionados por el módulo `logging` porque podría escribir su propia clase de formateador o manejador. Los parámetros de estas clases también pueden necesitar incluir objetos externos como `sys.stderr`. La sintaxis para describir estos objetos y conexiones se define en *Conexiones de objeto* a continuación.

Detalles del esquema del diccionario

El diccionario pasado a `dictConfig()` debe contener las siguientes claves:

- *version* - se establece en un valor entero que representa la versión del esquema. El único valor válido en este momento es 1, pero tener esta clave permite que el esquema evolucione sin perder la compatibilidad con versiones anteriores.

Todas las demás claves son opcionales, pero si están presentes se interpretarán como se describe a continuación. En todos los casos a continuación, donde se menciona un “dict de configuración”, se verificará la clave especial `'()'` para ver si se requiere una instanciación personalizada. Si es así, el mecanismo descrito en *Objetos definidos por el usuario* a continuación se usa para crear una instancia; de lo contrario, el contexto se usa para determinar qué instanciar.

- *formatters*: el valor correspondiente será un diccionario en el que cada clave es una identificación de formateador y cada valor es un diccionario que describe cómo configurar la instancia correspondiente *Formatter*.

La configuración diccionario se busca para las claves `format` y `datefmt` (con los valores predeterminados de `None`) y se utilizan para construir una instancia de *Formatter*.

Distinto en la versión 3.8: se puede agregar una tecla `validar` (con el valor predeterminado `True`) en la sección `formatters` de la configuración diccionario, esto es para validar el formato.

- *filters*: el valor correspondiente será un diccionario en el que cada clave es una identificación de filtro y cada valor es un diccionario que describe cómo configurar la instancia de filtro correspondiente.

El diccionario de configuración busca la clave `nombre` (por defecto en la cadena vacía) y esto se utiliza para construir una instancia de *logging.Filter*.

- *handlers*: el valor correspondiente será un diccionario en el que cada clave es una identificación de manejador y cada valor es un diccionario que describe cómo configurar la instancia del manejador correspondiente.

El diccionario de configuración busca las siguientes claves:

- `clase` (obligatorio). Este es el nombre completo de la clase de manejador.
- `nivel` (opcional). El nivel del manejador.
- `formateador` (opcional). La identificación del formateador para este manejador.
- `filtros` (opcional). Una lista de identificadores de los filtros para este manejador.

Todas las claves *other* se pasan como argumentos de palabras clave al constructor del manejador. Por ejemplo, dado el fragmento:

```
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
```

el manejador con id `consola` se instancia como *logging.StreamHandler*, usando `sys.stdout` como la secuencia subyacente. El manejador con la identificación `archivo` se instancia como *logging.handlers.RotatingFileHandler* con los argumentos de la palabra clave `filename='logconfig.log'`, `maxBytes=1024`, `backupCount=3`.

- *loggers*: el valor correspondiente será un diccionario en el que cada clave es un nombre de *logger* y cada valor es un diccionario que describe cómo configurar la instancia de *Logger* correspondiente.

El diccionario de configuración busca las siguientes claves:

- `nivel` (opcional). El nivel del registrador.
- `propagar` (opcional). La configuración de propagación del registrador.
- `filtros` (opcional). Una lista de identificadores de los filtros para este registrador.
- `manipuladores` (opcional). Una lista de identificadores de los manejadores para este registrador.

Los registradores especificados se configurarán de acuerdo con el nivel, la propagación, los filtros y los manejadores especificados.

- *root* - Esta será la configuración para el registrador *root*. El procesamiento de la configuración será como para cualquier registrador, excepto que la configuración de `propagar` no será aplicable.
- *incremental* - si la configuración debe interpretarse como incremental a la configuración existente. Este valor predeterminado es `False`, lo que significa que la configuración especificada reemplaza la configuración existente con la misma semántica que la utilizada por la API existente `fileConfig()`.

Si el valor especificado es `True`, la configuración se procesa como se describe en la sección sobre [Configuración incremental](#).

- *disable_existing_loggers* - si se debe deshabilitar cualquier registrador no *root* existente. Esta configuración refleja el parámetro del mismo nombre en `fileConfig()`. Si está ausente, este parámetro tiene el valor predeterminado `True`. Este valor se ignora si *incremental* es `True`.

Configuración incremental

Es difícil proporcionar flexibilidad completa para la configuración incremental. Por ejemplo, debido a que los objetos como los filtros y formateadores son anónimos, una vez que se configura una configuración, no es posible hacer referencia a dichos objetos anónimos al aumentar una configuración.

Además, no hay un caso convincente para alterar arbitrariamente el gráfico de objetos de registradores, manejadores, filtros, formateadores en tiempo de ejecución, una vez que se configura una configuración; la verbosidad de los registradores y manejadores se puede controlar simplemente estableciendo niveles (y, en el caso de los registradores, flags de propagación). Cambiar el gráfico de objetos de manera arbitraria y segura es problemático en un entorno de subprocesos múltiples; Si bien no es imposible, los beneficios no valen la complejidad que agrega a la implementación.

Por lo tanto, cuando la tecla *incremental* de un diccionario de configuración está presente y es `True`, el sistema ignorará por completo cualquier entrada de `formateadores` y `filtros`, y procesará solo el `nivel` configuraciones en las entradas de `manejadores`, y las configuraciones de `nivel` y `propagar` en las entradas de `registradores` y `raíz`.

El uso de un valor en la configuración diccionario permite que las configuraciones se envíen a través del cable como dictados en vinagre a un escucha de socket. Por lo tanto, la verbosidad de registro de una aplicación de larga ejecución puede modificarse con el tiempo sin necesidad de detener y reiniciar la aplicación.

Conexiones de objeto

El esquema describe un conjunto de objetos de registro (registradores, manejadores, formateadores, filtros) que están conectados entre sí en un gráfico de objetos. Por lo tanto, el esquema necesita representar conexiones entre los objetos. Por ejemplo, supongamos que, una vez configurado, un registrador particular le ha adjuntado un manejador particular. A los fines de esta discusión, podemos decir que el registrador representa la fuente y el manejador el destino de una conexión entre los dos. Por supuesto, en los objetos configurados esto está representado por el registrador que tiene una referencia al manejador. En la configuración dict, esto se hace dando a cada objeto de destino una identificación que lo identifica inequívocamente, y luego utilizando la identificación en la configuración del objeto de origen para indicar que existe una conexión entre el origen y el objeto de destino con esa identificación.

Entonces, por ejemplo, considere el siguiente fragmento de YAML:

```
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]
```

(Nota: YAML se usa aquí porque es un poco más legible que el formulario fuente Python equivalente para el diccionario.)

Los identificadores para los registradores son los nombres de los registradores que se utilizarían mediante programación para obtener una referencia a esos registradores, por ejemplo `foo.bar.baz`. Los identificadores para Formateadores y Filtros pueden ser cualquier valor de cadena (como `breve`, `preciso` arriba) y son transitorios, ya que solo son significativos para procesar el diccionario de configuración y se utilizan para determinar conexiones entre objetos, y no persisten en ninguna parte cuando se completa la llamada de configuración.

El fragmento anterior indica que el registrador llamado `foo.bar.baz` debe tener dos manejadores adjuntos, que se describen mediante los identificadores de manejador `h1` y `h2`. El formateador para `h1` es el descrito por identificador `brief`, y el formateador para `h2` es el descrito por identificador `precise`.

Objetos definidos por el usuario

El esquema admite objetos definidos por el usuario para manejadores, filtros y formateadores. (Los registradores no necesitan tener diferentes tipos para diferentes instancias, por lo que no hay soporte en este esquema de configuración para las clases de registrador definidas por el usuario).

Los objetos a configurar son descritos por diccionarios que detallan su configuración. En algunos lugares, el sistema de registro podrá inferir del contexto cómo se va a instanciar un objeto, pero cuando se va a instanciar un objeto definido por el usuario, el sistema no sabrá cómo hacerlo. Con el fin de proporcionar una flexibilidad completa para la creación de instancias de objetos definidos por el usuario, el usuario debe proporcionar una “fábrica”, una llamada que se llama con un diccionario de configuración y que retorna el objeto instanciado. Esto se indica mediante una ruta de importación absoluta a la fábrica disponible bajo la tecla especial `'()`'. Aquí hay un ejemplo concreto:

```
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
    bar: baz
    spam: 99.9
    answer: 42
```

El fragmento de YAML anterior define tres formateadores. El primero, con identificador “breve”, es una instancia estándar `logging.Formatter` con la cadena de formato especificada. El segundo, con identificador predeterminado, tiene un formato más largo y también define el formato de hora explícitamente, y dará como resultado `logging.Formatter` inicializado con esas dos cadenas de formato. En forma de fuente Python, los formateadores `breve` y `predeterminado` tienen sub-diccionarios de configuración:

```
{
    'format' : '%(message)s'
}
```

y:

```
{
    'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
    'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

respectivamente, y como estos diccionarios no contienen la clave especial '()', la instanciación se infiere del contexto: como resultado, se crean las instancias estándar `logging.Formatter`. El sub-diccionario de configuración para el tercer formateador, con identificador personalizado, es:

```
{
    '()' : 'my.package.customFormatterFactory',
    'bar' : 'baz',
    'spam' : 99.9,
    'answer' : 42
}
```

y esto contiene la clave especial '()', lo que significa que se desea la creación de instancias definida por el usuario. En este caso, se utilizará la llamada especificada de fábrica especificada. Si es una llamada real, se usará directamente; de lo contrario, si especifica una cadena (como en el ejemplo), la llamada real se ubicará utilizando mecanismos de importación normales. Se llamará al invocable con los elementos **restantes** en el sub-diccionario de configuración como argumentos de palabras clave. En el ejemplo anterior, se supondrá que el formateador con identificador personalizado será retornado por la llamada:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

La clave '()' se ha utilizado como clave especial porque no es un nombre de parámetro de palabra clave válido, por lo que no entrará en conflicto con los nombres de los argumentos de palabras clave utilizados en la llamada. El '()' también sirve como mnemónico de que el valor correspondiente es invocable.

Acceso a objetos externos

Hay momentos en que una configuración debe referirse a objetos externos a la configuración, por ejemplo, `sys.stderr`. Si el diccionario de configuración se construye utilizando el código Python, esto es sencillo, pero surge un problema cuando la configuración se proporciona a través de un archivo de texto (por ejemplo, JSON, YAML). En un archivo de texto, no hay una forma estándar de distinguir `sys.stderr` de la cadena literal `'sys.stderr'`. Para facilitar esta distinción, el sistema de configuración busca ciertos prefijos especiales en valores de cadena y los trata especialmente. Por ejemplo, si la cadena literal `'ext://sys.stderr'` se proporciona como un valor en la configuración, entonces la `ext://` se eliminará y se procesará el resto del valor utilizando mecanismos normales de importación.

El manejo de dichos prefijos se realiza de manera análoga al manejo del protocolo: existe un mecanismo genérico para buscar prefijos que coincidan con la expresión regular `^(?P<prefix>[a-z]+):/(?P<suffix>.*)$` por el cual, si se reconoce el `prefix`, el `suffix` se procesa de manera dependiente del prefijo y el resultado del procesamiento reemplaza el valor de la cadena. Si no se reconoce el prefijo, el valor de la cadena se dejará tal cual.

Acceso a objetos internos

Además de los objetos externos, a veces también es necesario hacer referencia a los objetos en la configuración. El sistema de configuración hará esto implícitamente para las cosas que conoce. Por ejemplo, el valor de cadena 'DEBUG' para un nivel en un registrador o manejador se convertirá automáticamente al valor `logging.DEBUG`, y los manejadores, las entradas de filtros y formateador tomarán una identificación de objeto y se resolverán en el objeto de destino apropiado.

Sin embargo, se necesita un mecanismo más genérico para los objetos definidos por el usuario que no conoce el módulo `logging`. Por ejemplo, considere `logging.handlers.MemoryHandler`, que toma un argumento `target` que es otro manejador para delegar. Dado que el sistema ya conoce esta clase, entonces en la configuración, el objetivo dado solo necesita ser la identificación del objeto del manejador de destino relevante, y el sistema resolverá el manejador desde la identificación. Sin embargo, si un usuario define un `my.package.MyHandler` que tiene un manejador alternativo, el sistema de configuración no sabría que el `alternativo` se refería a un manejador. Para atender esto, un sistema de resolución genérico permite al usuario especificar:

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
  alternate: cfg://handlers.file
```

La cadena literal `'cfg://handlers.file'` se resolverá de manera análoga a las cadenas con el prefijo `ext:/`, pero buscando en la configuración misma en lugar del espacio de nombres de importación. El mecanismo permite el acceso por punto o por índice, de manera similar a la proporcionada por `str.format`. Por lo tanto, dado el siguiente fragmento:

```
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.
```

en la configuración, la cadena `'cfg://handlers'` se resolvería al diccionario con la clave `handlers`, la cadena de caracteres `'cfg://handlers.email'` se resolvería al diccionario con clave correo electrónico en el diccionario `manejadores`, y así sucesivamente. La cadena de caracteres `'cfg://handlers.email.toaddrs[1]'` se resolvería en `'dev_team.domain.tld'` y la cadena de caracteres `'cfg://handlers.email.toaddrs[0]'` resolvería el valor `'support_team@domain.tld'`. Se puede acceder al valor de asunto usando `'cfg://handlers.email.subject'` o, de manera equivalente, `'cfg://handlers.email[subject]'`. La última forma solo debe usarse si la clave contiene espacios o caracteres no alfanuméricos. Si un valor de índice consta solo de dígitos decimales, se intentará acceder utilizando el valor entero correspondiente, volviendo al valor de cadena si es necesario.

Dada una cadena `cfg://handlers.myhandler.mykey.123`, esto se resolverá en `config_dict['handlers']['myhandler']['mykey']['123']`. Si la cadena se especifica como `cfg://handlers.myhandler.mykey[123]`, el sistema intentará recuperar el valor de `config_dict['handlers']['myhandler']['mykey'][123]`, y vuelva a `config_dict['handlers']['myhandler']['mykey']['123']` si eso falla.

Resolución de importación e importadores personalizados

La resolución de importación, por defecto, utiliza la función incorporada `__import__()` para importar. Es posible que desee reemplazar esto con su propio mecanismo de importación: si es así, puede reemplazar el atributo `importer` de `DictConfigurator` o su superclase, la clase `BaseConfigurator`. Sin embargo, debe tener cuidado debido a la forma en que se accede a las funciones desde las clases a través de descriptores. Si está utilizando un Python invocable para realizar sus importaciones, y lo desea definir a nivel de clase en lugar de a nivel de instancia, debe envolverlo con `staticmethod()`. Por ejemplo:

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

No necesita envolver con `staticmethod()` si está configurando la importación invocable en un configurador *instance*.

16.7.3 Formato de archivo de configuración

El formato del archivo de configuración que entiende `fileConfig()` se basa en la funcionalidad `configparser`. El archivo debe contener secciones llamadas `[loggers]`, `[handlers]` y `[formateadores]` que identifican por nombre las entidades de cada tipo que se definen en el archivo. Para cada una de esas entidades, hay una sección separada que identifica cómo se configura esa entidad. Por lo tanto, para un registrador llamado `log01` en la sección `[loggers]`, los detalles de configuración relevantes se encuentran en una sección `[logger_log01]`. Del mismo modo, un manejador llamado `hand01` en la sección `[handlers]` tendrá su configuración en una sección llamada `[handler_hand01]`, mientras que un formateador llamado `form01` en el `[formateadores]` sección tendrá su configuración especificada en una sección llamada `[formatter_form01]`. La configuración del registrador raíz debe especificarse en una sección llamada `[logger_root]`.

Nota: La API `fileConfig()` es más antigua que la API `dictConfig()` y no proporciona funcionalidad para cubrir ciertos aspectos del registro. Por ejemplo, no puede configurar objetos `Filter`, que permiten el filtrado de mensajes más allá de niveles enteros simples, usando `fileConfig()`. Si necesita tener instancias de `Filter` en su configuración de registro, deberá usar `dictConfig()`. Tenga en cuenta que las mejoras futuras a la funcionalidad de configuración se agregarán a `dictConfig()`, por lo que vale la pena considerar la transición a esta API más nueva cuando sea conveniente hacerlo.

A continuación se dan ejemplos de estas secciones en el archivo.

```
[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
```

El registrador raíz debe especificar un nivel y una lista de manejadores. A continuación se muestra un ejemplo de una sección de registrador raíz.

```
[logger_root]
level=NOTSET
handlers=hand01
```

La entrada de nivel puede ser una de `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL` o `NOTSET`. Solo para el registrador raíz, `NOTSET` significa que todos los mensajes se registrarán. Los valores de nivel son `eval()` uados en el contexto del espacio de nombres del paquete `logging`.

La entrada `manejadores` es una lista separada por comas de nombres de manejadores, que debe aparecer en la sección `[manejadores]`. Estos nombres deben aparecer en la sección `[manejadores]` y tener las secciones correspondientes en el archivo de configuración.

Para los registradores que no sean el registrador raíz, se requiere información adicional. Esto se ilustra en el siguiente ejemplo.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

Las entradas de `level` y `manejadores` se interpretan como para el registrador raíz, excepto que si el nivel de un registrador no raíz se especifica como `NOTSET`, el sistema consulta a los registradores más arriba en la jerarquía para determinar el nivel efectivo del registrador. La entrada `propagar` se establece en 1 para indicar que los mensajes deben propagarse a los manejadores que están más arriba en la jerarquía del registrador, o 0 para indicar que los mensajes **no** se propagan a los manejadores en la jerarquía superior. La entrada `qualname` es el nombre jerárquico del canal del registrador, es decir, el nombre utilizado por la aplicación para obtener el registrador.

Las secciones que especifican la configuración del manejador se ejemplifican a continuación.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

La entrada `class` indica la clase del manejador (según lo determinado por `eval()` en el espacio de nombres del paquete `logging`). El “nivel” se interpreta como para los registradores, y `NOTSET` se entiende como “registrar todo”.

La entrada `formateador` indica el nombre clave del formateador para este manejador. Si está en blanco, se utiliza un formateador predeterminado (`logging._defaultFormatter`). Si se especifica un nombre, debe aparecer en la sección `[formateadores]` y tener una sección correspondiente en el archivo de configuración.

La entrada `args`, cuando `eval()` sea en el contexto del espacio de nombres del paquete `logging`, es la lista de argumentos para el constructor de la clase de manejador. Consulte los constructores de los manejadores relevantes, o los ejemplos a continuación, para ver cómo se construyen las entradas típicas. Si no se proporciona, el valor predeterminado es `()`.

La entrada opcional `kwargs`, cuando `eval()` sea en el contexto del espacio de nombres del paquete `logging`, es el argumento de palabra clave diccionario al constructor para la clase de manejador. Si no se proporciona, el valor predeterminado es `{}`.

```
[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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
[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')
kwargs={'timeout': 10.0}

[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')
kwargs={'secure': True}
```

Las secciones que especifican la configuración del formateador se caracterizan por lo siguiente.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

La entrada `format` es la cadena de formato general, y la entrada `datefmt` es una cadena de formato fecha/hora `strftime()` compatible. Si está vacío, el paquete sustituye algo que es casi equivalente a especificar la cadena de formato de fecha `'%Y-%m-%d %H:%M:%S'`. Este formato también especifica milisegundos, que se agregan al resultado del uso de la cadena de formato anterior, con un separador de coma. Un ejemplo de tiempo en este formato es `2003-01-23 00:29:50,411`.

La entrada `clase` es opcional. Indica el nombre de la clase del formateador (como módulo de puntos y nombre de clase). Esta opción es útil para crear instancias de una subclase `Formatter`. Las subclases de `Formatter` pueden presentar trazas de excepción en un formato expandido o condensado.

Nota: Debido al uso de `eval()` como se describió anteriormente, existen riesgos potenciales de seguridad que resultan del uso de `listen()` para enviar y recibir configuraciones a través de sockets. Los riesgos se limitan a donde múltiples usuarios sin confianza mutua ejecutan código en la misma máquina; consulte la documentación de `listen()` para obtener más información.

Ver también:

Módulo `logging` Referencia de API para el módulo de registro.

Módulo `logging.handlers` Manejadores útiles incluidos con el módulo de registro.

16.8 logging.handlers — Gestores de logging

Código fuente [Lib/logging/handlers.py](#)

Important

Esta página contiene solo información de referencia. Para tutoriales por favor véase

- Tutorial Básico
- Tutorial avanzado
- Libro de cocina de *Logging*

Estos gestores son muy útiles y están provistos en este paquete. Nota que tres de los gestores de las clases (*StreamHandler*, *FileHandler* and *NullHandler*) están definidos en propio módulo *logging* pero fueron documentados aquí junto con los otros gestores.

16.8.1 StreamHandler

La clase *StreamHandler* ubicada en el paquete núcleo *logging* envía la salida del logging a un *stream* como *sys.stdout*, *sys.stderr* o cualquier objeto tipo archivo (o mas precisamente cualquier objeto que soporte los métodos *write()* y *flush()*).

class logging.*StreamHandler* (*stream=None*)

Retorna una nueva instancia de la clase *StreamHandler*. Si *stream* esta especificado, la instancia lo usará para la salida del registro, sino se usará *sys.stderr*.

emit (*record*)

Si esta especificado un formateador se usará para formatear el registro. Luego se escribe el registro al *stream* con un terminador. Si hay información de excepción presente, se formateará usando *traceback.print_exception()* y se añadirá al *stream*.

flush ()

Descarga el *stream* llamando a su método *flush()*. Nota que el método *close()* es heredado de la clase *Handler* y por lo tanto no produce ninguna salida. Por eso muchas veces será necesario invocar al método explícito *flush()*.

setStream (*stream*)

Establece el *stream* de la instancia a un valor especifico, si este es diferente. El anterior *stream* es vaciado antes de que el nuevo *stream* sea establecido.

Parámetros *stream* – El *stream* que el gestor debe usar.

Devuelve el anterior *stream*. si el *stream* cambió o *None* si no cambió.

Nuevo en la versión 3.7.

Distinto en la versión 3.2: La clase *StreamHandler* ahora tiene un atributo *terminator* con valor por defecto `''\n'`, que se usa como terminador cuando se escribe un registro formateado a un *stream*. Si no se quiere esta terminación de nueva línea, se puede establecer el atributo de la instancia del gestor “terminator” para vaciar la cadena de caracteres. En versiones anteriores el terminador era codificado como `'\n'`.

16.8.2 FileHandler

La clase `FileHandler` está localizada en el paquete núcleo `logging`, envía la salida del `logging` a un archivo de disco. Hereda la funcionalidad de salida de la clase `StreamHandler`.

class `logging.FileHandler` (*filename, mode='a', encoding=None, delay=False*)

Retorna una nueva instancia de la clase `FileHandler`. Abre el archivo especificado y se usa como un *stream* para `logging`. Si *mode* no se especifica se usa 'a'. Si *encoding* no es `None`, se usará para abrir el archivo con esa codificación. Si *delay* es `true` entonces se difiere la apertura del archivo hasta la primer llamada al método `emit()`. Por defecto el archivo crece indefinidamente.

Distinto en la versión 3.6: Así como valores de cadena de caracteres, también se aceptan objetos de la clase `Path` para el argumento «*filename*».

close()

Cierra el archivo.

emit (*record*)

Da la salida del registro al archivo.

16.8.3 NullHandler

Nuevo en la versión 3.1.

La clase `NullHandler` está ubicada en el núcleo biblioteca `logging`. No realiza ningún formateo o salida. Es en esencia un gestor “no-op” para uso de desarrolladores de bibliotecas.

class `logging.NullHandler`

Retorna una nueva instancia de la clase `NullHandler`.

emit (*record*)

Este método no realiza ninguna acción.

handle (*record*)

Este método no realiza ninguna acción.

createLock ()

Este método retorna `None` para el bloqueo, dado que no hay una E/S subyacente cuyo acceso se necesite serializar.

Véase `library-config` para mas información en como usar la clase `NullHandler`.

16.8.4 WatchedFileHandler

La clase `WatchedFileHandler` está ubicada en el módulo `logging.handlers`, es una clase `FileHandler` que vigila a que archivo se está enviando el `logging`. Si el archivo cambia, este se cerrará y se volverá a abrir usando el nombre de archivo.

Puede suceder que haya un cambio de archivo por uso de programas como `newsyslog` y `logrotate` que realizan una rotación del archivo log. Este gestor destinado para uso bajo Unix/Linux controla el archivo para ver si hubo cambios desde la última emisión. (Un archivo se considera que cambió si su dispositivo o nodo índice cambió). Si el archivo cambió entonces el anterior *stream* de archivo se cerrará, y se abrirá el nuevo para obtener un nuevo *stream*.

Este gestor no es apropiado para uso bajo Windows porque bajo Windows los archivos log abiertos no se pueden mover o renombrar. `Logging` abre los archivos con bloqueos exclusivos y entonces no hay necesidad de usar el gestor. Por otra parte `ST_INO` no es soportado bajo Windows. La función `stat()` siempre retorna cero para este valor.

class `logging.handlers.WatchedFileHandler` (*filename, mode='a', encoding=None, delay=False*)

Retorna una nueva instancia de la clase `WatchedFileHandler`. El archivo especificado se abre y usa como el *stream* para `logging`. Si *mode* no se especifica se usará 'a'. Si *encoding* no es `None`, se usa para abrir el archivo con esa codificación. Si *delay* es `true` se diferirá la apertura del archivo hasta tanto se llame al método `emit()`. Por defecto el archivo crecerá indefinidamente.

Distinto en la versión 3.6: Así como valores de cadena de caracteres, también se aceptan objetos de la clase `Path` para el argumento «*filename*».

reopenIfNeeded()

Revisa si el archivo cambió. Si hubo cambio, el *stream* existente se vacía y cierra y el archivo se abre nuevamente. Típicamente es un precursor para dar salida del registro a un archivo.

Nuevo en la versión 3.6.

emit(record)

Da salida al registro a un archivo, pero primero invoca al método `reopenIfNeeded()` para reabrir el archivo si es que cambió.

16.8.5 BaseRotatingHandler

La clase `BaseRotatingHandler` ubicada en el módulo `logging.handlers` es la clase base para rotar los gestores de archivos de clases `RotatingFileHandler` y `TimedRotatingFileHandler`. No debería ser necesario instanciar esta clase, pero tiene métodos y atributos que quizá se necesiten sobrescribir (*override*).

class `logging.handlers.BaseRotatingHandler` (*filename*, *mode*, *encoding=None*, *delay=False*)

Los parámetros son como los de la clase `FileHandler`. Los atributos son:

namer

Si este atributo se establece como invocable, el método `rotation_filename()` delega a este invocable. Los parámetros pasados al invocable son aquellos pasados al método `rotation_filename()`.

Nota: La función de nombrado es invocada unas cuantas veces durante el volcado (*rollover*), entonces debe ser tan simple y rápida como sea posible. Debe también retornar siempre la misma salida para una misma entrada, de otra manera el volcado puede no funcionar como se espera.

Nuevo en la versión 3.3.

rotator

Si este atributo se estableció como invocable, el método `rotate()` delega a este invocable. Los parámetros pasados al invocable son aquellos pasados al método `rotate()`.

Nuevo en la versión 3.3.

rotation_filename(default_name)

Modifica el nombre de un archivo log cuando esta rotando.

Esto esta previsto para que pueda usarse un nombre de archivo personalizado.

La implementación por defecto llama al atributo “namer” del gestor, si este es invocable, pasando el nombre por defecto a él. Si el atributo no es invocable (por defecto es `None`) el nombre se retorna sin cambios.

Parámetros default_name – El nombre por defecto para el archivo de log.

Nuevo en la versión 3.3.

rotate(source, dest)

Cuando está rotando, rotar el actual log.

La implementación por defecto llama al atributo “rotator” del gestor, si es invocable, pasando los argumentos de origen y destino a él. Si no se puede invocar (porque el atributo por defecto es “None”) el origen es simplemente renombrado al destino.

Parámetros

- **source** – El nombre de archivo origen. Normalmente el nombre de archivo base, por ejemplo “test.log”.

- **dest** – El nombre de archivo de destino. Normalmente es el nombre al que se rota el archivo origen por ejemplo “test.log.1”.

Nuevo en la versión 3.3.

La razón de que existen los atributos es para evitar tener que usar una subclase - puedes usar los mismos invocadores para instancias de clase `RotatingFileHandler` y `TimedRotatingFileHandler`. Si el rotador invocable o la función de nombrado plantean una excepción esta se manejará de la misma manera que cualquier otra excepción durante una llamada al método `emit()` por ejemplo a través del método `handleError()` del gestor.

Si necesitas hacer cambios mas significativos al proceso de rotación puedes obviar los métodos.

Para un ejemplo véase `cookbook-rotator-namer`.

16.8.6 RotatingFileHandler

La clase `RotatingFileHandler`, localizada en el módulo `logging.handlers` soporta la rotación de archivos log de disco.

class `logging.handlers.RotatingFileHandler` (*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=False*)

Retorna una nueva instancia de la clase `RotatingFileHandler`. El archivo especificado es abierto y usado como un *stream* para *logging*. Si *mode* no se especifica, se usará 'a'. Si *encoding* no es None se usará para abrir el archivo con esa codificación. Si *delay* es verdadero entonces la apertura del archivo se diferirá hasta la primer llamada al método `emit()`. Por defecto el archivo crece indefinidamente.

Se pueden usar los valores *maxBytes* y *backupCount* para permitir que el archivo *rollover* tenga un tamaño predeterminado. Cuando el tamaño del archivo está a punto de excederse, se cerrará y un nuevo archivo se abrirá silenciosamente para salida. El volcado (*rollover*) ocurre cada vez que el actual archivo log esta cerca de *maxBytes* en tamaño, pero si cualquiera *maxBytes* o *backupCount* es cero, el volcado (*rollover*) no ocurre. Por eso generalmente necesitas establecer *backupCount* por lo menos en 1 y no tener cero en *maxBytes*. Cuando *backupCount* no es cero, el sistema guardará los anteriores archivos log agregando las extensiones “.1”, “.2” etc. al nombre del archivo. Por ejemplo con un *backupCount* de 5 y un nombre de archivo base de `app.log`, tendrás como nombre de archivo `app.log`, `app.log.1`, `app.log.2`, hasta `app.log.5`. El archivo que esta siendo escrito es siempre `app.log`. Cuando este se completa, se cierra y se renombra a `app.log.1` y si ya existen `app.log.1`, `app.log.2`, etc. Entonces se renombrará como `app.log.2`, `app.log.3` etc. respectivamente.

Distinto en la versión 3.6: Así como valores de cadena de caracteres, también se aceptan objetos de la clase `Path` para el argumento «*filename*».

doRollover()

Realiza un volcado (*rollover*) como se describe arriba.

emit(record)

Da la salida del registro al archivo, dando suministro para el volcado (*rollover*) como está descrito anteriormente.

16.8.7 TimedRotatingFileHandler

La clase `TimedRotatingFileHandler` está ubicada en el módulo `logging.handlers`. Soporta la rotación de archivos de log a ciertos intervalos de tiempo.

class `logging.handlers.TimedRotatingFileHandler` (*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*, *atTime=None*)

Retorna una nueva instancia de la clase `TimedRotatingFileHandler`. El archivo especificado es abierto y usado como *stream* para el historial de log. En la rotación también establece el sufijo del nombre de archivo. La rotación ocurre basada en el producto de *when* y *interval*.

Puedes usar el *when* para especificar el tipo de intervalo *interval*. La lista de posibles valores esta debajo. Nota que no distingue mayúsculas y minúsculas.

Valor	Tipo de intervalo	Si/como <i>atTime</i> es usado
'S'	Segundos	Ignorado
'M'	Minutos	Ignorado
'H'	Horas	Ignorado
'D'	Días	Ignorado
'W0' - 'W6'	Día de la semana (0=Lunes)	Usado para calcular la hora inicial del volcado <i>rollover</i>
'midnight'	Volcado (<i>rollover</i>) a medianoche , si <i>atTime</i> no está especificado, sino el volcado se hará <i>atTime</i>	Usado para calcular la hora inicial del volcado <i>rollover</i>

Cuando se usa rotación basada en día de la semana, especifica “W0” para Lunes, “W1” para Martes y así, hasta “W6” para Domingo. en este caso el valor pasado para *Interval* no se usa.

El sistema guardará los archivos de log anteriores agregándoles extensiones al nombre de archivo. Las extensiones están basadas en día-hora usando el formato `%Y-%m-%d_%H-%M-%S` o un prefijo respecto el intervalo del volcado (*rollover*).

Cuando se calcula la hora del siguiente volcado (*rollover*) por primera vez (cuando el gestor es creado), la última hora de modificación de un archivo log existente o sino la hora actual, se usa para calcular cuando será la próxima rotación.

Si el argumento *utc* es *true* se usará la hora en UTC, sino se usará la hora local.

Si *backupCount* no es cero, se conservará como máximo una cantidad de archivos especificada en *backupCount*, y si son creados más, cuando ocurre el volcado (*rollover*) se borrará el último. La lógica de borrado usa el intervalo para determinar que archivos borrar, pues entonces cambiando el intervalo puede dejar viejos archivos abandonados.

Si *delay* es *true* entonces la apertura del archivo se demorará hasta la primer llamada a *emit()*.

Si *atTime* no es «None», debe haber una instancia `datetime.time` que especifica la hora que ocurre el volcado (*rollover*) , para los casos en que el volcado esta establecido para ocurrir «a medianoche» o «un día en particular». Nótese que en estos casos el valor *atTime* se usa para calcular el valor *initial* del volcado (*rollover*) y los subsecuentes volcados serán calculados a través del calculo normal de intervalos.

Nota: El cálculo de la hora en que se realizara el volcado (*rollover*) inicial cuando se inicializa el gestor. El cálculo de la hora de los siguientes volcados (*rollovers*) se realiza solo cuando este ocurre, y el volcado ocurre solo cuando se emite una salida. Si esto no se tiene en cuenta puede generar cierta confusión. Por ejemplo si se establece un intervalo de «cada minuto» eso no significa que siempre se verán archivos log con hora (en el nombre del archivo) separados por un minuto. Si durante la ejecución de la aplicación el *logging* se genera con mayor frecuencia que un minuto entonces se pueden esperar archivos log separados por un minuto. Si por otro lado los mensajes *logging* son establecidos cada digamos cinco minutos, entonces habrá brechas de tiempo en los archivos correspondientes a los minutos que no hubo salida (y no ocurrió por tanto volcado alguno).

Distinto en la versión 3.4: Se agregó el parámetro *atTime*.

Distinto en la versión 3.6: Así como valores de cadena de caracteres, también se aceptan objetos de la clase *Path* para el argumento «*filename*».

doRollover()

Realiza un volcado (*rollover*) como se describe arriba.

emit(record)

Da la salida del registro a un archivo , proveyendo la información para el volcado (*rollover*) como esta descripto anteriormente.

16.8.8 SocketHandler

La clase `SocketHandler` esta localizada en el módulo `logging.handlers`, envía el *logging* a un socket de la red. La clase base usa *sockets* TCP.

class `logging.handlers.SocketHandler` (*host*, *port*)

Retorna una nueva instancia de la clase `SocketHandler` destinada para comunicarse con una terminal remota cuya dirección esta dada por *host* y *port*.

Distinto en la versión 3.4: Si «port» se especifica como “None” se crea un socket de dominio Unix, usando el valor en “host” - de otra manera se creará un socket TCP.

close ()

Cierra el socket.

emit ()

Serializa (*Pickles*) el registro del diccionario de atributos y lo escribe en el socket en formato binario. Si hay un error con el socket, silenciosamente descarta el paquete. Si la conexión se perdió previamente, la restablece. Para deserializar (*unpickle*) un registro en el extremo receptor a una clase `LogRecord`, usa la función `makeLogRecord()`.

handleError ()

Maneja un error que ocurrió durante el método `emit()`. La causa mas común es una perdida de conexión. Cierra el socket para que podamos reintentar en el próximo evento.

makeSocket ()

Este es un método patrón que permite subclases para definir el tipo preciso de socket que se necesita. La implementación por defecto crea un socket TCP(`socket.SOCK_STREAM`).

makePickle (*record*)

Serializa (*pickles*) el registro del diccionario de atributos en formato binario con un prefijo de tamaño, y lo retorna listo para transmitir a través del socket. Los detalles de esta operación son equivalentes a:

```
data = pickle.dumps(record_attr_dict, 1)
datalen = struct.pack('>L', len(data))
return datalen + data
```

Nota que los serializados (*pickles*) no son totalmente seguros. Si te preocupa la seguridad desearás evitar este método para implementar un mecanismo mas seguro. Por ejemplo puedes firmar *pickles* usando HMAC y verificarlos después en el extremo receptor. O alternatively puedes deshabilitar la deserialización (*unpickling*) de objetos globales en el extremo receptor.

send (*packet*)

Envía un paquete serializado (*pickled*) de cadena de caracteres al socket. El formato de la cadena de bytes enviada es tal como se describe en la documentación de `makePickle()`.

Esta función permite envíos parciales, que pueden ocurrir cuando la red esta ocupada.

createSocket ()

Intenta crear un socket, si hay una falla usa un algoritmo de marcha atrás exponencial. En el fallo inicial el gestor desechará el mensaje que intentaba enviar. Cuando los siguientes mensajes sean gestionados por la misma instancia no intentará conectarse hasta que haya transcurrido cierto tiempo. Los parámetros por defecto son tales que el retardo inicial es un segundo y si después del retardo la conexión todavía no se puede realizar, el gestor doblará el retardo cada vez hasta un máximo de 30 segundos.

Este comportamiento es controlado por los siguientes atributos del gestor:

- `retryStart` (retardo inicial por defecto 1.0 segundos)
- `retryFactor` (multiplicador por defecto 2.0)
- `retryMax` (máximo retardo por defecto 30.0 segundos)

Esto significa que si el oyente remoto (*listener*) comienza después de que se usó el gestor, pueden perderse mensajes (dado que el gestor no puede siquiera intentar una conexión hasta que se haya cumplido el retardo, y silenciosamente desechará los mensajes mientras se cumpla el retardo).

16.8.9 DatagramHandler

La clase `DatagramHandler` está ubicada en el módulo `logging.handlers`, hereda de la clase `SocketHandler` para realizar el soporte de mensajes *logging* por los *sockets* UDP.

class `logging.handlers.DatagramHandler` (*host*, *port*)

Retorna una nueva instancia de la clase `DatagramHandler` destinada para comunicarse con la terminal remota cuya dirección es dada por *host* y *port*.

Distinto en la versión 3.4: Si “*port*” se especifica como «None», se crea un socket de dominio Unix, usando el valor en «*host*» - de otra manera se crea un socket UDP.

emit ()

Serializa (*pickles*) el registro del diccionario de atributos y lo escribe en el socket en formato binario. Si hay un error con el socket, silenciosamente desecha el paquete. Para deserializar (*unpickle*) el registro en el extremo de recepción a una clase `LogRecord`, usa la función `makeLogRecord()`.

makeSocket ()

El método original de la clase `SocketHandler` se omite para crear un socket UDP (`socket.SOCK_DGRAM`).

send (*s*)

Enviar una cadena de caracteres serializada (*pickled*) a un socket de red. El formato de la cadena de *bytes* enviado es tal como se describe en la documentación para `SocketHandler.makePickle()`.

16.8.10 Gestor SysLog (SysLogHandler)

La clase `SysLogHandler` está ubicada en el módulo `logging.handlers`. Realiza el soporte de los mensajes de *logging* a un *syslog* Unix local o remoto.

class `logging.handlers.SysLogHandler` (*address*=('localhost', `SYSLOG_UDP_PORT`), *facility*=`LOG_USER`, *socktype*=`socket.SOCK_DGRAM`)

Retorna una nueva instancia de la clase `SysLogHandler` concebida para comunicarse con una terminal remota Unix cuya dirección esta dada por *address* en la forma de una tupla (*host*, *port*) . Si *address* no se especifica se usará ('localhost', 514) . La dirección se usa para abrir el socket. Una alternativa a consignar una tupla (*host*, *port*) es proveer una dirección como cadena de caracteres, por ejemplo “/dev/log”. En este caso se usa un socket de dominio Unix para enviar el mensaje al syslog. Si *facility* no se especifica se usara `LOG_USER` . El tipo de socket abierto usado depende del argumento *socktype* , que por defecto es `socket.SOCK_DGRAM` y por lo tanto abre un socket UDP . Para abrir un socket TCP (para usar con los nuevos *daemons* *syslog* como *Rsyslog*) se debe especificar un valor de `socket.SOCK_STREAM`.

Nótese que si el servidor no esta escuchando el puerto UDP 514, la clase `SysLogHandler` puede parecer no funcionar. En ese caso chequea que dirección deberías usar para un socket de dominio . Es sistema-dependiente. Por ejemplo en Linux generalmente es “/dev/log” pero en OS/X es “/var/run/syslog”. Será necesario chequear tu plataforma y usar la dirección adecuada (quizá sea necesario realizar este chequeo mientras corre la aplicación si necesita correr en diferentes plataformas). En Windows seguramente tengas que usar la opción UDP.

Distinto en la versión 3.2: Se agregó *socktype*.

close ()

Cierra el socket del host remoto.

emit (*record*)

El registro es formateado, y luego enviado al servidor *syslog*. Si hay información de excepción presente entonces no se enviará al servidor.

Distinto en la versión 3.2.1: (Véase el [bpo-12168](#).) En versiones anteriores , los mensajes enviados a los *daemons* *syslog* siempre terminaban con un byte NUL ya que versiones anteriores de estos *daemons* esperaban un mensaje NUL de terminación. Incluso a pesar que no es relevante la especificación ([RFC 5424](#)). Versiones mas recientes de estos *daemons* no esperan el byte NUL pero lo quitan si esta ahí. Versiones aún mas recientes que están mas cercanas a la especificación RFC 5424 pasan el byte NUL como parte del mensaje.

Para habilitar una gestión mas sencilla de los mensajes *syslog* respecto de todos esos *daemons* de diferentes comportamientos el agregado del byte NUL es configurable a través del uso del atributo de nivel de clase “append_nul”. Este es por defecto “True (preservando el comportamiento ya existente) pero se puede establecer a “False” en una instancia `SysLogHandler` como para que esa instancia no añada el terminador NUL.

Distinto en la versión 3.3: (Véase: issue “12419”) en versiones anteriores, no había posibilidades para un prefijo “ident” o “tag” para identificar el origen del mensaje. Esto ahora se puede especificar usando un atributo de nivel de clase, que por defecto será «”» para preservar el comportamiento existente , pero puede ser sorteado en una instancia `SysLogHandler` para que esta instancia anteponga el *ident* a todos los mensajes gestionados. Nótese que el *ident* provisto debe ser texto, no bytes y se antepone al mensaje tal como es.

encodePriority (*facility, priority*)

Codifica la funcionalidad y prioridad en un entero. Puedes pasar cadenas de caracteres o enteros, si pasas cadenas de caracteres se usarán los diccionarios de mapeo interno para convertirlos en enteros.

Los valores simbólicos LOG_ están definidos en `SysLogHandler` e invierten los valores definidos en el archivo de encabezado `sys/syslog.h`.

Prioridades

Nombre (cadena de caracteres)	Valor simbólico
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

Nombre (cadena de caracteres)	Valor simbólico
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*)

Mapea un nombre de nivel *logging* a un nombre de prioridad *syslog*. Puedes necesitar omitir esto si estas

usando niveles personalizados, o si el algoritmo por defecto no es aplicable a tus necesidades. El algoritmo por defecto mapea `DEBUG`, `INFO`, `WARNING`, `ERROR` y `CRITICAL` a sus nombres equivalentes *syslog*, y todos los demás nombres de nivel a “warning”.

16.8.11 Gestor de eventos *NTELog* (*NTEventLogHandler*)

La clase *NTEventLogHandler* esta localizada en el módulo *logging.handlers*, soporta el envío de mensajes de *logging* a un log de eventos local Windows NT, Windows 2000 o Windows XP. Antes de que puedas usarlo, necesitarás tener instaladas las extensiones de Win32 de Mark Hammond para Python.

```
class logging.handlers.NTEventLogHandler (appname,          dllname=None,          logtype=  
                                         pe='Application')
```

Retorna una nueva instancia de la clase *NTEventLogHandler* la *appname* se usa para definir el nombre de la aplicación tal como aparece en el log de eventos. Se crea una entrada de registro apropiada usando este nombre. El *dllname* debe dar la ruta completa calificada de un *.dll* o *.exe* que contiene definiciones de mensaje para conservar en el log. (si no esta especificada, se usará “win32service.pyd” esto es instalado con las extensiones de Win32 y contiene algunas definiciones básicas de mensajes de conservación de lugar. Nótese que el uso de estos conservadores de lugar harán tu log de eventos extenso, dado que el origen completo del mensaje es guardado en el log. Si quieres *logs* menos extensos deberás pasar el nombre de tu propio *.dll* o *.exe* que contiene la definición de mensajes que quieres usar en el log. El *logtype* puede ser de 'Application', 'System' or 'Security' y sino por defecto será de 'Application'.

close()

Llegado a este punto puedes remover el nombre de aplicación del registro como origen de entrada de log de eventos. Sin embargo si haces esto no te será posible ver los eventos tal como has propuesto en el visor del log de eventos - necesita ser capaz de acceder al registro para tomar el nombre *.dll*. Esto no lo hace la versión actual.

emit (record)

Determina el ID del mensaje, categoría y tipo de evento y luego registra el mensaje en el log de eventos NT.

getEventCategory (record)

Retorna la categoría de evento del registro. Evita esto si quieres especificar tus propias categorías. Esta versión retorna 0.

getEventType (record)

Retorna el tipo de evento del registro. Haz caso omiso de esto si quieres especificar tus propios tipos. Esta versión realiza un mapeo usando el atributo *typemap* del gestor, que se establece en `__init__()` a un diccionario que contiene mapeo para `DEBUG`, `INFO`, `WARNING`, `ERROR` y `CRITICAL`. Si estas usando tus propios niveles, necesitarás omitir este método o colocar un diccionario a medida en el atributo *typemap* del gestor.

getMessageID (record)

Retorna el ID de mensaje para el registro. Si estas usando tus propios mensajes, podrás hacerlo pasando el *msg* al *logger* siendo un ID mas que un formato de cadena de caracteres. Luego aquí puedes usar una búsqueda de diccionario para obtener el ID de mensaje. Esta versión retorna 1, que es el ID de mensaje base en *win32service.pyd*.

16.8.12 SMTPHandler

La clase `SMTPHandler` esta ubicada en el módulo `logging.handlers` y soporta el envío de mensajes de `logging` a un a dirección de correo electrónico a través de SMTP.

class `logging.handlers.SMTPHandler` (*mailhost, fromaddr, toaddrs, subject, credentials=None, secure=None, timeout=1.0*)

Retorna una nueva instancia de la clase `SMTPHandler`. Esta instancia se inicializa con la dirección de: y para: y asunto: del correo electrónico. El *toaddrs* debe ser una lista de cadena de caracteres. Para especificar un puerto SMTP no estandarizado usa el formato de tupla (host, puerto) para el argumento *mailhost*. Si usas una cadena de caracteres, se utiliza el puerto estándar SMTP. Si tu servidor SMTP necesita autenticación, puedes especificar una tupla (usuario, contraseña) para el argumento de *credentials*.

Para especificar el uso de un protocolo de seguridad (TLS), pasa una tupla al argumento *secure*. Esto solo se utilizará cuando sean provistas las credenciales de autenticación. La tupla deberá ser una tupla vacía o una tupla con único valor con el nombre de un archivo-clave *keyfile*, o una tupla de 2 valores con el nombre del archivo-clave *keyfile* y archivo certificado. (Esta tupla se pasa al método `smtplib.SMTP.starttls()` method.).

Se puede especificar un tiempo de espera para comunicación con el servidor SMTP usando el argumento *timeout*.

Nuevo en la versión 3.3: Se agregó el argumento *timeout*.

emit (*record*)

Formatea el registro y lo envía a las direcciones especificadas.

getSubject (*record*)

Si quieres especificar una línea de argumento que es registro-dependiente, sobrescribe (*override*) este método.

16.8.13 MemoryHandler

La clase `MemoryHandler` esta ubicada en el módulo `logging.handlers`. Soporta el almacenamiento temporal de registros `logging` en memoria. Periódicamente los descarga al gestor *target*. Esto ocurre cuando el búfer está lleno o cuando ocurre un evento de cierta importancia.

`MemoryHandler` es una subclase de la clase mas general `BufferingHandler`, que es una clase abstracta. Este almacena temporalmente registros `logging` en la memoria. Cada vez que un registro es agregado al búfer, se realiza una comprobación llamando al método `shouldFlush()` para ver si dicho búfer debe ser descargado. Si debiera, entonces el método `flush()` se espera que realice la descarga.

class `logging.handlers.BufferingHandler` (*capacity*)

Inicializa el gestor con un búfer de una capacidad específica. Aquí capacidad significa el número de registros `logging` en el almacenamiento temporal.

emit (*record*)

Añade un registro al búfer. Si el método `shouldFlush()` retorna *true*, entonces llama al método `flush()` para procesar el búfer.

flush ()

Puedes sobrescribir (*override*) esto para implementar un comportamiento “a medida” de la descarga. Esta versión solo vacía el búfer.

shouldFlush (*record*)

Retorna *True* si el búfer tiene aún capacidad. Este método puede ser omitido para implementar estrategias a medida de vaciado.

class `logging.handlers.MemoryHandler` (*capacity, flushLevel=ERROR, target=None, flushOnClose=True*)

Retorna una nueva instancia de la clase `MemoryHandler`. La instancia se inicializa con un búfer del tamaño *capacity*. Si el *flushLevel* no se especifica, se usará *ERROR*. Si no se especifica *target* el objetivo deberá especificarse usando el método `setTarget()` -Antes de esto el gestor no realizará nada útil. Si se especifica

flushOnClose como `False`, entonces el búfer no se vaciará cuando el gestor se cierra. Si no se especifica o se especifica como `True`, el comportamiento previo de vaciado del búfer sucederá cuando se cierre el gestor.

Distinto en la versión 3.6: Se añadió el parámetro *flushOnClose*.

close()

Invoca al método *flush()* y establece el objetivo a “None” y vacía el búfer.

flush()

Para la clase *MemoryHandler* el vaciado significa simplemente enviar los registros del búfer al objetivo, si es que hay uno. El búfer además se vacía cuando esto ocurre. Sobrescribe (*override*) si deseas un comportamiento diferente.

setTarget(target)

Establece el gestor de objetivo para este gestor.

shouldFlush(record)

Comprueba si el búfer esta lleno o un registro igual o mas alto que *flushLevel*.

16.8.14 HTTPHandler

La clase *HTTPHandler* está ubicada en el módulo *logging.handlers*. Soporta el envío de mensajes *logging* a un servidor Web, usando la semántica `GET` o `POST`.

class `logging.handlers.HTTPHandler`(*host, url, method='GET', secure=False, credentials=None, context=None*)

Retorna una nueva instancia de la clase *HTTPHandler*. el *host* puede ser de la forma «host:puerto», y necesitarás usar un numero de puerto específico. Si no se especifica *method* se usará `GET`. Si *secure* es `true` se usará una conexión HTTPS. El parámetro *context* puede establecerse a una instancia *ssl.SSLContext* para establecer la configuración de SSL usado en la conexión HTTPS. Si se especifica *credentials* debe ser una tupla doble, consistente en usuario y contraseña, que se colocará en un encabezado de autorización HTTP usando autenticación básica. Si especificas credenciales también deberás especificar *secure=True* así tu usuario y contraseña no son pasados como texto en blanco por la conexión.

Distinto en la versión 3.5: Se agregó el parámetro *context*.

mapLogRecord(record)

Provee un diccionario, basado en *record* para ser codificado en forma URL y enviado al servidor web. La implementación por defecto retorna *record.__dict__*. Este método puede omitirse si por ejemplo solo se enviará al servidor web un subconjunto de la clase *LogRecord* o si se requiere enviar al servidor algo mas específico.

emit(record)

Envía el registro al servidor Web como un diccionario con codificación URL. Se usa el método *mapLogRecord()* para convertir el registro al diccionario que debe ser enviado.

Nota: Dado que preparar un registro para enviar a un servidor Web no es lo mismo que una operación genérica de formato, usando el método *setFormatter()* para especificar una clase *Formatter* por una clase *HTTPHandler* no tiene efecto. En vez de llamar al método *format()* este gestor llama al método *mapLogRecord()* y después a la función *urllib.parse.urlencode()* para codificar el diccionario en una forma que sea adecuada para enviar a un servidor Web.

16.8.15 QueueHandler

Nuevo en la versión 3.2.

La clase `QueueHandler` localizada en el módulo `logging.handlers` soporta el envío de mensajes de *logging* a una cola, tal como los implementados en los módulos `queue` o `multiprocessing`.

Junto con la clase `QueueListener` la clase `QueueHandler` puede usarse para permitir a los gestores realizar su tarea en un hilo separado del que realiza el *logging*. Esto es importante en aplicaciones Web y también otras aplicaciones donde los clientes servidos por los hilos necesitan responder tan rápido como sea posible, mientras que cualquier operación potencialmente lenta (tal como enviar un correo electrónico a través la clase `SMTPHandler`) se realizan por un hilo diferente.

class `logging.handlers.QueueHandler` (*queue*)

Retorna una nueva instancia de la clase `QueueHandler`. La instancia se inicializa con la cola a la que se enviarán los mensajes. La cola puede ser cualquier objeto tipo-cola; es usado tal como por el método `enqueue()` que necesita saber como enviar los mensajes a ella. La cola no es *requerida* para tener una API de rastreo de tareas, lo que significa que puedes usar instancias de `SimpleQueue` como *queue*.

emit (*record*)

Pone en la cola el resultado de preparar el registro historial de log. Si ocurre una excepción (por ejemplo por que una cola de destino se llenó) entonces se llama al método `handleError()`. Esto puede resultar en que el registro se descarte (si `logging.raiseExceptions` es `False`) o que se imprima un mensaje a `sys.stderr` (si `logging.raiseExceptions` es `True`).

prepare (*record*)

Prepara un registro para poner en la cola. El objeto que retorna este método se colocará en cola.

La implementación base da formato al registro para unir la información de los mensajes, argumentos y excepciones, si están presentes. También remueve los elementos que no se pueden serializar (*unpickables*) de los registros in-situ.

Puedes querer hacer caso omiso de este método si quieres convertir el registro en un diccionario o cadena de caracteres JSON, o enviar una copia modificada del registro mientras dejas el original intacto.

enqueue (*record*)

Coloca en la cola al registro usando `put_nowait()`; puede que quieras sobrescribe (*override*) esto si quieres usar una acción de bloqueo, o un tiempo de espera, o una implementación de cola a medida.

16.8.16 QueueListener

Nuevo en la versión 3.2.

La clase `QueueListener` esta localizada en el módulo `logging.handlers`. Soporta la recepción de mensajes *logging* de una cola, tal como los implementados en los módulos `queue` or `multiprocessing`. Los mensajes son recibidos de una cola en un hilo interno y se pasan en el mismo hilo, a uno o mas gestores para procesarlos. Mientras la clase `QueueListener` no es en si misma un gestor, esta documentada aquí porque trabaja mano a mano con la clase `QueueHandler`.

Junto con la clase `QueueHandler`, la clase `QueueListener` puede usarse para permitir a los gestores hacer su labor en un hilo separado del que hace el *logging*. Esto es importante en aplicaciones Web y también otras aplicaciones de servicio donde los clientes servidos por los hilos necesitan una respuesta tan rápida como sea posible, mientras cualquier operación potencialmente lenta (tal como enviar un correo electrónico a través de la clase `SMTPHandler`) son atendidas por un hilo diferente.

class `logging.handlers.QueueListener` (*queue*, **handlers*, *respect_handler_level=False*)

Retorna una nueva instancia de la clase `QueueListener`. La instancia se inicializa con la cola para enviar mensajes a una lista de gestores que manejarán entradas colocadas en la cola. La cola puede ser cualquier objeto de tipo-cola, es usado tal como está por el método `dequeue()` que necesita saber como tomar los mensajes de esta. La cola no es *obligatoria* para tener la API de seguimiento de tareas (aunque se usa si está disponible), lo que significa que puede usar instancias `SimpleQueue` para *queue*.

Si `respect_handler_level` es `True`, se respeta un nivel de gestor (comparado con el nivel del mensaje) cuando decide si pasar el mensajes al gestor; de otra manera, el comportamiento es como en versiones anteriores de Python - de pasar cada mensaje a cada gestor.

Distinto en la versión 3.5: Se agregó el argumento `respect_handler_levels`.

dequeue (*block*)

Extrae de la cola un registro y lo retorna, con opción a bloquearlo.

La implementación base usa `get()`. Puedes sobrescribir (*override*) este método si quieres usar tiempos de espera o trabajar con colas implementadas a medida.

prepare (*record*)

Prepara un registro para ser gestionado.

Esta implementación solo retorna el registro que fue pasado. Puedes sobrescribir (*override*) este método para hacer una serialización a medida o una manipulación del registro antes de pasarlo a los gestores.

handle (*record*)

Manejar un registro.

Esto solo realiza un bucle a través de los gestores ofreciéndoles el registro para ser gestionado. El objeto actual pasado a los gestores es aquel que es retornado por el método `prepare()`.

start ()

Da comienzo al oyente (*listener*).

Esto da comienzo a un hilo en segundo plano para supervisar la cola de registros log a procesar.

stop ()

Detiene el oyente (*listener*).

Esto solicita terminar al hilo, y luego espera hasta que termine. Nota que si no llamas a esto antes de que tu aplicación salga, puede haber algunos registros que aun están en la cola, que no serán procesados.

enqueue_sentinel ()

Escribe un centinela (*sentinel*) en la cola para decir al oyente (*listener*) de salir. Esta implementación usa `put_nowait()`. Puedes sobrescribir (*override*) este método si quieres usar tiempos de espera o trabajar con implementaciones de cola a tu medida.

Nuevo en la versión 3.3.

Ver también:

Módulo `logging` Referencia API para el módulo de *logging*.

Módulo `logging.config` Configuración API para el módulo de *logging*.

16.9 getpass — Entrada de contraseña portátil

Código fuente: [Lib/getpass.py](#)

El módulo `getpass` proporciona dos funciones:

`getpass.getpass` (*prompt*='Password: ', *stream*=None)

Solicita al usuario una contraseña sin hacer eco. Se solicita al usuario mediante la cadena *prompt*, que por defecto es 'Password: '. En Unix, el indicador se escribe en el objeto similar a un archivo *stream* usando el controlador de errores de reemplazo si es necesario. *stream* toma por defecto el terminal de control (`/dev/tty`) o si no está disponible para `sys.stderr` (este argumento se ignora en Windows).

Si la entrada sin *echo* no está disponible, `getpass()` recurre a imprimir un mensaje de advertencia en *stream* y leer de `sys.stdin` y lanza un `GetPassWarning`.

Nota: Si llama a `getpass` desde IDLE, la entrada puede realizarse en la terminal desde la que inició IDLE en lugar de en la ventana inactiva en sí.

exception `getpass.GetPassWarning`

Una subclase `UserWarning` lanzada cuando la entrada de la contraseña puede repetirse.

`getpass.getuser()`

Retorna el «nombre de inicio de sesión» del usuario.

Esta función verifica las variables de entorno `LOGNAME`, `USER`, `LNAME` and `USERNAME`, en orden, y retorna el valor del primero que se establece en un cadena no vacía. Si no se establece ninguno, el nombre de inicio de sesión de la base de datos de contraseñas se retorna en los sistemas que admiten el módulo `pwd`; de lo contrario, se lanza una excepción.

En general, esta función debería preferirse respecto a `os.getlogin()`.

16.10 `curses` — Manejo de terminales para pantallas de celdas de caracteres

El módulo `curses` provee una interfaz para la librería `curses`, el estándar para el manejo avanzado de terminales portátiles.

Mientras que `curses` es más ampliamente usado en los ambientes Unix, las versiones están disponibles para Windows, DOS, y posiblemente para otros sistemas también. Esta extensión del módulo es diseñada para coincidir con el API de `ncurses`, una librería de código abierto almacenada en Linux y las variantes BSD de Unix.

Nota: Cuando la documentación menciona un *carácter*, esto puede ser especificado como un entero, una cadena Unicode de un carácter o una cadena de bytes de un byte.

Cuando la documentación menciona una *cadena de caracteres*, esto puede ser especificado como una cadena Unicode o una cadena de bytes.

Nota: Desde la versión 5.4, la librería `ncurses` decide cómo interpretar los datos no ASCII usando la función `nl_langinfo`. Eso significa que tú tienes que llamar a `locale.setlocale()` en la aplicación y codificar las cadenas Unicode usando una de las codificaciones disponibles del sistema. Este ejemplo usa la codificación del sistema por defecto:

```
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

Entonces usa `code` como la codificación para las llamadas `str.encode()`.

Ver también:

Módulo `curses.ascii` Utilidades para trabajar con caracteres ASCII, independientemente de tu configuración local.

Módulo `curses.panel` Una extensión de la pila de paneles que añade profundidad a las ventanas de `curses`.

Módulo `curses.textpad` Widget de texto editable para apoyar `curses` **Emacs**- como enlaces.

curses-howto Material del tutorial usando `curses` con Python, por *Andrew Kuchling* y *Eric Raymond*.

El directorio `Tools/demo/` en el recurso de distribución de Python contiene algunos programas de ejemplo usando los enlaces de `curses` previstos en este módulo.

16.10.1 Funciones

El módulo `curses` define la siguiente excepción:

exception `curses.error`

Una excepción se lanza cuando una función de la librería `curses` retorna un error.

Nota: Cuando los argumentos *x* o *y* para una función o un método son opcionales, se predetermina la ubicación actual del cursor. Cuando *attr* es opcional, por defecto es `A_NORMAL`.

El módulo `curses` define las siguientes funciones:

`curses.baudrate()`

Retorna la velocidad de salida de la terminal en bits por segundo. En emuladores de terminal de software esto tendrá un alto valor fijado. Incluido por razones históricas; en tiempos pasados, esto fue usado para escribir los ciclos de salida por retrasos de tiempo y ocasionalmente para cambiar interfaces dependiendo de la velocidad en la línea.

`curses.beep()`

Emite un corto sonido de atención.

`curses.can_change_color()`

Retorna «True» o «False», dependiendo ya sea que el programador puede cambiar los colores presentados por la terminal.

`curses.cbreak()`

Entra al modo *cbreak*. En el modo *cbreak* (a veces llamado modo «raro») del buffer de línea *tty* normal es apagado y los caracteres están disponibles para ser leídos uno por uno. Sin embargo, a diferencia del modo *raw*, los caracteres especiales (interrumpir, salir, suspender y control de flujo) retiene sus efectos en el manejador *tty* y el programa de llamada. Llamando primero `raw()` luego `cbreak()` dejando la terminal en modo *cbreak*.

`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 - 1`. Return a 3-tuple, 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(pair_number)`

Return the attribute value for displaying text in the specified color pair. Only the first 256 color pairs are supported. 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)`

Configura el estado del cursor. *visibilidad* puede estar configurado para «0», «1» o «2», para invisible, normal o muy visible. Si la terminal soporta la visibilidad requerida, retorna el estado del cursor previo; de otra manera ejecuta una excepción. En muchos terminales, el modo «visible» es un cursor subrayado y el modo «muy visible» es un cursor de bloque.

`curses.def_prog_mode()`

Guardar el modo del terminal actual como el modo «program», el modo cuando el programa corriendo está usando `curses`. (Su contraparte es el modo «shell», para cuando el programa no está en `curses`.) Seguido de la llamada a `reset_prog_mode()` restaurará este modo.

`curses.def_shell_mode()`

Guarde el modo de terminal actual como el modo «shell», el modo en el que el programa en ejecución no utiliza `curses`. (Su contraparte es el modo «program», cuando el programa está usando las capacidades de `curses`.) Las llamadas subsecuentes a `reset_shell_mode()` restaurarán este modo.

`curses.delay_output(ms)`

Inserte una pausa en milisegundo *ms* en la salida.

`curses.doupdate()`

Actualiza la pantalla física. La librería `curses` mantiene dos estructuras de datos, uno representa el contenido

de la pantalla física actual y una pantalla virtual representa el próximo estado deseado. La base `doupdate()` actualiza la pantalla física para comparar la pantalla virtual.

La pantalla virtual puede ser actualizada por una llamada `noutrefresh()` después de escribir las operaciones tales como `addstr()` ha sido ejecutada en una ventana. La llamada normal `refresh()` es simplemente `noutrefresh()` seguido por `doupdate()`; si tienes para actualizar múltiples ventanas, puedes aumentar el rendimiento y quizás reducir los parpadeos de la pantalla usando la llamada `noutrefresh()` en todas las ventanas, seguido por un simple `doupdate()`.

`curses.echo()`

Entrar en modo echo. En modo echo, cada caracter de entrada es repercutido a la pantalla como este es introducido.

`curses.endwin()`

Desinicializa la librería y retorne el terminal al estado normal.

`curses.erasechar()`

Retorne el carácter borrado del usuario actual como un objeto de bytes de un byte. Bajo el sistema de operaciones Unix esta es una propiedad de el controlador tty de el programa curses, y no es configurado por la librería curses en sí misma.

`curses.filter()`

La rutina `filter()`, si es usada, debe ser llamada antes que `initscr()` sea llamada. El efecto es que durante estas llamadas, `LINES` es configurada para «1»; las capacidades «clear», «cup», «cud», «cud1», «cuu1», «cuu», «vpa» son desactivadas; y la cadena «home» es configurada para el valor de «cr». El efecto es que el cursor es confinado para la línea actual, y también las pantallas son actualizadas. Este puede ser usado para habilitar la línea editando el carácter en un tiempo sin tocar el resto de las pantallas.

`curses.flash()`

La pantalla Flash. Eso es, cambiar lo a video inverso y luego cambie lo de nuevo en un corto intervalo. Algunas personas prefieren como “campana visible” para la señal de atención audible producida por `beep()`.

`curses.flushinp()`

Vacíe todos los búferes de entrada. Esto desecha cualquier mecanografiado que ha sido escrito por el usuario y no ha sido aún procesado por el programa.

`curses.getmouse()`

After `getch()` returns `KEY_MOUSE` to signal a mouse event, this method should be called 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()`

Retorna las coordenadas actuales del cursor en la pantalla virtual como una tupla (`y`, `x`). Si `leaveok` es actualmente `True` entonces retorna `(-1, -1)`.

`curses.getwin(file)`

Lee la ventana relacionada con los datos almacenados en el archivo por una llamada temprana a `putwin()`. La rutina entonces crea e inicializa una nueva ventana usando esos datos, retornando el nuevo objeto de ventana.

`curses.has_colors()`

Retorna «True» si el terminal puede desplegar colores, en caso contrario, retorna «False».

`curses.has_ic()`

Retorna `True` si el terminal tiene la capacidad de insertar y eliminar caracteres. Esta función es incluida por razones históricas solamente, ya que todos los emuladores de la terminal de software modernos tienen tales capacidades.

`curses.has_il()`

Retorna `True` si la terminal tiene la capacidad de insertar y eliminar caracteres o pueden simularlos usando las regiones de desplazamiento. Esta función es incluida por razones históricas solamente, como todos los emuladores de terminales de software modernos tienen tales capacidades.

`curses.has_key(ch)`

Toma una clave valor *ch*, y retorna «True» si el tipo de terminal actual reconoce una clave con ese valor.

`curses.halfdelay(tenths)`

Usado por el modo de medio retardo, el cual es similar al modo *cbreak* en que los caracteres escritos por el usuario están disponibles inmediatamente para el programa. Sin embargo, después de bloquearlos por *tenths* décimas de segundos, se levanta una excepción si nada ha sido escrito. El valor de *tenths* debe ser un número entre 1 y 255. Use *nocbreak()* para salir del modo de medio retardo.

`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* - 1. 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 True.

`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* - 1, or, after calling *use_default_colors()*, -1. 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()`

Inicializa la librería. Retorna un objeto *window* el cual representa a toda la pantalla.

Nota: Si hay un error al abrir el terminal, la librería *curses* subyacente puede causar que el interprete salga.

`curses.is_term_resized(nlines, ncols)`

Retorna «True» si *resize_term()* modificaría la estructura de la ventana, «False» en caso contrario.

`curses.isendwin()`

Retorna «True» si *endwin()* ha sido llamado (eso es que la librería *curses* ha sido desinicializada).

`curses.keyname(k)`

Retorna el nombre de la tecla enumerada *k* como un objeto de bytes. El nombre de una tecla que genera un carácter ASCII imprimible es el carácter de la tecla. El nombre de una combinación de teclas de control es un consistente objeto de bytes de dos bytes de un signo de intercalación (*b'^'*) seguido por el correspondiente carácter ASCII imprimible. El nombre de una combinación de tecla *alt* (128-255) es un objeto de bytes consistente del prefijo *b'M-'* seguido por el nombre del correspondiente carácter ASCII.

`curses.killchar()`

Retorna el carácter de eliminación de la línea actual del usuario como un objeto de bytes de un byte. Bajo el sistema operativo Unix esta es una propiedad del controlador *tty* del programa *curses*, y no está configurado por la librería *curses* por sí mismo.

`curses.longname()`

Retorna un objeto de bytes que contiene el campo de nombre largo *terminfo* que describe el terminal actual. La longitud máxima de una descripción verbosa es 128 caracteres. Esto es definido solamente después de la llamada a *initscr()*.

`curses.meta(flag)`

Si *flag* es «True», permite caracteres de 8 bits para ser introducidos. Si *flag* es «False», permite solamente caracteres de 7 bits.

`curses.mouseinterval(interval)`

Configura el tiempo máximo en milisegundos que pueden transcurrir entre los eventos de presionar y soltar para que se reconozcan como un click, y retornen el valor del intervalo anterior. El valor por defecto es 200 msec, o una quinta parte de un segundo.

`curses.mousemask(mousemask)`

Configure los eventos del mouse para ser reportados, y retorna una tupla (*availmask*, *oldmask*). *avail-*

mask indica cual de los eventos del ratón especificados pueden ser reportados; en caso de falla completa retorna 0. *oldmask* es el valor previo de la máscara de evento del mouse de la ventana dada. Si esta función nunca es llamada, los eventos del mouse nunca son reportados.

`curses.napms(ms)`

Duerme durante *ms* milisegundos.

`curses.newpad(nlines, ncols)`

Crea y retorna un apuntador para una nueva estructura de datos de *pad* con el número dado de líneas y columnas. Retorna un *pad* como un objeto de ventana.

Una almohadilla es como una ventana, excepto que no es restringida por el tamaño de la pantalla, y no está necesariamente asociada con una parte particular de la pantalla. Las almohadillas pueden ser usadas cuando un ventana grande es necesitada, y solamente una parte de la ventana estará en la pantalla de una sola vez. Actualizaciones automáticas de almohadillas (tales desde el desplazamiento o haciendo eco de la entrada) no ocurre. Los métodos `refresh()` y `noutrefresh()` de una almohadilla requiere 6 argumentos para especificar la parte de la almohadilla a ser mostrada y la locación en la ventana que se utilizará para la visualización. Los argumentos son *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*; el argumento *p* se refiere a la esquina superior izquierda de la región de la almohadilla a ser mostrada y el argumento *s* define una caja de recorte en la pantalla con la cual la región de la almohadilla será mostrada.

`curses.newwin(nlines, ncols)`

`curses.newwin(nlines, ncols, begin_y, begin_x)`

Retorna una nueva *window*, cuya esquina superior izquierda esta en (*begin_y*, *begin_x*), y cuyo alto/ancho es *nlines/ncols*.

Por defecto, la ventana extenderá desde la posición especificada para la esquina inferior derecha de la pantalla.

`curses.nl()`

Ingrese al modo de línea nueva. Este modo pone la tecla de retorno en una nueva línea en la entrada, y traduce la nueva línea en retorno y avance de línea en la salida. El modo de nueva línea está inicialmente encendida.

`curses.nocbreak()`

Salir del modo *cbreak*. Retorne al modo normal «cooked» con la línea del búfer.

`curses.noecho()`

Salir del modo echo. El eco de los caracteres de entrada está desactivado.

`curses.nonl()`

Dejar el modo de nueva línea. Desactiva la traducción de retorno en una nueva línea en la entrada, y desactiva la traducción a bajo nivel de una nueva línea en nueva línea/retorno en la salida (pero esto no cambia el comportamiento de `addch('\n')`, el cual siempre es el equivalente de retornar y avanzar la línea en la pantalla virtual). Con las traducciones apagadas, *curses* puede aumentar algunas veces la velocidad del movimiento vertical un poco; también, estará disponible para detectar la tecla de retorno en la entrada.

`curses.noqiflush()`

Cuando la rutina `noqiflush()` es usada, descarga normal de colas de entrada y salida asociadas con el INTR, los caracteres QUIT and SUSP no serán hechos. Puedes querer llamar `noqiflush()` en un manejador de señales si quieres que la salida continúe como si la interrupción no hubiera ocurrido después de que el manejador exista.

`curses.noraw()`

Salir del modo raw. Retorna al modo normal «cooked» con la línea del búfer.

`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 0 and `COLOR_PAIRS - 1`.

`curses.pair_number(attr)`

Retorna el numero del conjunto de pares de colores para el valor del atributo *attr*. `color_pair()` es la contraparte de esta función.

`curses.putp(str)`

Equivalente a `tputs(str, 1, putchar)`; emite el valor de una capacidad especificada *terminfo* para el terminal actual. Nota que la salida de `putp()` siempre va a la salida estándar.

`curses.qiflush([flag])`

Si *flag* es `False`, el efecto es el mismo como llamar `noqiflush()`. Si *flag* es `True`, o el argumento no es proveído, la cola será nivelada cuando estos caracteres de control son leídos.

`curses.raw()`

Entrar al modo crudo. En el modo crudo, el almacenamiento en búfer de la línea normal y procesamiento de las teclas de interrupción, salida, suspensión y control de flujo son apagadas; los caracteres son presentados a la función de entrada de *curses* una por una.

`curses.reset_prog_mode()`

Restaura la terminal para el modo «program», anteriormente guardado por `def_prog_mode()`.

`curses.reset_shell_mode()`

Restablece el terminal al modo «shell» como lo guardó previamente `def_shell_mode()`.

`curses.resetty()`

Restablece el estado del modo del terminal para lo que esto fue en el último llamado a `savetty()`.

`curses.resize_term(nlines, ncols)`

La función *backend* usada por `resizeterm()`, caracteriza más de el trabajo; cuando se redimensiona la ventana, `resize_term()` los rellenos blancos de las áreas que son extendidas. La aplicación de llamada llenaría en estas áreas con datos apropiados. La función `resize_term()` intenta redimensionar todas las ventanas. Sin embargo, debido a la convención de llamadas del las almohadillas, esto no es posible para redimensionar estos sin interacciones adicionales con la aplicación.

`curses.resizeterm(nlines, ncols)`

Ajusta el tamaño al estándar y la ventana actual para las dimensiones especificadas, y ajusta otros datos contables usados por la librería *curses* que registra las dimensiones de la ventana (en particular el manejador `SIGWINCH`).

`curses.savetty()`

Guarda el estado actual del modo del terminal en un búfer, usable por `resetty()`.

`curses.setsyx(y, x)`

Fija el cursor de la pantalla virtual para *y*, *x*. Si *y* y *x* son ambos «-1», entonces `leaveok` es configurado «True».

`curses.setupterm(term=None, fd=-1)`

Inicializa la terminal. *term* es una cadena de caracteres dando el nombre de la terminal, o `None`; si es omitido o `None`, el valor de la variable de entorno `TERM` será usada. *fd* es el archivo descriptor al cual alguna secuencia de inicialización será enviada; si no es suministrada o `-1`, el archivo descriptor para `sys.stdout` será usado.

`curses.start_color()`

Debe ser llamado si el programador quiere usar colores, y antes de que cualquier otra rutina de manipulación de colores sea llamada. Esta es una buena práctica para llamar esta rutina inmediatamente después de `initscr()`.

`start_color()` inicializa ocho colores básicos (negro, rojo, verde, amarillo, azul, magenta, cian, y blanco), y dos variables globales en el módulo *curses*, `COLORS` y `COLOR_PAIRS`, contiene el número máximo de colores y los pares de colores que la terminal puede soportar. Esto también restaura los colores en la terminal para los valores que ellos tienen cuando la terminal fue solo activada.

`curses.termattrs()`

Retorna un *OR* lógico de todos los atributos del video soportados por el terminal. Esta información es útil cuando un programa *curses* necesita completar el control sobre la apariencia de la pantalla.

`curses.termname()`

Retorna el valor de la variable de entorno `TERM`, como un objeto de bytes, trucado para 14 caracteres.

`curses.tigetflag(capname)`

Retorna el valor de la capacidad booleana correspondiente al nombre de la capacidad *terminfo* *capname* como un número entero. Retorna el valor `-1` si *capname* no es una capacidad booleana, o `0` si es cancelada o ausente desde la descripción de la terminal.

`curses.tigetnum(capname)`

Retorna el valor de la capacidad numérica correspondiente al nombre de la capacidad *terminfo* *capname* como

un entero. Regresa el valor `-2` si *capname* no es una capacidad numérica, o `-1` si esta es cancelada o ausente desde la descripción del terminal.

`curses.tigetstr(capname)`

Retorna el valor de la capacidad de la cadena de caracteres correspondiente al nombre de la capacidad *terminfo* *capname* como un objeto de bytes. Retorna `None` si *capname* no es una «capacidad de cadena de caracteres» de *terminfo*, o es cancelada o ausente desde la descripción de la terminal.

`curses.tparm(str[, ...])`

Instancia del objeto de bytes *str* con los parámetros suministrados, donde *str* sería un cadena de caracteres parametrizada obtenida desde la base de datos de *terminfo*. E.g. `tparm(tigetstr("cup"), 5, 3)` podría resultar en `b'\033[6;4H'`, el resultado exacto depende del tipo de terminal.

`curses.typeahead(fd)`

Especifica que el descriptor de archivo *fd* es usado para la verificación anticipada. Si *fd* es `<-1`, entonces no se realiza ninguna verificación anticipada.

La librería *curses* hace «la optimización del rompimiento de línea» por la búsqueda para mecanografiar periódicamente mientras se actualiza la pantalla. Si la entrada es encontrada, y viene desde un *tty*, la actualización actual es pospuesta hasta que se vuelva a llamar la actualización, permitiendo la respuesta más rápida para comandos escritos en avance. Esta función permite especificar un archivo diferente al descriptor para la verificación anticipada.

`curses.unctrl(ch)`

Regresa un objeto de bytes el cual es una representación imprimible del carácter *ch*. Los caracteres de control son representados como un símbolo de intercalación seguido del carácter, por ejemplo como `b'^C'`. La impresión de caracteres son dejados como están.

`curses.ungetch(ch)`

Presiona *ch* para que el siguiente `getch()` lo retorne.

Nota: Solamente un *ch* puede ser colocado antes `getch()` es llamada.

`curses.update_lines_cols()`

Actualiza `LINES` y `COLS`. Útil para detectar el cambio manual del tamaño de pantalla.

Nuevo en la versión 3.5.

`curses.unget_wch(ch)`

Presiona *ch* para que el siguiente `get_wch()` lo retorne.

Nota: Solamente un *ch* puede ser presionado antes `get_wch()` es llamada.

Nuevo en la versión 3.3.

`curses.ungetmouse(id, x, y, z, bstate)`

Coloca un evento `KEY_MOUSE` en la cola de entrada, asociando el estado de los datos dados con esto.

`curses.use_env(flag)`

Si es usado, esta función sería llamada antes de `initscr()` o `newterm` son llamados. Cuando *flag* es `False`, los valores de líneas y columnas especificadas en la base de datos *terminfo* será usado, incluso si las variables de entorno `LINES` y `COLUMNS` (usadas por defecto) son configuradas, o si *curses* está corriendo en una ventana (en cuyo caso el comportamiento por defecto sería usar el tamaño de ventana si `LINES` y `COLUMNS` no están configuradas).

`curses.use_default_colors()`

Permite usar valores por defecto para los colores en terminales apoyando esta característica. Use esto para apoyar la transparencia en tu aplicación. El color por defecto es asignada para el número de color `-1`. Después de llamar esta función, inicializa `init_pair(x, curses.COLOR_RED, -1)`, por ejemplo, los pares de color *x* a un primer plano de color rojo en el fondo por defecto.

`curses.wrapper(func, ...)`

Inicializa *curses* y llama a otro objeto invocable, *func*, el cual debería ser el resto de tu aplicación que usa *curses*. Si la aplicación lanza una excepción, esta función restaurará la terminal para un estado sano antes de re-lanzar la excepción y generar una pista. El objeto invocable *func* es entonces pasado a la ventana principal “stdscr” como su primer argumento, seguido por algún otro argumento pasado a `wrapper()`. Antes de llamar a *func*, `wrapper()` habilita el modo *cbreak*, desactiva echo, permite el teclado del terminal, e inicializa los colores si la terminal tiene soporte de color. En salida (ya sea normal o por excepción) esto restaura el modo *cooked*, habilita el echo, y desactiva el teclado del terminal.

16.10.2 Objetos de ventana

Los objetos ventana, retornados por `initscr()` y `newwin()` anteriores, tienen los siguientes métodos y atributos:

`window.addch(ch[, attr])`

`window.addch(y, x, ch[, attr])`

Paint character *ch* at (*y*, *x*) with attributes *attr*, overwriting any character previously painted at that location. By default, the character position and attributes are the current settings for the window object.

Nota: Escribiendo afuera de la ventana, sub-ventana, o *pad* genera un `curses.error`. Intentando escribir en la esquina inferior derecha de una ventana, sub-ventana, o *pad* causará una excepción a ser generada después de que el carácter es pintado.

`window.addnstr(str, n[, attr])`

`window.addnstr(y, x, str, n[, attr])`

Pintar como máximo *n* caracteres de la cadena de texto *str* en «(y,x)» con atributos *attr*, sobrescribiendo algo previamente en la pantalla.

`window.addstr(str[, attr])`

`window.addstr(y, x, str[, attr])`

Dibuja la cadena de caracteres *str* en «(y,x)» con atributos *attr*, sobrescribiendo cualquier cosa previamente en la pantalla.

Nota:

- Escribiendo afuera de la ventana, sub-ventana, o *pad* genera un `curses.error`. Intentando escribir en la esquina inferior derecha de una ventana, sub-ventana, o *pad* causará una excepción a ser generada después de que la cadena de caracteres es pintada.
 - Un *bug* en **ncurses**, el *backend* para este módulo de Python, puede causar *SegFaults* cuando re-dimensionan las ventanas. Esto es reparado en *ncurses-6.1-20190511*. Si tu estás atascado con un *ncurses* anterior, puedes evitar desencadenar este si no llamas `addstr()` con un *str* que tiene embebido nuevas líneas. En su lugar, llama `addstr()` separadamente por cada línea.
-

`window.attroff(attr)`

Remueve el atributo *attr* desde el conjunto «background» aplicado a todos los escritos para la ventana actual.

`window.attron(attr)`

Añade el atributo *attr* del conjunto del *background* aplicado para todas las escrituras de la ventana actual.

`window.attrset(attr)`

Establezca el conjunto de atributos «background para *attr*. Este conjunto es inicialmente «0» (sin atributos).

`window.bkgd(ch[, attr])`

Configura la propiedad de fondo de la ventana para el carácter *ch*, con atributos *atte*. El cambio es entonces aplicado para la posición de cada carácter en esa ventana:

- El atributo de cada carácter en la ventana es cambiado por el nuevo atributo de fondo.
- Dónde quiera que el carácter del fondo anterior aparezca, es cambiado al nuevo carácter de fondo.

`window.bkgdset (ch[, attr])`

Configura el fondo de la ventana. Un fondo de ventana consiste de un carácter y cualquier combinación de atributos. La parte del atributo del fondo es combinado (OR" ed) con todos los caracteres que no son blancos escritos en la ventana. Tanto las partes del carácter y del atributo del fondo son combinados con los caracteres en blanco. El fondo se convierte en una propiedad del carácter y se mueve con el carácter a través de cualquier operación de desplazamiento e inserción/eliminación de línea/carácter.

`window.border ([ls[, rs[, ts[, bs[, tl[, tr[, bl[, br]]]]]]])`

Dibuja un borde alrededor de las orillas de la ventana. Cada parámetro especifica el carácter a usar para una parte específica del borde; ve la tabla abajo para más detalles.

Nota: Un valor 0 para cualquier parámetro causará el carácter por defecto a ser usado para ese parámetro. parámetros de palabras clave pueden *no* ser usados. Los valores predeterminados son listados en esta tabla:

Parámetro	Descripción	Valor por defecto
<i>ls</i>	Lado izquierdo	ACS_VLINE
<i>rs</i>	Lado derecho	ACS_VLINE
<i>ts</i>	Arriba	ACS_HLINE
<i>bs</i>	Abajo	ACS_HLINE
<i>tl</i>	Esquina superior izquierda	ACS_ULCORNER
<i>tr</i>	Esquina superior derecha	ACS_URCORNER
<i>bl</i>	Esquina inferior izquierda	ACS_LLCORNER
<i>br</i>	Esquina inferior derecha	ACS_LRCORNER

`window.box ([vertch, horch])`

Similar a `border()`, pero ambos *ls* y **rs* son *vertch* y ambos *ts* y *bs* son *horch*. Los caracteres de esquina predeterminados son siempre usados por esta función.

`window.chgat (attr)`

`window.chgat (num, attr)`

`window.chgat (y, x, attr)`

`window.chgat (y, x, num, attr)`

Configura los atributos de *num* de caracteres en la posición del cursor, o una posición (*y*, *x*) si es suministrado. Si *num* no es dado o es -1, el atributo será configurado en todos los caracteres para el final de la línea. Esta función mueve el cursor a la posición (*y*, *x*) si es suministrado. La línea cambiada será tocada usando el método `touchline()` tal que el contenido será vuelto a mostrar por la actualización de la siguiente ventana.

`window.clear()`

Semejante a `erase()`, pero también causa que toda la ventana sea repintada sobre la siguiente llamada para `refresh()`.

`window.clearok (flag)`

Si *flag* es «True», la siguiente llamada para `refresh()` limpiará la ventana completamente.

`window.clrtoeol()`

Borrar desde el cursor hasta el final de la ventana: todas las líneas bajo el cursor son eliminadas, y entonces el equivalente de `clrtoeol()` es realizado.

`window.clrtoeol()`

Borrar desde el cursor hasta el final de la línea.

`window.cursyncup()`

Actualice la posición actual del cursor de todos los ancestros de la ventana para reflejar la posición actual del cursor de la ventana.

`window.delch ([y, x])`

Borrar cualquier carácter en «(y,x)».

`window.deleteln()`

Borra la línea bajo el cursor. Todas las líneas siguientes son movidas una línea hacia arriba.

`window.derwin(begin_y, begin_x)`

`window.derwin(nlines, ncols, begin_y, begin_x)`

Una abreviación para «ventana derivada», `derwin()` es el mismo cómo llamar `subwin()`, excepto que `begin_y` y `begin_x` son relativos al origen de la ventana, más bien relativo a la pantalla completa. Retorna un objeto ventana para la ventana derivada.

`window.echochar(ch[, attr])`

Añada el carácter `ch` con atributo `attr`, e inmediatamente llame a `refresh()` en la ventana.

`window.enclose(y, x)`

Pruebe si el par dado de las coordenadas de celda del carácter relativas de la ventana son encerrados por la ventana dada, retornando `True` o `False`. Esto es útil para determinar que subconjunto de las ventanas de pantalla encierran la locación de un evento del mouse.

`window.encoding`

La codificación utilizada para codificar argumentos de método (*Unicode* de caracteres y cadena de caracteres). El atributo codificado es heredado desde la ventana padre cuando una sub-ventana es creada, por ejemplo con `window.subwin()`. Por defecto, la codificación local es usada (ver `locale.getpreferredencoding()`).

Nuevo en la versión 3.3.

`window.erase()`

Limpiar la ventana.

`window.getbegyx()`

Retorna una tupla «(y,x)» de coordenadas de la esquina superior izquierda.

`window.getbkgd()`

Retorna el par carácter/atributo dada la ventana actual.

`window.getch([y, x])`

Obtener un carácter. Nota que el entero retornado *no* tiene que estar en el rango ASCII: teclas de función, claves de teclado y los demás son representados por números mayores que 255. En el modo sin demora, retorna `-1` si no hay entrada, en caso contrario espere hasta que una tecla es presionada.

`window.get_wch([y, x])`

Obtener un carácter amplio. Retorna un carácter para la mayoría de las teclas, o un entero para las teclas de función, teclas del teclado, y otras teclas especiales. En modo de no retorna, genera una excepción si no hay entrada.

Nuevo en la versión 3.3.

`window.getkey([y, x])`

Obtener un carácter, retornando una cadena de caracteres en lugar de un entero, como `getch()` lo hace. Las teclas de función, teclas del teclado y otras teclas especiales retornan una cadena *multibyte* conteniendo el nombre de la tecla. En modo de no retardo, genera una excepción si no hay entrada.

`window.getmaxyx()`

Retorna una tupla «(y,x)» de el alto y ancho de la ventana.

`window.getparyx()`

Retorne las coordenadas iniciales de esta ventana relativa para su ventana padre como una tupla `(y, x)`. Retorna `(-1, -1)` si esta ventana no tiene padre.

`window.getstr()`

`window.getstr(n)`

`window.getstr(y, x)`

`window.getstr(y, x, n)`

Lee un objeto de bytes desde el usuario, con capacidad de edición de línea primitiva.

`window.getyx()`
 Retorna una tupla (*y*, *x*) de la posición actual del cursor relativa para la esquina superior izquierda de la ventana.

`window.hline(ch, n)`
`window.hline(y, x, ch, n)`
 Muestra una línea horizontal iniciando en (*y*, *x*) con longitud *n* consistente de el carácter *ch*.

`window.idcok(flag)`
 Si *flag* es `False`, *curses* ya no considera el uso de la función de insertar/eliminar caracteres de hardware del terminal; si *flag* `True`, el uso de inserción y borrado de caracteres está habilitado. Cuando *curses* es inicializado primero, el uso de caracteres de inserción / borrado está habilitado por defecto.

`window.idlok(flag)`
 Si *flag* es `True`, *curses* tratará y usará las funciones de la línea de edición de hardware. En caso contrario, la línea inserción/borrado están deshabilitadas.

`window.immedok(flag)`
 Si *flag* es `True`, cualquier cambio en la imagen de la ventana automáticamente causará que la ventana sea refrescada; ya no tienes que llamar a `refresh()` por ti mismo. Sin embargo, esto puede degradar el rendimiento considerablemente, debido a las repeticiones de llamada a `wrefresh`. Esta opción está deshabilitada por defecto.

`window.inch([y, x])`
 Retorna el carácter en la posición dada en la ventana. Los 8bits inferiores son los caracteres propiamente dichos, y los bits superiores son los atributos.

`window.insch(ch[, attr])`
`window.insch(y, x, ch[, attr])`
 Pinta el carácter *ch* en (*y*, *x*) con atributos *attr*, moviendo la línea desde la posición *x* a la derecha un carácter.

`window.insdelln(nlines)`
 Inserta *nlines* líneas en la ventana especificada arriba de la línea actual. Las *nlines* líneas de fondo se pierden. Para *nlines* negativos, elimine *nlines* líneas comenzando con la que está debajo del cursor, y mueve las restantes hacia arriba. Se borran las *nlines* líneas inferiores. La posición del cursor actual se mantiene igual.

`window.insertln()`
 Inserte una línea en blanco bajo el cursos. Las líneas siguientes se moverán una línea hacia abajo.

`window.insnstr(str, n[, attr])`
`window.insnstr(y, x, str, n[, attr])`
 Inserte una cadena de caracteres (tantos caracteres quepan en la línea) antes los caracteres bajo el cursor, sobre los *n* caracteres. Si *n* es cero o negativo, la cadena entera es insertada. Todos los caracteres a la derecha de el cursor son desplazados a la derecha, perdiéndose los caracteres de la derecha en la línea. La posición del cursor no cambia (después de mover a *y*, *x*, si es especificado).

`window.insstr(str[, attr])`
`window.insstr(y, x, str[, attr])`
 Inserte una cadena de caracteres (tantos caracteres encajen en la línea) antes los caracteres bajo el cursor. Todos los caracteres a la derecha de el cursor son desplazados a la derecha, perdiéndose los caracteres de la derecha en línea.

`window.instr([n])`
`window.instr(y, x[, n])`
 Regresa un objeto de bytes de caracteres, extraídos desde la ventana comenzando en la posición actual del cursor, o en *y*, *x* si es especificado. Los atributos son despojados desde los caracteres. Si *n* es especificado, `instr()` retorna una cadena de caracteres como máximo de *n* caracteres (exclusivo de la NULL final).

`window.is_linetouched(line)`
 Retorna `True` si la línea especificada fue modificada desde la última llamada a `refresh()`; de otra manera retorna `False`. Genera una excepción `curses.error` si *line* no es válida para la ventana dada.

`window.is_wintouched()`

Retorna `True` si la ventana especificada fue modificada desde la última llamada para `refresh()`; en caso contrario, retorna `False`.

`window.keypad(flag)`

Si `flag` es `True`, evita las secuencias generadas por algunas teclas (teclado, teclas de función) será interpretadas por `curses`. Si `flag` es `False`, evita las secuencias que serán dejadas como está en la corriente de entrada.

`window.leaveok(flag)`

Si `flag` es `True`, el cursor es dejado donde está en la actualización, por ejemplo estando en «cursor position.» Esto reduce el movimiento del cursor siempre que sea posible. Si es posible, el cursor se hará invisible.

Si `flag` es «`False`», el cursor siempre estará en «cursor position» después de una actualización.

`window.move(new_y, new_x)`

Mueve el cursor a «(new_y, new_x)».

`window.mvderwin(y, x)`

Mueve la ventana adentro de su ventana padre. Los parámetros relativos a la pantalla de la ventana no son cambiados. Esta rutina es usada para mostrar diferentes partes de la ventana padre en la misma posición física en la pantalla.

`window.mvwin(new_y, new_x)`

Mueve la ventana a su esquina superior izquierda que está en «(new_y,new_x)».

`window.nodelay(flag)`

Si `flag` es «`True`», `getch()` no será bloqueada.

`window.notimeout(flag)`

Si `flag` es «`True`», las secuencias de escape no serán agotadas.

Si `flag` es `False`, después de pocos milisegundos, una secuencia de escape no será interpretada, y será dejada en el flujo de entrada como está.

`window.noutrefresh()`

Marque para actualizar pero espere. Esta función actualiza la estructura de datos representando el estado deseado de la ventana, pero no fuerza una actualización en la pantalla física. Para realizar eso, llame `doupdate()`.

`window.overlay(destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])`

Cubre la ventana en la parte superior de `destwin`. La ventana no necesita ser del mismo tamaño, solamente se solapa la región copiada. Esta copia no es destructiva, lo que significa que el carácter de fondo actual no sobre-escribe el contenido viejo de `destwin`.

Para obtener el control de grano fino sobre la región copiada, la segunda forma de `overlay()` puede ser usada. `sminrow` y `smincol` son las coordenadas superior izquierda de la ventana de origen, y las otras variables marcan un rectángulo en la ventana destino.

`window.overwrite(destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])`

Sobre-escribe la ventana en la parte superior de `destwin`. La ventana no necesita ser del mismo tamaño, en cuyo caso solamente se sobrepone la región que es copiada. Esta copia es destructiva, lo cual significa que el carácter de fondo actual sobre-escribe el contenido viejo de `destwin`.

Para obtener el control de grano fino sobre la región copiada, la segunda forma de `overwrite()` puede ser usada. `sminrow` y `smincol` son las coordenadas superior izquierda de la ventana origen, las otras variables marcan un rectángulo en la ventana de destino.

`window.putwin(file)`

Escribe todos los datos asociados con la ventana en el objeto de archivo proveído. Esta información puede estar después recuperada usando la función `getwin()`.

`window.redrawln(beg, num)`

Indica que el `num` de líneas de la pantalla, empiezan en la línea `beg`, son corruptos y deberían ser completamente redibujado en la siguiente llamada `refresh()`.

`window.redrawwin()`

Toca la ventana completa, causando que se vuelva a dibujar completamente en la siguiente llamada `refresh()`.

`window.refresh` (*[pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol]*)

Actualiza la pantalla inmediatamente (sincronice la pantalla actual con los métodos previos de dibujar/borrar).

Los 6 argumentos opcionales pueden ser solamente especificados cuando la ventana es una almohadilla creada con `newpad()`. Los parámetros adicionales son necesarios para indicar que parte de la almohadilla y pantalla son envueltos. *pminrow* y *pmincol* especifica la esquina superior de la mano izquierda del rectángulo a ser mostrado en el *pad*. *sminrow*, *smincol*, *smaxrow*, y *smaxcol* especifica los bordes del rectángulo a ser mostrados en la pantalla. La esquina inferior de la mano derecha del rectángulo a ser mostrado en el *pad* es calculado desde las coordenadas de la pantalla, desde que el rectángulo debe ser del mismo tamaño. Ambos rectángulos deben ser completamente contenidos con su estructura respectiva. Valores negativos de *pminrow*, *pmincol*, *sminrow*, o *smincol* son tratados como si fueran cero.

`window.resize` (*nlines, ncols*)

Redistribuir el almacenamiento para una ventana *curses* para ajustar su dimensión para los valores especificados. Si la dimensión también es mayor que los valores actuales, los datos de la ventana serán llenados con espacios en blanco que tiene la ejecución del fondo actual (como ajustado por `bkgdset()`) uniéndolos.

`window.scroll` (*[lines=1]*)

Desplace la pantalla o la región de desplazamiento hacia arriba por *lines* líneas.

`window.scrolllok` (*flag*)

Control que sucede cuando el cursor de una ventana es movida fuera del borde de la ventana o región de desplazamiento, también como un resultado de una acción de nueva línea en la línea inferior, o escribiendo el último carácter de la última línea. Si *flag* es `False`, el cursor está a la izquierda sobre la línea inferior. Si *flag* es `True`, la ventana será desplazada hacia arriba por una línea. Note que en orden para obtener el efecto del desplazamiento físico en el terminal, también será necesario llamar `idlok()`.

`window.setscrreg` (*top, bottom*)

Configura la región de desplazamiento desde la línea *top* hasta la línea *bottom*. Todas las acciones de desplazamiento tomarán lugar en esta región.

`window.standend` ()

Desactive el atributo destacado. En algunos terminales esto tiene el efecto secundario de desactivar todos los atributos.

`window.standout` ()

Active el atributo `A_STANDOUT`.

`window.subpad` (*begin_y, begin_x*)

`window.subpad` (*nlines, ncols, begin_y, begin_x*)

Retorna la sub-ventana, cuya esquina superior izquierda esta en (*begin_y*, *begin_x*), y cuyo ancho/alto es *ncols/nlines*.

`window.subwin` (*begin_y, begin_x*)

`window.subwin` (*nlines, ncols, begin_y, begin_x*)

Retorna la sub-ventana, cuya esquina superior izquierda esta en (*begin_y*, *begin_x*), y cuyo ancho/alto es *ncols/nlines*.

Por defecto, la sub-ventana extenderá desde la posición especificada para la esquina inferior derecha de la ventana.

`window.syncdown` ()

Toca cada ubicación en la ventana que ha sido tocado por alguna de sus ventanas padres. Esta rutina es llamada por `refresh()`, esto casi nunca debería ser necesario para llamarlo manualmente.

`window.syncok` (*flag*)

Si *flag* es `True`, entonces `syncup()` es llamada automáticamente cuando hay un cambio en la ventana.

`window.syncup` ()

Toque todas las locaciones de los ancestros de la ventana que han sido cambiados en la ventana.

`window.timeout` (*delay*)

Configura el bloqueo o no bloqueo del comportamiento para la ventana. Si el *delay* es negativo, bloqueando la lectura usada (el cual esperará indefinidamente para la entrada). Si *delay* es zero, entonces no se bloqueará

la lectura usada, y `getch()` retornará `-1` si la entrada no está esperando. Si `delay` es positivo, entonces `getch()` se bloqueará por `delay` milisegundos, y retorna `-1` si aún no entra en el final de ese tiempo.

`window.touchline(start, count[, changed])`

Supone `count` líneas han sido cambiadas, iniciando con la línea `start`. Si `changed` es suministrado, especifica que las líneas afectadas son marcadas como hayan sido cambiadas (`changed=True`) o no cambiadas (`changed=False`).

`window.touchwin()`

Suponga que toda la ventana ha sido cambiada, con el fin de optimizar el dibujo.

`window.untouchwin()`

Marque todas las líneas en la ventana como no cambiadas desde la última llamada para `refresh()`.

`window.vline(ch, n)`

`window.vline(y, x, ch, n)`

Muestre una línea vertical iniciando en `(y, x)` con longitud `n` consistente de el carácter `ch`.

16.10.3 Constantes

El módulo `curses` define los siguientes miembros de datos:

`curses.ERR`

Algunas rutinas de `curses` que retornan un entero, tales como `getch()`, retorna `ERR` sobre el fallo.

`curses.OK`

Algunas rutinas de `curses` que retornan un entero, tales como `napms()`, retorna `OK` tras el éxito.

`curses.version`

Un objeto de bytes representando la versión actual de el módulo. También disponible como `__version__`.

`curses.ncurses_version`

Una tupla nombrada contiene los 3 componentes de la versión de la librería `ncurses` *major*, *minor*, y *patch*. Todos los valores son enteros. Los componentes pueden también ser accedidos por nombre, así `curses.ncurses_version[0]` es equivalente a `curses.ncurses_version.major` y así.

Disponibilidad: si la librería `ncurses` es usada.

Nuevo en la versión 3.8.

Algunas constantes están disponibles para especificar los atributos del celdas de caracteres. Las constantes exactas disponible dependen del sistema.

Atributo	Significado
<code>A_ALTCHARSET</code>	Modo de conjunto de caracteres alternativo
<code>A_BLINK</code>	Modo parpadeo
<code>A_BOLD</code>	Modo negrita
<code>A_DIM</code>	Modo tenue
<code>A_INVIS</code>	Modo invisible o en blanco
<code>A_ITALIC</code>	Modo cursiva
<code>A_NORMAL</code>	Atributo normal
<code>A_PROTECT</code>	Modo protegido
<code>A_REVERSE</code>	Fondo inverso y colores de primer plano
<code>A_STANDOUT</code>	Modo destacado
<code>A_UNDERLINE</code>	Modo subrayado
<code>A_HORIZONTAL</code>	Resaltado horizontal
<code>A_LEFT</code>	Resaltado a la izquierda
<code>A_LOW</code>	Resaltado bajo
<code>A_RIGHT</code>	Resaltado a la derecha
<code>A_TOP</code>	Resaltado arriba
<code>A_VERTICAL</code>	Resaltado vertical
<code>A_CHARTEXT</code>	Mascara de bits para extraer un caracter

Nuevo en la versión 3.7: `A_ITALIC` fue añadido.

Varias constantes están disponibles para extraer los atributos correspondientes retornados por algunos métodos.

Mascara de bits	Significado
<code>A_ATTRIBUTES</code>	Mascara de bits para extraer atributos
<code>A_CHARTEXT</code>	Mascara de bits para extraer un caracter
<code>A_COLOR</code>	Mascara de bits para extraer información de campo de pares de colores

Las teclas se denominan constantes enteras con nombres que comienzan con `KEY_`. Las teclas exactas disponibles son dependientes del sistema.

Clave constante	Clave
<code>KEY_MIN</code>	Valor mínimo de la clave
<code>KEY_BREAK</code>	Clave rota (no confiable)
<code>KEY_DOWN</code>	Flecha hacia abajo
<code>KEY_UP</code>	Flecha hacia arriba
<code>KEY_LEFT</code>	Flecha hacia la izquierda
<code>KEY_RIGHT</code>	Flecha hacia la derecha
<code>KEY_HOME</code>	Tecla de inicio (flecha hacia arriba + flecha hacia la izquierda)
<code>KEY_BACKSPACE</code>	Retroceso (no confiable)
<code>KEY_F0</code>	Teclas de función. Más de 64 teclas de función son soportadas.
<code>KEY_Fn</code>	Valor de tecla de función <i>n</i>
<code>KEY_DL</code>	Borrar línea
<code>KEY_IL</code>	Inserte línea
<code>KEY_DC</code>	Borrar caracter
<code>KEY_IC</code>	Insertar carácter o ingresara al modo de inserción
<code>KEY_EIC</code>	Salir del modo de inserción de caracteres
<code>KEY_CLEAR</code>	Limpiar pantalla
<code>KEY_EOS</code>	Limpiar al final de pantalla
<code>KEY_EOL</code>	Limpiar al final de la línea
<code>KEY_SF</code>	Desplazar una línea hacia adelante
<code>KEY_SR</code>	Desplazar una línea hacia atrás (reversa)
<code>KEY_NPAGE</code>	Página siguiente
<code>KEY_PPAGE</code>	Página anterior
<code>KEY_STAB</code>	Establecer pestaña
<code>KEY_CTAB</code>	Limpiar pestaña
<code>KEY_CATAB</code>	Limpiar todas las pestañas
<code>KEY_ENTER</code>	Ingresar o enviar (no confiable)
<code>KEY_SRESET</code>	Reinicio suave (parcial) (no confiable)
<code>KEY_RESET</code>	Reinicio o reinicio fuerte (no confiable)
<code>KEY_PRINT</code>	Imprimir
<code>KEY_LL</code>	Inicio abajo o abajo (abajo a la izquierda)
<code>KEY_A1</code>	Superior izquierda del teclado
<code>KEY_A3</code>	Superior derecha de el teclado
<code>KEY_B2</code>	Centro del teclado
<code>KEY_C1</code>	Inferior izquierdo del teclado
<code>KEY_C3</code>	Inferior derecho del teclado
<code>KEY_BTAB</code>	Pestaña trasera
<code>KEY_BEG</code>	Mendigar (Comienzo)
<code>KEY_CANCEL</code>	Cancelar
<code>KEY_CLOSE</code>	Cerrar
<code>KEY_COMMAND</code>	Cmd (Comando)
<code>KEY_COPY</code>	Copiar
<code>KEY_CREATE</code>	Crear

Continúa en la página siguiente

Tabla 1 – proviene de la página anterior

Clave constante	Clave
KEY_END	Final
KEY_EXIT	Salir
KEY_FIND	Encontrar
KEY_HELP	Ayudar
KEY_MARK	Marca
KEY_MESSAGE	Mensaje
KEY_MOVE	Mover
KEY_NEXT	Siguiente
KEY_OPEN	Abrir
KEY_OPTIONS	Opciones
KEY_PREVIOUS	Anterior
KEY_REDO	Rehacer
KEY_REFERENCE	Referencia
KEY_REFRESH	Refrescar
KEY_REPLACE	Re-emplazar
KEY_RESTART	Re-iniciar
KEY_RESUME	Resumir
KEY_SAVE	Guardar
KEY_SBEG	Mendicidad desplazada (comienzo)
KEY_SCANCEL	Cancelar cambiado
KEY_SCOMMAND	Comando cambiado
KEY_SCOPY	Copiado cambiado
KEY_SCREATE	Crear con desplazamiento
KEY_SDC	Borrar carácter desplazado
KEY_SDL	Borrar línea desplazada
KEY_SELECT	Seleccionar
KEY_SEND	Final desplazado
KEY_SEOL	Limpiar línea desplazada
KEY_SEXIT	Salida desplazada
KEY_SFIND	Búsqueda desplazada
KEY_SHELP	Ayuda desplazada
KEY_SHOME	Inicio cambiado
KEY_SIC	Entrada cambiada
KEY_SLEFT	Flecha hacia la izquierda desplazada
KEY_SMESSAGE	Mensaje cambiado
KEY_SMOVE	Movimiento desplazado
KEY_SNEXT	Siguiente desplazado
KEY_SOPTIONS	Opciones cambiadas
KEY_SPREVIOUS	Anterior cambiado
KEY_SPRINT	Imprimir desplazado
KEY_SREDO	Rehacer cambiado
KEY_SREPLACE	Re-emplazo cambiado
KEY_SRIGHT	Flecha hacia la derecha cambiada
KEY_SRSUME	Resumen cambiado
KEY_SSAVE	Guardar desplazado
KEY_SSUSPEND	Suspender cambiado
KEY_SUNDO	Deshacer desplazado
KEY_SUSPEND	Suspender
KEY_UNDO	Deshacer
KEY_MOUSE	Evento del ratón ha ocurrido
KEY_RESIZE	Evento de cambio de tamaño del terminal
KEY_MAX	Valor máximo de clave

En VT100s y su emulador de software, tal como X emulador de terminal, normalmente hay al menos cuatro funciones

de tecla (`KEY_F1`, `KEY_F2`, `KEY_F3`, `KEY_F4`) disponibles, y la tecla flecha mapeada para `KEY_UP`, `KEY_DOWN`, `KEY_LEFT` y `KEY_RIGHT` en la manera obvia. Si tu máquina tiene un teclado de PC, esto asegura la expectativa de las teclas flecha y doce teclas de función (el más viejo de los teclados de PC pueden tener solamente diez teclas de función); también, las siguientes funciones de teclado son estándar:

Tecla	Constante
Insert	<code>KEY_IC</code>
Delete	<code>KEY_DC</code>
Home	<code>KEY_HOME</code>
End	<code>KEY_END</code>
Page Up	<code>KEY_PPAGE</code>
Page Down	<code>KEY_NPAGE</code>

La siguiente tabla lista los caracteres desde el conjunto alterno de caracteres. Estos son heredados desde el terminal VT100, y generalmente estará disponible en emuladores de software tales como terminales X. Cuando no hay gráficos disponibles, *curses* cae en una aproximación cruda de ASCII imprimibles.

Nota: Estos están disponibles solamente después de que `initscr()` ha sido llamada.

Código ACS	Significado
<code>ACS_BBSS</code>	nombre alternativo para la esquina superior derecha
<code>ACS_BLOCK</code>	bloque cuadrado sólido
<code>ACS_BOARD</code>	tablero de cuadrados
<code>ACS_BSBS</code>	nombre alternativo para línea horizontal
<code>ACS_BSSB</code>	nombre alternativo para esquina superior izquierda
<code>ACS_BSSS</code>	nombre alternativo para el soporte superior
<code>ACS_BTEE</code>	soporte inferior
<code>ACS_BULLET</code>	bala
<code>ACS_CKBOARD</code>	tablero inspector (punteado)
<code>ACS_DARROW</code>	flecha punteada hacia abajo
<code>ACS_DEGREE</code>	símbolo de grado
<code>ACS_DIAMOND</code>	diamante
<code>ACS_GEQUAL</code>	mayor que o igual a
<code>ACS_HLINE</code>	línea horizontal
<code>ACS_LANTERN</code>	símbolo de la linterna
<code>ACS_LARROW</code>	flecha izquierda
<code>ACS_LEQUAL</code>	menor que o igual a
<code>ACS_LLCORNER</code>	esquina inferior izquierda
<code>ACS_LRCORNER</code>	esquina inferior derecha
<code>ACS_LTEE</code>	soporte izquierdo
<code>ACS_NEQUAL</code>	signo no igual
<code>ACS_PI</code>	letra pi
<code>ACS_PLMINUS</code>	signo más o menos
<code>ACS_PLUS</code>	gran signo más
<code>ACS_RARROW</code>	flecha hacia la derecha
<code>ACS_RTEE</code>	soporte derecho
<code>ACS_S1</code>	escanear línea 1
<code>ACS_S3</code>	escanear línea 3
<code>ACS_S7</code>	escanear línea 7
<code>ACS_S9</code>	escanear línea 9
<code>ACS_SBBS</code>	nombre alterno para la esquina inferior derecha
<code>ACS_SBSB</code>	nombre alterno para la línea vertical
<code>ACS_SBSS</code>	nombre alterno para el soporte derecho

Continúa en la página siguiente

Tabla 2 – proviene de la página anterior

Código ACS	Significado
ACS_SSBB	nombre alternativo para la esquina inferior izquierda
ACS_SSBS	nombre alternativo para el soporte inferior
ACS_SSSB	nombre alternativo para el soporte izquierdo
ACS_SSSS	nombre alternativo para <i>crossover</i> o un gran más
ACS_STERLING	libra esterlina
ACS_TTEE	soporte superior
ACS_UARROW	flecha hacia arriba
ACS_ULCORNER	esquina superior izquierda
ACS_URCORNER	esquina superior derecha
ACS_VLINE	línea vertical

La siguiente tabla lista los colores pre-definidos:

Constante	Color
COLOR_BLACK	Negro
COLOR_BLUE	Azul
COLOR_CYAN	Cian (azul verdoso claro)
COLOR_GREEN	Verde
COLOR_MAGENTA	Magenta (rojo violáceo)
COLOR_RED	Rojo
COLOR_WHITE	Blanco
COLOR_YELLOW	Amarillo

16.11 `curses.textpad`— Widget de entrada de texto para programas de curses

El módulo `curses.textpad` provee una clase `Textbox` que maneja edición de texto primario en una ventana `curses`, apoyando un conjunto de atajos de teclado re-ensamblando estos de Emacs (esto, también de *Netscape Navigator*, *BEdit 6.x*, *FrameMaker*, y muchos otros programas). Los módulos también proveen una función de dibujo de rectángulo útil para enmarcar las cajas de texto o para otros propósitos.

El módulo `curses.textpad` define la siguiente función:

`curses.textpad.rectangle` (*win, uly, ulx, lry, lrx*)

Dibuja un rectángulo. El primer argumento debe ser un objeto de ventana, los argumentos restantes son coordenadas relativas para esa ventana. El segundo y tercer argumento son las coordenadas “x” y “y” de la esquina superior de la mano izquierda del rectángulo a ser dibujado, el cuarto y quinto argumento son las coordenadas “x” y “y” de la esquina inferior de la mano derecha. El rectángulo será dibujado usando formularios de caracteres VT100/IBM PC en terminales que hace esto posible (incluyendo *xterm* y mas otro emulador de terminal de software). Por otra parte será dibujado con guiones, barras verticales, y signos de más de ASCII.

16.11.1 Objeto de caja de texto

Tú puedes instanciar un objeto `Textbox` como sigue:

class `curses.textpad.Textbox` (*win*)

Retorna un objeto *widget* de caja de texto. El argumento *win* debería ser un objeto `curses.window` en el cual la caja de texto es para ser contenido. El cursor de la caja de texto está inicialmente localizado en la esquina superior de la mano izquierda de la ventana que lo contiene, con coordenadas (0, 0). La bandera instanciada `stripspaces` está inicialmente encendida.

Objeto `Textbox` tiene los siguientes métodos:

edit ([*validator*])

Este es el punto de entrada que normalmente usarás. Esto acepta la edición de las pulsaciones de tecla hasta que uno de las pulsaciones de finalización es introducida. Si *validator* es suministrado, esto debe ser una función. Será llamado para cada pulsación de tecla introducida con la pulsación de tecla como un parámetro; el comando ejecutado está hecho en el resultado. Este método retorna el contenido de la ventana como una cadena de caracteres; si se incluyen espacios en blanco en la ventana se ve afectado por el atributo *stripspaces*.

do_command (*ch*)

Procesa un simple comando al pulsar una tecla. Aquí está las funciones de tecla especiales admitidas:

Pulsación de tecla	Acción
Control-A	Ve al borde izquierdo de la ventana.
Control-B	Cursor hacia la izquierda, pasando a la línea anterior si corresponde.
Control-D	Borra el carácter bajo el cursor.
Control-E	Ve al borde derecho (quita los espacios) o al final de línea (eliminar los espacios).
Control-F	Cursor hacia la derecha, pasando a la línea siguiente cuando corresponda.
Control-G	Terminar, retornando el contenido de la ventana.
Control-H	Eliminar carácter al revés.
Control-J	Terminar si la ventana es de 1 línea, en caso contrario, inserte una nueva línea.
Control-K	Si la línea es blanca bórrela, en caso contrario, limpie hasta el final de línea.
Control-L	Refrescar la pantalla.
Control-N	Cursor hacia abajo; mueve hacia abajo una línea.
Control-O	Inserte una línea en blanco en la posición del cursor.
Control-P	Cursor hacia arriba; mueve hacia arriba una línea.

Las operaciones de movimiento no hacen nada si el cursor está en un borde donde el movimiento no es posible. Los siguientes sinónimos están apoyados siempre que sea posible:

Constante	Pulsación de tecla
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

Todas las otras pulsaciones de teclas están tratadas como un comando para insertar el carácter dado y muévase a la derecha (con ajuste en la línea).

gather ()

Retorne el contenido de la ventana como una cadena de texto; si se incluyen espacios en blanco en la ventana se ve afectado por el miembro *stripspaces*.

stripspaces

Este atributo es una bandera que controla la interpretación de los espacios en blanco en la ventana. Cuando está encendido, los espacios en blanco al final en cada línea son ignorados; cualquier movimiento del cursor que lo coloque en un espacio en blanco final va hasta el final de esa línea, y los espacios en blanco finales son despejados cuando se recopila el contenido de la ventana.

16.12 `curses.ascii` — Utilidades para los caracteres ASCII

El módulo `curses.ascii` proporciona constantes de nombre para caracteres ASCII y funciones para probar la pertenencia a varias clases de los caracteres ASCII. Las constantes proporcionadas son nombres para caracteres de control de la siguiente manera:

Nombre	Significado
NUL	
SOH	Inicio del encabezado, interrupción de la consola
STX	Inicio del texto
ETX	Final del texto
EOT	Fin de la transmisión
ENQ	Consulta, va con el control de flujo ACK
ACK	Reconocimiento
BEL	Campana
BS	Retroceso
TAB	Tabulación
HT	Alias para TAB: «Tabulación horizontal»
LF	Línea de alimentación
NL	Alias para LF: «Nueva línea»
VT	Tabulación vertical
FF	Alimentación de formulario
CR	Retorno de carro (<i>Carriage return</i> en inglés)
SO	<i>Shift-out</i> , comenzar un conjunto de caracteres alternativo
SI	<i>Shift-in</i> , reanudar el conjunto de caracteres predeterminado
DLE	Escape de enlace de datos
DC1	XON, para control de flujo
DC2	Control de dispositivo 2, control de flujo en modo bloque
DC3	XOFF, para control de flujo
DC4	Control de dispositivo 4
NAK	Reconocimiento negativo
SYN	Inactivo sincrónico
ETB	Bloque de transmisión final
CAN	Cancelar
EM	Fin del medio
SUB	Sustituir
ESC	Escapar
FS	Separador de archivos
GS	Separador de grupos
RS	Separador de registros, finalizador en modo bloque
US	Separador de unidades
SP	Espacio
DEL	Eliminar

Tenga en cuenta que muchos de estos tienen poca importancia práctica en el uso moderno. Los mnemónicos se derivan de las convenciones de la teleimpresora que son anteriores a las computadoras digitales.

El módulo proporciona las siguientes funciones, siguiendo el patrón de las de la biblioteca C estándar:

`curses.ascii.isalnum(c)`

Comprueba un carácter alfanumérico ASCII; esto es equivalente a `isalpha(c)` or `isdigit(c)`.

`curses.ascii.isalpha(c)`

Comprueba si hay un carácter alfabético ASCII; es equivalente a `isupper(c)` or `islower(c)`.

`curses.ascii.isascii(c)`

Comprueba un valor de carácter que se ajuste al conjunto ASCII de 7 bits.

`curses.ascii.isblank(c)`

Comprueba si hay un carácter de espacio en blanco ASCII; espacio o tabulación horizontal.

`curses.ascii.iscntrl(c)`

Comprueba un carácter de control ASCII (en el rango de 0x00 a 0x1f o 0x7f).

`curses.ascii.isdigit(c)`

Comprueba si hay un dígito decimal ASCII, desde '0' hasta '9'. Esto es equivalente a `c in string.digits`.

`curses.ascii.isgraph(c)`

Comprueba en ASCII cualquier carácter imprimible excepto el espacio.

`curses.ascii.islower(c)`

Comprueba un carácter ASCII en minúscula.

`curses.ascii.isprint(c)`

Comprueba cualquier carácter imprimible ASCII, incluido el espacio.

`curses.ascii.ispunct(c)`

Comprueba si hay algún carácter ASCII imprimible que no sea un espacio o un carácter alfanumérico.

`curses.ascii.isspace(c)`

Comprueba los caracteres de espacio en blanco ASCII; espacio, línea de alimentación, retorno de carro, formulario de alimentación, tabulación horizontal, tabulación vertical.

`curses.ascii.isupper(c)`

Comprueba una letra mayúscula ASCII.

`curses.ascii.isxdigit(c)`

Comprueba si hay un dígito hexadecimal ASCII. Esto es equivalente a `c in string.hexdigits`.

`curses.ascii.isctrl(c)`

Comprueba un carácter de control ASCII (valores ordinales de 0 a 31)

`curses.ascii.ismeta(c)`

Comprueba si hay un carácter no ASCII (valores ordinales 0x80 y superiores).

Estas funciones aceptan enteros o cadenas de un solo carácter; cuando el argumento es una cadena de caracteres, primero se convierte utilizando la función *built-in* `ord()`.

Tenga en cuenta que todas estas funciones verifican los valores de bits ordinales derivados del carácter de la cadena que ingresa; en realidad, no saben nada sobre la codificación de caracteres de la máquina host.

Las siguientes dos funciones toman una cadena de un solo carácter o un valor de byte entero; devuelven un valor del mismo tipo.

`curses.ascii.ascii(c)`

Retorna el valor ASCII correspondiente a los 7 bits bajos de *c*.

`curses.ascii.ctrl(c)`

Retorna el carácter de control correspondiente al carácter dado (el valor del bit del carácter es bit a bit (* bitwise-anded*) con 0x1f).

`curses.ascii.alt(c)`

Retorna el carácter de 8 bits correspondiente al carácter ASCII dado (el valor del bit de carácter se escribe bit a bit (*bitwise-ored*) con 0x80).

La siguiente función toma una cadena de un solo carácter o un valor entero; devuelve una cadena.

`curses.ascii.unctrl(c)`

Retorna una representación de cadena del carácter ASCII *c*. Si *c* es imprimible, esta cadena es el propio carácter. Si el carácter es un carácter de control (0x00–0x1f) la cadena consta de un signo de intercalación ('^') seguido de la letra mayúscula correspondiente. Si el carácter es una eliminación ASCII (0x7f), la cadena es '^?'. Si

el carácter tiene su meta bit establecido (0x80), el meta bit se elimina, se aplican las reglas anteriores y se antepone '!' al resultado.

`curses.ascii.controlnames`

Una matriz de cadena de caracteres de 33 elementos que contiene los mnemónicos ASCII para los treinta y dos caracteres de control ASCII desde 0 (NUL) a 0x1f (US), en orden, más el mnemónico "SP" para el carácter de espacio.

16.13 `curses.panel` — Una extensión de pila de panel para `curses`

Los paneles son ventanas con la característica de profundidad añadida, por lo que se pueden apilar una encima de la otra, y solo se mostrarán las partes visibles de cada ventana. Los paneles se pueden agregar, mover hacia arriba o hacia abajo en la pila y eliminarse.

16.13.1 Funciones

El módulo `curses.panel` define las siguientes funciones:

`curses.panel.bottom_panel()`

Retorna el panel inferior en la pila del panel.

`curses.panel.new_panel(win)`

Retorna un objeto de panel, asociándolo con la ventana dada *win*. Tenga en cuenta que debe mantener explícitamente el objeto de panel retornado al que se hace referencia. Si no lo hace, el objeto de panel se recoge como basura y elimina de la pila del panel.

`curses.panel.top_panel()`

Retorna el panel superior de la pila de paneles.

`curses.panel.update_panels()`

Actualiza la pantalla virtual después de los cambios en la pila del panel. Esto no llama a `curses.doupdate()`, por lo que tendrás que hacerlo tú mismo.

16.13.2 Objetos de Panel

Los objetos panel, retornados por `new_panel()` arriba, son ventanas con un orden de apilamiento. Siempre hay una ventana asociada a un panel que determina el contenido, mientras que los métodos del panel son responsables de la profundidad de la ventana en la pila del panel.

Los objetos de panel tienen los siguientes métodos:

`Panel.above()`

Retorna el panel situado encima del panel actual.

`Panel.below()`

Retorna el panel debajo del panel actual.

`Panel.bottom()`

Empuja el panel hasta la parte inferior de la pila.

`Panel.hidden()`

Retorna `True` si el panel está oculto (no visible), `False` en caso contrario.

`Panel.hide()`

Ocultar el panel. Esto no elimina el objeto, solo hace que la ventana en la pantalla sea invisible.

`Panel.move(y, x)`

Mueve el panel a las coordenadas de pantalla“(y, x)”.

`Panel.replace(win)`

Cambia la ventana asociada con el panel a la ventana *win*.

`Panel.set_userptr(obj)`

Establece el puntero de usuario del panel en *obj*. Esto se usa para asociar un dato arbitrario con el panel y puede ser cualquier objeto de Python.

`Panel.show()`

Muestra el panel (que podría haber estado oculto).

`Panel.top()`

Empuja el panel hacia la parte superior de la pila.

`Panel.userptr()`

Retorna el puntero del usuario para el panel. Puede ser cualquier objeto de Python.

`Panel.window()`

Retorna el objeto de ventana asociado con el panel.

16.14 `platform` — Acceso a los datos identificativos de la plataforma subyacente

código fuente: `Lib/platform.py`

Nota: Plataformas específicas listadas alfabéticamente, con Linux incluido en la sección de Unix.

16.14.1 Plataforma cruzada

`platform.architecture(executable=sys.executable, bits="", linkage="")`

Consulta el ejecutable provisto (por defecto el archivo binario del intérprete de Python) para obtener información de diversas arquitecturas.

Retorna una tupla (*bits*, *linkage*), siendo *bits* la información sobre la arquitectura del procesador y *linkage* el formato de conexión usado por el ejecutable. Ambos valores se retornan como cadenas.

Los valores que no se pueden determinar se retornan según lo indicado por los ajustes por defecto de los parámetros. Si *bits* se da como `' '`, el `sizeof(pointer)` (o `sizeof(long)` en la versión de Python < 1.5.2) se utiliza como indicador para el tamaño del puntero admitido.

La función se basa en el comando `file` del sistema para realizar la tarea. Está disponible en la mayoría de las plataformas Unix, si no en todas, y en algunas plataformas que no son de Unix y solo si el ejecutable apunta al intérprete de Python. Unos valores por defecto se utilizan cuando no se satisfacen las necesidades anteriores.

Nota: En Mac OS X (y quizás en otras plataformas), los archivos ejecutables pueden ser archivos universales que contienen varias arquitecturas.

Para llegar a los «64-bits» del intérprete actual, es más seguro consultar el atributo `sys.maxsize`:

```
is_64bits = sys.maxsize > 2**32
```

`platform.machine()`

Retorna el tipo de máquina, por ejemplo `'i386'`. Si no se puede determinar el valor, se retorna una cadena vacía.

`platform.node()`

Retorna el nombre de la red del ordenador (¡tal vez no sea el nombre completo!). Si no se puede determinar el valor, se retorna una cadena vacía.

`platform.platform(aliased=0, terse=0)`

Retorna una sola cadena identificando la plataforma subyacente con la mayor información útil posible.

La salida se intenta que sea *humanamente legible* más que tratable por una máquina. Tal vez la salida sea diferente en diversas plataformas y eso mismo es lo que se pretende.

Si *aliased* es verdadero, la función usará aliases para varias plataformas que informen de nombres de sistema que sean diferentes a sus nombres comunes. Por ejemplo, SunOS se reportará como Solaris. La función `system_alias()` ha sido usada para implementar esto.

Estableciendo *terse* a verdadero provoca que la función retorne el mínimo de información necesaria para identificar la plataforma.

Distinto en la versión 3.8: En macOS, la función ahora usa `mac_ver()`, si retorna una cadena no vacía para obtener la versión de macOS más que la versión de darwin.

`platform.processor()`

Retorna el nombre (real) del procesador. E.j. 'amd64'.

Una cadena vacía se retorna si el valor no se puede determinar. Destacar que muchas plataformas no proveen esta información o simplemente retorna los mismos valores que para `machine()`, como hace NetBSD.

`platform.python_build()`

Retorna una tupla (`buildno`, `builddate`) con *buildno* indicando el número de la build de Python y *builddate* su fecha de publicación como cadenas.

`platform.python_compiler()`

Retorna la string con la identificación del compilador usado para compilar Python.

`platform.python_branch()`

Retorna la string identificando la implementación de la rama SCM de Python.

`platform.python_implementation()`

Retorna la string identificando la implementación de Python. Algunos valores posibles son: "CPython", "IronPython", "Jython", "PyPy".

`platform.python_revision()`

Retorna la string identificando la implementación de la revisión SCM de Python.

`platform.python_version()`

Retorna la versión de Python en formato de cadena de caracteres con la forma 'major.minor.patchlevel' siendo *major* la versión principal, *minor* la versión menor y *patchlevel* el último parche aplicado.

Destacar que a diferencia del `sys.version` de Python, el valor retornado siempre incluirá el último parche aplicado (siendo 0 por defecto).

`platform.python_version_tuple()`

Retorna la versión de Python como una tupla (`major`, `minor`, `patchlevel`) de cadena, siendo *major* la versión principal, *minor* la versión menor y *patchlevel* último parche aplicado.

Destacar que a diferencia del `sys.version` de Python, el valor retornado siempre incluirá el último parche aplicado (siendo '0' por defecto).

`platform.release()`

Retorna la versión de publicación del sistema. Por ejemplo '2.2.0' o 'NT'. Si no se puede determinar el valor, se retorna una cadena vacía.

`platform.system()`

Retorna el nombre del sistema/SO, como 'Linux', 'Darwin', 'Java', 'Windows'. Si no se puede determinar el valor, se retorna una cadena vacía.

`platform.system_alias(system, release, version)`

Retorna la tupla (`system`, `release`, `version`) con los aliases de los nombres comerciales usados por

algunos sistemas siendo *system* el nombre comercial del sistema, *release* como la versión principal de publicación y *version* como el número de la versión del sistema. También hace cierta reordenación de la información en algunos casos donde se produjera algún tipo de confusión.

`platform.version()`

Retorna la versión de la publicación del sistema. Por ejemplo: `'#3 n degas'`. Una cadena vacía se retorna en el caso de que el valor no pueda ser determinado.

`platform.uname()`

Interfaz `uname` relativamente portable. Retorna una `namedtuple()` con seis atributos: `system`, `node`, `release`, `version`, `machine`, and `processor`.

Destacar que añade un sexto atributo (`processor`) que no está presente en el resultado de la función `os.uname()`. Los dos primeros atributos tienen nombres diferentes a los que tiene `os.uname()`, que los llama `sysname` y `nodename`.

Cualquier entrada que no pueda ser determinada se establece como `' '`.

Distinto en la versión 3.3: El resultado ha cambiado de tupla a `namedtuple`.

16.14.2 Plataforma Java

`platform.java_ver(release="", vendor="", vminfo=(",", ", "), osinfo=(",", ", "))`

Versión de la interfaz de Jython.

Retorna una tupla (`release`, `vendor`, `vminfo`, `osinfo`) con `vminfo` siendo una tupla (`vm_name`, `vm_release`, `vm_vendor`) y `osinfo` siendo una tupla (`os_name`, `os_version`, `os_arch`). Los valores que no se pueden determinar se establecen por defecto de los parámetros (que todos los valores predeterminados son `' '`).

16.14.3 Plataforma windows

`platform.win32_ver(release="", version="", csd="", ptype="")`

Obtiene información adicional sobre la versión desde el registro de Windows y retorna una tupla `()` que hace referencia a la versión del SO, el número de la versión, el nivel CSD (service pack) y el tipo de SO (multiprocesador o no)

Como sugerencia: `ptype` es `'Uniprocessor Free'` en máquinas NT de procesador único y `""Multiprocessor Free""` en máquinas multiprocesador. El `"Free"` se refiere a que la versión del sistema operativo está libre de código de depuración. También podría indicar `"Checked"` lo que significa que la versión del sistema operativo utiliza código de depuración, es decir, código que comprueba argumentos, rangos, etc.

`platform.win32_edition()`

Retorna una cadena que representa la edición actual de Windows. Los valores posibles incluyen, entre otros, `'Enterprise'`, `'IoTAP'`, `'ServerStandard'` y `'nanoserver'`.

Nuevo en la versión 3.8.

`platform.win32_is_iot()`

Retorna `True` si la edición de Windows retornada por `win32_edition()` se reconoce como una edición IoT.

Nuevo en la versión 3.8.

16.14.4 Plataforma Mac OS

`platform.mac_ver(release="", versioninfo="", machine=")`

Obtiene información de la versión de Mac OS y retorna una tupla (`release`, `versioninfo`, `machine`) con `versioninfo` siendo una tupla (`versión`, `dev_stage`, `non_release_version`).

Cualquier registro que no puede ser determinado se establece como `' '`. Todos los registros de la tupla son cadenas.

16.14.5 Plataformas Unix

`platform.libc_ver(executable=sys.executable, lib="", version="", chunksize=16384)`

Intenta determinar la versión `libc` al que está enlazado el fichero ejecutable (por defecto el intérprete de Python). Retorna una tupla de cadenas (`lib`, `version`) que tiene por defecto los parámetros que han sido introducidos en caso de que la búsqueda fallase.

Destacar que esta función tiene un conocimiento íntimo de cómo las diferentes versiones de `libc` agregan símbolos al ejecutable. Probablemente sólo se puede utilizar para los ejecutables compilados mediante **gcc**.

El archivo se lee y se analiza en fragmentos de bytes `chunksize`.

16.15 `errno` — Símbolos estándar del sistema `errno`

Este módulo pone a disposición los símbolos estándar del sistema `errno`. El valor de cada símbolo es el valor entero correspondiente. Los nombres y descripciones están tomados de `linux/include/errno.h`, que debería ser bastante completo.

`errno.errorcode`

Diccionario que proporciona un mapeo del valor de `errno` al nombre de la cadena en el sistema subyacente. Por ejemplo, `errno.errorcode[errno.EPERM]` se asigna a `'EPERM'`.

Para traducir un código de error numérico en un mensaje de error, use `os.strerror()`.

De la siguiente lista, los símbolos que no se utilizan en la plataforma actual no están definidos por el módulo. La lista específica de símbolos definidos está disponible como `errno.errorcode.keys()`. Los símbolos disponibles pueden incluir:

`errno.EPERM`

Operación no permitida

`errno.ENOENT`

El fichero o directorio no existe

`errno.ESRCH`

No hay tal proceso

`errno.EINTR`

Llamada al sistema interrumpida.

Ver también:

Este error se asigna a la excepción `InterruptedError`.

`errno.EIO`

Error de E/S

`errno.ENXIO`

No existe tal dispositivo o dirección

`errno.E2BIG`

Lista de argumentos demasiado larga

`errno.ENOEXEC`
Error de formato de ejecución

`errno.EBADF`
Número de archivo incorrecto

`errno.ECHILD`
Sin procesos secundarios

`errno.EAGAIN`
Vuelva a intentar

`errno.ENOMEM`
Sin memoria

`errno.EACCES`
Permiso denegado

`errno.EFAULT`
Dirección incorrecta

`errno.ENOTBLK`
Bloquear dispositivo requerido

`errno.EBUSY`
Dispositivo o recurso ocupado

`errno.EEXIST`
El archivo existe

`errno.EXDEV`
Enlace entre dispositivos

`errno.ENODEV`
Hay tal dispositivo

`errno.ENOTDIR`
No es un directorio

`errno.EISDIR`
Es un directorio

`errno.EINVAL`
Argumento inválido

`errno.ENFILE`
Desbordamiento de la tabla de archivos

`errno.EMFILE`
Demasiados archivos abiertos

`errno.ENOTTY`
No es un typewriter

`errno.ETXTBSY`
Archivo de texto ocupado

`errno.EFBIG`
Archivo demasiado grande

`errno.ENOSPC`
No queda espacio en el dispositivo

`errno.ESPIPE`
Búsqueda ilegal

`errno.EROFS`
Sistema de archivos de sólo lectura

`errno.EMLINK`
Demasiados enlaces

`errno.EPIPE`
Tubería rota

`errno.EDOM`
Argumento matemático fuera del dominio de función

`errno.ERANGE`
Resultado matemático no representable

`errno.EDEADLK`
Podría ocurrir un bloqueo de recursos

`errno.ENAMETOOLONG`
Nombre de archivo demasiado largo

`errno.ENOLCK`
No hay bloqueos de registro disponibles

`errno.ENOSYS`
Función no implementada

`errno.ENOTEMPTY`
Directorio no vacío

`errno.ELOOP`
Se han encontrado demasiados enlaces simbólicos

`errno.EWOULDBLOCK`
La operación podría bloquearse

`errno.ENOMSG`
Ningún mensaje del tipo deseado

`errno.EIDRM`
Identificador eliminado

`errno.ECHRNG`
Número de canal fuera de rango

`errno.EL2NSYNC`
Nivel 2 no sincronizado

`errno.EL3HLT`
Nivel 3 detenido

`errno.EL3RST`
Nivel 3 restablecido

`errno.ELNRNG`
Número de enlace fuera de rango

`errno.EUNATCH`
Controlador de protocolo no adjunto

`errno.ENOCSI`
No hay estructura CSI disponible

`errno.EL2HLT`
Nivel 2 detenido

`errno.EBADE`
Intercambio inválido

`errno.EBADR`
Descriptor de solicitud inválido

`errno.EXFULL`
Intercambio completo

`errno.ENOANO`
Sin ánodo

`errno.EBADRQC`
Código de solicitud inválido

`errno.EBADSLT`
Ranura inválida

`errno.EDEADLOCK`
Error de interbloqueo de bloqueo de archivos

`errno.EBFONT`
Formato de archivo de fuente incorrecto

`errno.ENOSTR`
El dispositivo no es una secuencia

`errno.ENODATA`
Datos no disponibles

`errno.ETIME`
Temporizador expirado

`errno.ENOSR`
Recursos fuera de flujos

`errno.ENONET`
La computadora no está en la red

`errno.ENOPKG`
Paquete no instalado

`errno.EREMOTE`
El objeto es remoto

`errno.ENOLINK`
El enlace ha sido cortado

`errno.EADV`
Error de publicidad

`errno.ESRMNT`
Error de Srmount

`errno.ECOMM`
Error de comunicación al enviar

`errno.EPROTO`
Error de protocolo

`errno.EMULTIHOP`
Intento de salto múltiple

`errno.EDOTDOT`
Error específico de RFS (por su significado en inglés *Remote File System*)

`errno.EBADMSG`
No es un mensaje de datos

`errno.EOVERFLOW`
Valor demasiado grande para el tipo de datos definido

`errno.ENOTUNIQ`
Nombre no único en la red

`errno.EBADFD`
Descriptor de archivo en mal estado

`errno.EREMCHG`
La dirección remota cambió

`errno.ELIBACC`
No se puede acceder a una biblioteca compartida necesaria

`errno.ELIBBAD`
Accediendo a una biblioteca compartida dañada

`errno.ELIBSCN`
Sección .lib en a.out corrupta

`errno.ELIBMAX`
Intentando vincular demasiadas bibliotecas compartidas

`errno.ELIBEXEC`
No se puede ejecutar una biblioteca compartida directamente

`errno.EILSEQ`
Secuencia de byte ilegal

`errno.ERESTART`
Llamada al sistema interrumpida debe reiniciarse

`errno.ESTRPIPE`
Error de tubería de flujos

`errno.EUSERS`
Demasiados usuarios

`errno.ENOTSOCK`
Operación de socket en no-socket

`errno.EDESTADDRREQ`
Dirección de destino requerida

`errno.EMSGSIZE`
Mensaje demasiado largo

`errno.EPROTOTYPE`
Protocolo de tipo incorrecto para socket

`errno.ENOPROTOOPT`
Protocolo no disponible

`errno.EPROTONOSUPPORT`
Protocolo no soportado

`errno.ESOCKTNOSUPPORT`
Tipo de socket no soportado

`errno.EOPNOTSUPP`
Operación no soportada en el punto final de transporte

`errno.EPFNOSUPPORT`
Familia de protocolo no soportada

`errno.EAFNOSUPPORT`
Familia de direcciones no soportada por protocolo

`errno.EADDRINUSE`
Dirección ya en uso

`errno.EADDRNOTAVAIL`
No se puede asignar la dirección solicitada

`errno.ENETDOWN`
Red caída

`errno.ENETUNREACH`
Red es inalcanzable

`errno.ENETRESET`
Conexión de red interrumpida debido al reinicio

`errno.ECONNABORTED`
El software causó falla de conexión

`errno.ECONNRESET`
Conexión restablecida por par

`errno.ENOBUFS`
No hay espacio de búfer disponible

`errno.EISCONN`
El punto final de transporte ya está conectado

`errno.ENOTCONN`
El punto final de transporte no está conectado

`errno.ESHUTDOWN`
No se puede enviar después de apagar el punto final de transporte

`errno.ETOOMANYREFS`
Demasiadas referencias: no se puede empalmar

`errno.ETIMEDOUT`
Tiempo de conexión agotado

`errno.ECONNREFUSED`
Conexión rechazada

`errno.EHOSTDOWN`
Anfitrión caído

`errno.EHOSTUNREACH`
Sin ruta al anfitrión

`errno.EALREADY`
Operación ya en curso

`errno.EINPROGRESS`
Operación ahora en progreso

`errno.ESTALE`
Manejador de archivos NFS (por su significado en inglés *Network File System*) obsoleto

`errno.EUCLEAN`
La estructura necesita limpieza

`errno.ENOTNAM`
No es un archivo de tipo con nombre XENIX

`errno.ENAVAIL`
No hay semáforos XENIX disponibles

`errno.EISNAM`
Es un archivo de tipo con nombre

`errno.EREMOTEIO`
Error de E/S remota

`errno.EDQUOT`
Cuota excedida

16.16 ctypes — Una biblioteca de funciones foráneas para Python

`ctypes` es una biblioteca de funciones foráneas para Python. Proporciona tipos de datos compatibles con C y permite llamar a funciones en archivos DLL o bibliotecas compartidas. Se puede utilizar para envolver estas bibliotecas en Python puro.

16.16.1 tutorial de ctypes

Nota: Los ejemplos de código de este tutorial utilizan `doctest` para asegurarse de que realmente funcionan. Dado que algunos ejemplos de código se comportan de manera diferente en Linux, Windows o Mac OS X, contienen directivas de prueba en los comentarios.

Nota: Algunos ejemplos de código hacen referencia al tipo ctypes `c_int`. En las plataformas donde `sizeof(long) == sizeof(int)` es un alias de `c_long`. Por lo tanto, no debe confundirse si `c_long` se imprime si espera `c_int` — son en realidad del mismo tipo.

Carga de bibliotecas de enlaces dinámicos

`ctypes` exporta los objetos `cdll` y en Windows `windll` y `oledll`, para cargar bibliotecas de enlaces dinámicos.

Las bibliotecas se cargan accediendo a ellas como atributos de estos objetos. `cdll` carga bibliotecas que exportan funciones utilizando la convención de llamada estándar `cdecl`, mientras que las bibliotecas `windll` llaman a funciones mediante la convención de llamada `stdcall`. `oledll` también utiliza la convención de llamada `stdcall` y asume que las funciones retornan un código de error Windows `HRESULT`. El código de error se utiliza para generar automáticamente una excepción `OSError` cuando se produce un error en la llamada a la función.

Distinto en la versión 3.3: Los errores de Windows solían generar `WindowsError`, que ahora es un alias de `OSError`.

Estos son algunos ejemplos para Windows. Tener en cuenta que “`msvcrt`” es la biblioteca estándar de MS C que contiene la mayoría de las funciones C estándar y utiliza la convención de llamada `cdecl`:

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows agrega automáticamente la extensión común `.dll`.

Nota: Acceder a la biblioteca estándar de C a través de `cdll.msvcrt` utilizará una versión obsoleta de la biblioteca que puede ser incompatible con la utilizada por Python. Cuando sea posible, use la funcionalidad nativa de Python, o bien importe y use el módulo `msvcrt`.

En Linux, se requiere especificar el nombre de archivo *incluyendo* la extensión para cargar una biblioteca, por lo que no se puede utilizar el acceso por atributos para cargar las bibliotecas. Se debe usar el método `LoadLibrary()` de los cargadores de dll, o se debe cargar la biblioteca creando una instancia de `CDLL` llamando al constructor:

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

Acceder a las funciones de los dll cargados

Las funciones se acceden como atributos de los objetos dll:

```
>>> 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
>>>
```

Nótese que las dlls del sistema win32 como `kernel32` y `user32` a menudo exportan versiones ANSI y UNICODE de una función. La versión UNICODE se exporta con una `W` añadida al nombre, mientras que la versión ANSI se exporta con una `A` añadida al nombre. La función `GetModuleHandle` de win32, que retorna un *manejador de módulo* para un nombre de módulo dado, tiene el siguiente prototipo de C, y se usa una macro para exponer uno de ellos como `GetModuleHandle` dependiendo de si UNICODE está definido o no:

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

`windll` no intenta seleccionar una de ellas por arte de magia, se debe acceder a la versión que se necesita especificando `GetModuleHandleA` o `GetModuleHandleW` explícitamente, y luego llamarlo con bytes u objetos de cadena respectivamente.

A veces, las dlls exportan funciones con nombres que no son identificadores válidos de Python, como `"??2@YAPAXI@Z"`. En este caso tienes que usar `getattr()` para recuperar la función:

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

En Windows, algunas dlls exportan funciones no por nombre sino por ordinal. Se pueden acceder a estas funciones indexando el objeto dll con el número ordinal:

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
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
>>>
```

Funciones de llamada

Puedes llamar a estas funciones como cualquier otra función en Python. Este ejemplo utiliza la función `time()`, que retorna el tiempo del sistema en segundos desde la época de Unix, y la función `GetModuleHandleA()`, que retorna un manejador de módulo de win32.

Este ejemplo llama a ambas funciones con un puntero NULL (`None` debe ser usado como el puntero NULL):

```
>>> print(libc.time(None))
1150640792
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
>>>
```

`ValueError` es lanzado cuando se llama a una función `stdcall` con la convención de llamada `cdecl`, o viceversa:

```
>>> 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(b"spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

Para saber la convención de llamada correcta, hay que mirar en el archivo de encabezado C o en la documentación de la función que se quiere llamar.

En Windows, `ctypes` utiliza la gestión de excepciones estructurada de win32 para evitar que se produzcan fallos de protección general cuando se llaman funciones con valores de argumento inválidos:

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000020
>>>
```

Sin embargo, hay suficientes maneras de bloquear Python con `ctypes`, así que debes tener cuidado de todos modos. El módulo `faulthandler` puede ser útil para depurar bloqueos (por ejemplo, provenientes de fallos de segmentación producidos por llamadas erróneas a la biblioteca C).

Los objetos `None`, enteros, bytes y cadenas (unicode) son los únicos objetos nativos de Python que pueden ser usados directamente como parámetros en estas llamadas a funciones. `None` se pasa como puntero de C `NULL`, los objetos bytes y las cadenas se pasan como puntero al bloque de memoria que contiene sus datos (`char *owchar_t *`). Los enteros de Python se pasan como por defecto en la plataforma como tipo `int` de C, su valor se enmascara para que encuadre en el tipo C.

Antes de pasar a llamar funciones con otros tipos de parámetros, tenemos que aprender más sobre los tipos de datos `ctypes`.

Tipos de datos fundamentales

`ctypes` define un número de tipos de datos primitivos compatibles con C:

tipo ctypes	Tipo C	Tipo Python
<code>c_bool</code>	<code>_Bool</code>	bool (1)
<code>c_char</code>	<code>char</code>	Un objeto bytes de 1-caracter
<code>c_wchar</code>	<code>wchar_t</code>	Una cadena de 1-caracter
<code>c_byte</code>	<code>char</code>	entero
<code>c_ubyte</code>	<code>unsigned char</code>	entero
<code>c_short</code>	<code>short</code>	entero
<code>c_ushort</code>	<code>unsigned short</code>	entero
<code>c_int</code>	<code>int</code>	entero
<code>c_uint</code>	<code>unsigned int</code>	entero
<code>c_long</code>	<code>long</code>	entero
<code>c_ulong</code>	<code>unsigned long</code>	entero
<code>c_longlong</code>	<code>__int64</code> o <code>long long</code>	entero
<code>c_ulonglong</code>	<code>unsigned __int64</code> o <code>unsigned long long</code>	entero
<code>c_size_t</code>	<code>size_t</code>	entero
<code>c_ssize_t</code>	<code>ssize_t</code> o <code>Py_ssize_t</code>	entero
<code>c_float</code>	<code>float</code>	flotante
<code>c_double</code>	<code>double</code>	flotante
<code>c_longdouble</code>	<code>long double</code>	flotante
<code>c_char_p</code>	<code>char *</code> (NUL terminated)	objeto de bytes o None
<code>c_wchar_p</code>	<code>wchar_t *</code> (NUL terminated)	cadena o None
<code>c_void_p</code>	<code>void *</code>	entero o None

(1) El constructor acepta cualquier objeto con valor verdadero.

Todos estos tipos pueden ser creados llamándolos con un inicializador opcional del tipo y valor correctos:

```
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

Dado que estos tipos son mutables, su valor también puede ser cambiado después:

```
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
>>> i.value = -99
>>> print(i.value)
-99
>>>
```

Asignando un nuevo valor a las instancias de los tipos de punteros `c_char_p`, `c_wchar_p`, y `c_void_p` cambia el *lugar de memoria* al que apuntan, *no el contenido* del bloque de memoria (por supuesto que no, porque los objetos de bytes de Python son inmutables):

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> print(c_s.value)
Hello World
>>> c_s.value = "Hi, there"
>>> print(c_s)           # the memory location has changed
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)             # first object is unchanged
Hello, World
>>>
```

Sin embargo, debe tener cuidado de no pasarlos a funciones que esperan punteros a la memoria mutable. Si necesitas bloques de memoria mutables, ctypes tiene una función `create_string_buffer()` que los crea de varias maneras. El contenido actual del bloque de memoria puede ser accedido (o cambiado) con la propiedad `raw`; si quieres acceder a él como cadena terminada NUL, usa la propiedad `value`:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)           # create a 3 byte buffer, initialized_
↳to NUL bytes
>>> print(sizeof(p), repr(p.raw))
3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")    # create a buffer containing a NUL_
↳terminated string
>>> print(sizeof(p), repr(p.raw))
6 b'Hello\x00'
>>> print(repr(p.value))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10) # create a 10 byte buffer
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00'
>>>
```

La función `create_string_buffer()` reemplaza a la función `c_buffer()` (que todavía está disponible como un alias), así como a la función `c_string()` de versiones anteriores de ctypes. Para crear un bloque de memoria mutable que contenga caracteres unicode del tipo C `wchar_t` utilice la función `create_unicode_buffer()`.

Funciones de llamada, continuación

Note que `printf` imprime al canal de salida estándar real, no a `sys.stdout`, por lo que estos ejemplos sólo funcionarán en el prompt de la consola, no desde dentro de *IDLE* o *Python Win*:

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf(b"%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
>>>
```

Como se ha mencionado antes, todos los tipos de Python, excepto los enteros, cadenas y objetos bytes, tienen que ser envueltos en su correspondiente tipo *ctypes*, para que puedan ser convertidos al tipo de datos C requerido:

```
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

Funciones de llamada con sus propios tipos de datos personalizados

También puedes personalizar la conversión de argumentos de *ctypes* para permitir que las instancias de tus propias clases se usen como argumentos de función. *ctypes* busca un atributo `_as_parameter_` y lo usa como argumento de función. Por supuesto, debe ser uno de entero, cadena o bytes:

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

Si no quieres almacenar los datos de la instancia en la variable de instancia `_as_parameter_`, puedes definir una *property* que haga que el atributo esté disponible a petición.

Especificar los tipos de argumentos requeridos (prototipos de funciones)

Es posible especificar los tipos de argumentos necesarios de las funciones exportadas desde las DLL estableciendo el atributo `argtypes`.

`argtypes` debe ser una secuencia de tipos de datos de C (la función `printf` probablemente no es un buen ejemplo aquí, porque toma un número variable y diferentes tipos de parámetros dependiendo del formato de la cadena, por otro lado esto es bastante útil para experimentar con esta característica):

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

La especificación de un formato protege contra los tipos de argumentos incompatibles (al igual que un prototipo para una función C), e intenta convertir los argumentos en tipos válidos:

```
>>> printf(b"%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(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
13
>>>
```

Si has definido tus propias clases las cuales pasas a las llamadas a funciones, tienes que implementar un método de clase `from_param()` para que puedan ser usadas en la secuencia `argtypes`. El método de clase `from_param()` recibe el objeto Python que se le pasa a la llamada a función, debería hacer una comprobación de tipo o lo que sea necesario para asegurarse de que este objeto es aceptable, y luego retornar el objeto en sí, su atributo `_as_parameter_`, o lo que se quiera pasar como argumento de la función C en este caso. De nuevo, el resultado debe ser un entero, una cadena, unos bytes, una instancia *ctypes*, o un objeto con el atributo `_as_parameter_`.

Tipos de retorno

Por defecto, se supone que las funciones retornan el tipo C `int`. Se pueden especificar otros tipos de retorno estableciendo el atributo `restype` del objeto de la función.

Aquí hay un ejemplo más avanzado, utiliza la función `strchr`, que espera un puntero de cadena y un carácter, y retorna un puntero a una cadena:

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p      # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

Si quieres evitar las llamadas `ord("x")` de arriba, puedes establecer el atributo `argtypes`, y el segundo argumento se convertirá de un objeto de un solo carácter de bytes de Python a un `char`:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
'def'
>>>
```

También puedes usar un objeto Python invocable (una función o una clase, por ejemplo) como el atributo `restype`, si la función foránea retorna un número entero. El objeto invocable será llamado con el *entero* que la función C retorna, y el resultado de esta llamada será utilizado como resultado de la llamada a la función. Esto es útil para comprobar si hay valores de retorno de error y plantear automáticamente una excepción:

```
>>> 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
OSError: [Errno 126] The specified module could not be found.
>>>
```

`WinError` es una función que llamará a la api Windows `FormatMessage` para obtener la representación de la cadena de un código de error, y retornará una excepción. `WinError` toma un parámetro de código de error opcional, si no se usa ninguno, llama a `GetLastError()` para recuperarlo.

Tenga en cuenta que un mecanismo de comprobación de errores mucho más potente está disponible a través del atributo `errcheck`; consulte el manual de referencia para obtener más detalles.

Pasar los punteros (o: pasar los parámetros por referencia)

A veces una función api C espera un *puntero* a un tipo de datos como parámetro, probablemente para escribir en el lugar correspondiente, o si los datos son demasiado grandes para ser pasados por valor. Esto también se conoce como *pasar parámetros por referencia*.

`ctypes` exporta la función `byref()` que se utiliza para pasar parámetros por referencia. El mismo efecto se puede conseguir con la función `pointer()`, aunque `pointer()` hace mucho más trabajo ya que construye un objeto puntero real, por lo que es más rápido usar `byref()` si no se necesita el objeto puntero en el propio Python:

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc sscanf(b"1 3.14 Hello", b"%d %f %s",
...             byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
>>>
```

Estructuras y uniones

Las estructuras y uniones deben derivar de las clases base `Structure` y `Union` que se definen en el módulo `ctypes`. Cada subclase debe definir un atributo `_fields_`. `_fields_` debe ser una lista de 2-tuplas, que contenga un *nombre de campo* y un *tipo de campo*.

El tipo de campo debe ser un tipo `ctypes` como `c_int`, o cualquier otro tipo `ctypes` derivado: estructura, unión, matriz, puntero.

Aquí hay un ejemplo simple de una estructura POINT, que contiene dos enteros llamados *x* y *y*, y también muestra cómo inicializar una estructura en el 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>
TypeError: too many initializers
>>>
```

Sin embargo, se pueden construir estructuras mucho más complicadas. Una estructura puede contener por sí misma otras estructuras usando una estructura como tipo de campo.

Aquí hay una estructura RECT que contiene dos POINTs llamados *upperleft* (superior izquierda) y *lowerright* (abajo a la derecha):

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>>
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>>
```

Las estructuras anidadas también pueden ser inicializadas en el constructor de varias maneras:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

El campo *descriptor* puede ser recuperado de la *class*, son útiles para la depuración porque pueden proporcionar información útil:

```
>>> print(POINT.x)
<Field type=c_long, ofs=0, size=4>
>>> print(POINT.y)
<Field type=c_long, ofs=4, size=4>
>>>
```

Advertencia: *ctypes* no soporta el paso de uniones o estructuras con campos de bits a funciones por valor. Aunque esto puede funcionar en 32-bit x86, la biblioteca no garantiza que funcione en el caso general. Las uniones y estructuras con campos de bits siempre deben pasarse a las funciones por puntero.

Alineación de estructura/unión y orden de bytes

Por defecto, los campos de Estructura y Unión están alineados de la misma manera que lo hace el compilador C. Es posible anular este comportamiento especificando un atributo de clase `_pack_` en la definición de la subclase. Este debe ser establecido como un entero positivo y especifica la alineación máxima de los campos. Esto es lo que `#pragma pack(n)` también hace en MSVC.

ctypes utiliza el orden de bytes nativos para las Estructuras y Uniones. Para construir estructuras con un orden de bytes no nativo, puedes usar una de las clases base *BigEndianStructure*, *LittleEndianStructure*, *BigEndianUnion*, y *LittleEndianUnion*. Estas clases no pueden contener campos puntero.

Campos de bits en estructuras y uniones

Es posible crear estructuras y uniones que contengan campos de bits. Los campos de bits sólo son posibles para campos enteros, el ancho de bit se especifica como el tercer ítem en las tuplas `_fields_`:

```
>>> 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>
>>>
```

Arreglos

Los arreglos son secuencias, que contienen un número fijo de instancias del mismo tipo.

La forma recomendada de crear tipos de arreglos es multiplicando un tipo de dato por un entero positivo:

```
TenPointsArrayType = POINT * 10
```

Aquí hay un ejemplo de un tipo de datos algo artificial, una estructura que contiene 4 POINTs entre otras cosas:

```
>>> 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
>>>
```

Las instancias se crean de la manera habitual, llamando a la clase:

```
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

El código anterior imprime una serie de líneas 0 0, porque el contenido del arreglos se inicializa con ceros.

También se pueden especificar inicializadores del tipo correcto:

```
>>> 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, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>>
```

Punteros

Las instancias de puntero se crean llamando a la función `pointer()` en un tipo `ctypes`:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

Las instancias del puntero tienen un atributo `contents` que retorna el objeto al que apunta el puntero, el objeto `i` arriba:

```
>>> pi.contents
c_long(42)
>>>
```

Ten en cuenta que `ctypes` no tiene OOR (original object return), construye un nuevo objeto equivalente cada vez que recuperas un atributo:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

Asignar otra instancia `c_int` al atributo de contenido del puntero causaría que el puntero apunte al lugar de memoria donde se almacena:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

Las instancias de puntero también pueden ser indexadas con números enteros:

```
>>> pi[0]
99
>>>
```

Asignando a un índice entero cambia el valor señalado:

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
>>> print(i)
c_long(22)
>>>
```

También es posible usar índices diferentes de 0, pero debes saber lo que estás haciendo, al igual que en C: Puedes acceder o cambiar arbitrariamente las ubicaciones de memoria. Generalmente sólo usas esta característica si recibes un puntero de una función C, y *sabes* que el puntero en realidad apunta a un arreglo en lugar de a un solo elemento.

Entre bastidores, la función `pointer()` hace más que simplemente crear instancias de puntero, tiene que crear primero punteros *tipos*. Esto se hace con la función `POINTER()`, que acepta cualquier tipo de `ctypes`, y retorna un nuevo tipo:

```
>>> 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...>
>>>
```

Llamar al tipo de puntero sin un argumento crea un puntero NULL. Los punteros NULL tienen un valor booleano falso..:

```
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
>>>
```

`ctypes` comprueba si hay NULL cuando los punteros de referencia (pero los punteros no válidos de referencia no-NULL se romperán en 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
>>>
```

Conversiones de tipos

Por lo general, los ctypes hacen un control estricto de los tipos. Esto significa que si tienes `POINTER(c_int)` en la lista `argtypes` de una función o como el tipo de un campo miembro en una definición de estructura, sólo se aceptan instancias exactamente del mismo tipo. Hay algunas excepciones a esta regla, en las que ctypes acepta otros objetos. Por ejemplo, se pueden pasar instancias de arreglo compatibles en lugar de tipos de puntero. Así, para `POINTER(c_int)`, ctypes acepta un arreglo de `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
>>>
```

Además, si se declara explícitamente que un argumento de función es de tipo puntero (como `POINTER(c_int)`) en `argtypes`, se puede pasar un objeto de tipo puntero (`c_int` en este caso) a la función. ctypes aplicará la conversión `byref()` requerida en este caso automáticamente.

Para poner un campo de tipo `POINTER` a `NULL`, puedes asignar `None`:

```
>>> bar.values = None
>>>
```

A veces se tienen instancias de tipos incompatibles. En C, puedes cambiar un tipo por otro tipo. ctypes proporciona una función `cast()` que puede ser usada de la misma manera. La estructura `Bar` definida arriba acepta punteros `POINTER(c_int)` o arreglos `c_int` para su campo `values`, pero no instancias de otros tipos:

```
>>> 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
>>>
```

Para estos casos, la función `cast()` es muy útil.

La función `cast()` puede ser usada para lanzar una instancia ctypes en un puntero a un tipo de datos ctypes diferente. `cast()` toma dos parámetros, un objeto ctypes que es o puede ser convertido en un puntero de algún tipo, y un tipo de puntero ctypes. retorna una instancia del segundo argumento, que hace referencia al mismo bloque de memoria que el primer argumento:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

Así, `cast()` puede ser usado para asignar al campo `values` de `Bar` la estructura:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
>>>
```

Tipos incompletos

Los *Tipos Incompletos* son estructuras, uniones o matrices cuyos miembros aún no están especificados. En C, se especifican mediante declaraciones a futuro, que se definen más adelante:

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

La traducción directa al código de `ctypes` sería esta, pero no funciona:

```
>>> 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
>>>
```

porque la nueva `class cell` no está disponible en la propia declaración de clase. En `ctypes`, podemos definir la clase `cell` y establecer el atributo `_fields_` más tarde, después de la declaración de clase:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

Vamos a intentarlo. Creamos dos instancias de `cell`, y dejamos que se apunten una a la otra, y finalmente seguimos la cadena de punteros unas cuantas veces:

```
>>> c1 = cell()
>>> c1.name = b"foo"
>>> c2 = cell()
>>> c2.name = b"bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

Funciones de retrollamadas (*callback*)

`ctypes` permite crear punteros de función invocables C a partir de los invocables de Python. A veces se llaman *funciones de retrollamada*.

Primero, debes crear una clase para la función de retrollamada. La clase conoce la convención de llamada, el tipo de retorno, y el número y tipos de argumentos que esta función recibirá.

La función de fábrica `CFUNCTYPE()` crea tipos para las funciones de retrollamada usando la convención de llamada `cdecl`. En Windows, la función de fábrica `WINFUNCTYPE()` crea tipos para funciones de retrollamadas usando la convención de llamadas `stdcall`.

Ambas funciones de fábrica se llaman con el tipo de resultado como primer argumento, y las funciones de llamada de retorno con los tipos de argumentos esperados como los argumentos restantes.

Presentaré un ejemplo aquí que utiliza la función `qsort()` de la biblioteca estándar de C, que se utiliza para ordenar los elementos con la ayuda de una función de retrollamada. `qsort()` se utilizará para ordenar un conjunto de números enteros:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

`qsort()` debe ser llamada con un puntero a los datos a ordenar, el número de elementos en el array de datos, el tamaño de un elemento, y un puntero a la función de comparación, la llamada de retorno. La llamada de retorno se llamará entonces con dos punteros a los ítems, y debe retornar un entero negativo si el primer ítem es más pequeño que el segundo, un cero si son iguales, y un entero positivo en caso contrario.

Así que nuestra función de retrollamada recibe punteros a números enteros, y debe retornar un número entero. Primero creamos el tipo para la función de retrollamada:

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

Para empezar, aquí hay una simple llamada que muestra los valores que se pasan:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

El resultado:

```
>>> 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
>>>
```

Ahora podemos comparar los dos artículos y obtener un resultado útil:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>>
>>> 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
>>>
```

Como podemos comprobar fácilmente, nuestro arreglo está ordenado ahora:

```
>>> for i in ia: print(i, end=" ")
...
1 5 7 33 99
>>>
```

Las funciones de fabrica pueden ser usadas como decoradores de fabrica, así que podemos escribir:

```
>>> @CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
... def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), 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
>>>
```

Nota: Asegúrate de mantener las referencias a los objetos `CFUNCTYPE()` mientras se usen desde el código C. `ctypes` no lo hace, y si no lo haces, pueden ser basura recolectada, colapsando tu programa cuando se hace una llamada.

Además, nótese que sí se llama a la función de retrollamada en un hilo creado fuera del control de Python (por ejemplo, por el código foráneo que llama a la retrollamada), `ctypes` crea un nuevo hilo Python tonto en cada invocación. Este comportamiento es correcto para la mayoría de los propósitos, pero significa que los valores almacenados con `threading.local` no sobreviven a través de diferentes llamadas de retorno, incluso cuando esas llamadas se hacen desde el mismo hilo C.

Acceder a los valores exportados de los dlls

Algunas bibliotecas compartidas no sólo exportan funciones, sino también variables. Un ejemplo en la propia biblioteca de Python es el `Py_OptimizeFlag`, un entero establecido en 0, 1, o 2, dependiendo del flag `-O` o `-OO` dado en el inicio.

`ctypes` puede acceder a valores como este con los métodos de la clase `in_dll()` del tipo. *pythonapi* es un símbolo predefinido que da acceso a la API de Python C:

```
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print(opt_flag)
c_long(0)
>>>
```

Si el intérprete se hubiera iniciado con `-O`, el ejemplo habría impreso `c_long(1)`, o `c_long(2)` si `-OO` se hubiera especificado.

Un ejemplo extendido que también demuestra el uso de punteros accediendo al puntero `PyImport_FrozenModules` exportado por Python.

Citando los documentos para ese valor:

Este puntero está inicializado para apuntar a un arreglo de registros `struct _frozen``, terminada por uno cuyos miembros son todos `NULL` o cero. Cuando se importa un módulo congelado, se busca en esta tabla. El código de terceros podría jugar con esto para proporcionar una colección creada dinámicamente de módulos congelados.

Así que manipular este puntero podría incluso resultar útil. Para restringir el tamaño del ejemplo, sólo mostramos cómo esta tabla puede ser leída con `ctypes`:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int)]
...
>>>
```

Hemos definido el tipo de datos `struct _frozen`, para que podamos obtener el puntero de la tabla:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
>>>
```

Como `tabla` es un puntero al arreglo de registros `struct_frozen`, podemos iterar sobre ella, pero sólo tenemos que asegurarnos de que nuestro bucle termine, porque los punteros no tienen tamaño. Tarde o temprano, probablemente se caerá con una violación de acceso o lo que sea, así que es mejor salir del bucle cuando le demos a la entrada `NULL`:

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
__hello__ 161
__phello__ -161
__phello__.spam 161
>>>
```

El hecho de que la Python estándar tenga un módulo congelado y un paquete congelado (indicado por el miembro tamaño negativo) no se conoce bien, sólo se usa para hacer pruebas. Pruébalo con `import __hello__` por ejemplo.

Sorpresas

Hay algunas aristas en `ctypes` en las que podrías esperar algo distinto de lo que realmente sucede.

Considere el siguiente ejemplo:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> 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. Ciertamente esperábamos que la última declaración imprimiera 3 4 1 2. ¿Qué ha pasado? Aquí están los pasos de la línea `rc.a, rc.b = rc.b, rc.a` arriba:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

Note que `temp0` y `temp1` son objetos que todavía usan el buffer interno del objeto `rc` de arriba. Así que ejecutando `rc.a = temp0` copia el contenido del buffer de `temp0` en el buffer de `rc`. Esto, a su vez, cambia el contenido de `temp1`. Por lo tanto, la última asignación `rc.b = temp1`, no tiene el efecto esperado.

Tengan en cuenta que la recuperación de subobjetos de Estructuras, Uniones y Arreglos no *copia* el subobjeto, sino que recupera un objeto contenido que accede al búfer subyacente del objeto raíz.

Otro ejemplo que puede comportarse de manera diferente a lo que uno esperaría es este:

```
>>> s = c_char_p()
>>> s.value = b"abc def ghi"
>>> s.value
b'abc def ghi'
>>> s.value is s.value
False
>>>
```

Nota: Los objetos instanciados desde `c_char_p` sólo pueden tener su valor fijado en bytes o enteros.

¿Por qué está imprimiendo `False`? Las instancias `ctypes` son objetos que contienen un bloque de memoria más algunos *descriptors* que acceden al contenido de la memoria. Almacenar un objeto Python en el bloque de memoria no almacena el objeto en sí mismo, en su lugar se almacenan los contenidos del objeto. ¡Acceder a los contenidos de nuevo construye un nuevo objeto Python cada vez!

Tipos de datos de tamaño variable

`ctypes` proporciona algo de soporte para matrices y estructuras de tamaño variable.

La función `resize()` puede ser usada para redimensionar el buffer de memoria de un objeto `ctypes` existente. La función toma el objeto como primer argumento, y el tamaño solicitado en bytes como segundo argumento. El bloque de memoria no puede hacerse más pequeño que el bloque de memoria natural especificado por el tipo de objeto, se lanza un `ValueError` si se intenta:

```
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

Esto está bien, pero ¿cómo se puede acceder a los elementos adicionales contenidos en este arreglo? Dado que el tipo todavía sabe sólo 4 elementos, obtenemos errores al acceder a otros elementos:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
...
IndexError: invalid index
>>>
```

Otra forma de utilizar tipos de datos de tamaño variable con `ctypes` es utilizar la naturaleza dinámica de Python, y (re)definir el tipo de datos después de que se conozca el tamaño requerido, caso por caso.

16.16.2 referencia ctypes

Encontrar bibliotecas compartidas

Cuando se programa en un lenguaje compilado, se accede a las bibliotecas compartidas cuando se compila/enlaza un programa, y cuándo se ejecuta el programa.

El propósito de la función `find_library()` es localizar una biblioteca de forma similar a lo que hace el compilador o el cargador en tiempo de ejecución (en plataformas con varias versiones de una biblioteca compartida se debería cargar la más reciente), mientras que los cargadores de bibliotecas `ctypes` actúan como cuando se ejecuta un programa, y llaman directamente al cargador en tiempo de ejecución.

El módulo `ctypes.util` proporciona una función que puede ayudar a determinar la biblioteca a cargar.

`ctypes.util.find_library(name)`

Intenta encontrar una biblioteca y retornar un nombre. *name* es el nombre de la biblioteca sin ningún prefijo como *lib*, sufijo como *.so*, *.dylib* o número de versión (esta es la forma usada para la opción del enlazador `posix -l`). Si no se puede encontrar ninguna biblioteca, retorna `None`.

La funcionalidad exacta depende del sistema.

En Linux, `find_library()` intenta ejecutar programas externos (`/sbin/ldconfig`, `gcc`, `objdump` y `ld`) para encontrar el archivo de la biblioteca. retorna el nombre del archivo de la biblioteca.

Distinto en la versión 3.6: En Linux, el valor de la variable de entorno `LD_LIBRARY_PATH` se utiliza cuando se buscan bibliotecas, si una biblioteca no puede ser encontrada por ningún otro medio.

Aquí hay algunos ejemplos:

```
>>> 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'
>>>
```

En OS X, `find_library()` intenta varios esquemas de nombres y rutas predefinidas para localizar la biblioteca, y retorna una ruta completa si tiene éxito:

```
>>> 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'
>>>
```

En Windows, `find_library()` busca a lo largo de la ruta de búsqueda del sistema, y retorna la ruta completa, pero como no hay un esquema de nombres predefinido, una llamada como `find_library("c")` fallará y retornará `None`.

Si envolvemos una biblioteca compartida con `ctypes`, puede ser mejor determinar el nombre de la biblioteca compartida en tiempo de desarrollo, y codificarlo en el módulo de envoltura en lugar de usar `find_library()` para localizar la biblioteca en tiempo de ejecución.

Cargando bibliotecas compartidas

Hay varias maneras de cargar las bibliotecas compartidas en el proceso Python. Una forma es instanciar una de las siguientes clases:

```
class ctypes.CDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False,
                  use_last_error=False, winmode=0)
```

Las instancias de esta clase representan bibliotecas compartidas cargadas. Las funciones de estas bibliotecas usan la convención estándar de llamada C, y se asume que retornan `int`.

On Windows creating a `CDLL` instance may fail even if the DLL name exists. When a dependent DLL of the loaded DLL is not found, a `OSError` error is raised with the message «[WinError 126] The specified module could not be found». This error message does not contain the name of the missing DLL because the Windows API does not return this information making this error hard to diagnose. To resolve this error and determine which DLL is not found, you need to find the list of dependent DLLs and determine which one is not found using Windows debugging and tracing tools.

Ver también:

[Microsoft DUMPBIN tool](#) – A tool to find DLL dependents.

```
class ctypes.OleDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False,
                   use_last_error=False, winmode=0)
```

Sólo Windows: Las instancias de esta clase representan bibliotecas compartidas cargadas, las funciones en estas bibliotecas usan la convención de llamada `stdcall`, y se asume que retornan el código específico de windows `HRESULT`. Los valores `HRESULT` contienen información que especifica si la llamada a la función falló o tuvo éxito, junto con un código de error adicional. Si el valor de retorno señala un fracaso, se eleva automáticamente un `OSError`.

Distinto en la versión 3.3: `WindowsError` solía ser lanzado.


```
class ctypes.WinDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False,
                    use_last_error=False, winmode=0)
```

Sólo Windows: Las instancias de esta clase representan bibliotecas compartidas cargadas, las funciones de estas bibliotecas usan la convención de llamada `stdcall`, y se supone que retornan `int` por defecto.

En Windows CE sólo se utiliza la convención de llamadas estándar, para mayor comodidad las `WinDLL` y `OleDLL` utilizan la convención de llamadas estándar en esta plataforma.

El termino Python *global interpreter lock* es lanzado antes de llamar a cualquier función exportada por estas librerías, y se requiere después.

```
class ctypes.PyDLL(name, mode=DEFAULT_MODE, handle=None)
```

Las instancias de esta clase se comportan como instancias `CDLL`, excepto que el GIL de Python es *no* liberado durante la llamada a la función, y después de la ejecución de la función se comprueba si esta activo el flag de error de Python. Si el flag de error esta activado, se lanza una excepción Python.

Por lo tanto, esto sólo es útil para llamar directamente a las funciones api C de Pythoni.

Todas estas clases pueden ser instanciadas llamándolas con al menos un argumento, la ruta de la biblioteca compartida. Si tienes un manejador existente de una biblioteca compartida ya cargada, se puede pasar como el parámetro llamado `handle`, de lo contrario la función `dlopen` o `LoadLibrary` de la plataforma subyacente es utilizada para cargar la biblioteca en el proceso, y obtener un manejador de la misma.

El parámetro `mode` puede utilizarse para especificar cómo se carga la biblioteca. Para más detalles, consulte la página `dlopen(3)` manpage. En Windows, `mode` es ignorado. En los sistemas posix, `RTLD_NOW` siempre se agrega, y no es configurable.

El parámetro `use_errno`, cuando se establece en `true`, habilita un mecanismo ctypes que permite acceder al número de error del sistema `errno` de forma segura. ctypes mantiene una copia local del hilo de la variable del sistema `errno`; si llamas a funciones extranjeras creadas con `use_errno=True` entonces el valor `errno` antes de la llamada a la función se intercambia con la copia privada de ctypes, lo mismo ocurre inmediatamente después de la llamada a la función.

La función `ctypes.get_errno()` retorna el valor de la copia privada de ctypes, y la función `ctypes.set_errno()` cambia la copia privada de ctypes a un nuevo valor y retorna el valor anterior.

El parámetro `use_last_error`, cuando se establece en `true`, habilita el mismo mecanismo para el código de error de Windows que es administrado por las funciones de la API de Windows `GetLastError()` y `SetLastError()`; `ctypes.get_last_error()` y `ctypes.set_last_error()` se utilizan para solicitar y cambiar la copia privada ctypes del código de error de Windows.

El parámetro `winmode` se utiliza en Windows para especificar cómo se carga la biblioteca (ya que `mode` se ignora). Toma cualquier valor que sea válido para el parámetro flags de la API de Win32 `LoadLibraryEx`. Cuando se omite, el valor por defecto es usar los flags que resultan en la carga de DLL más segura para evitar problemas como el secuestro de DLL. Pasar la ruta completa a la DLL es la forma más segura de asegurar que se cargan la biblioteca y las dependencias correctas.

Distinto en la versión 3.8: Añadido el parámetro `winmode`.

```
ctypes.RTLD_GLOBAL
```

Flag para usar como parámetro `modo`. En las plataformas en las que esta bandera no está disponible, se define como el cero entero.

```
ctypes.RTLD_LOCAL
```

Flag para usar como parámetro `modo`. En las plataformas en las que esto no está disponible, es lo mismo que `RTLD_GLOBAL`.

```
ctypes.DEFAULT_MODE
```

El modo por defecto que se utiliza para cargar las bibliotecas compartidas. En OSX 10.3, esto es `RTLD_GLOBAL`, de lo contrario es lo mismo que `RTLD_LOCAL`.

Las instancias de estas clases no tienen métodos públicos. Se puede acceder a las funciones exportadas por la biblioteca compartida como atributos o por índice. Tenga en cuenta que al acceder a la función a través de un atributo se almacena en caché el resultado y, por lo tanto, al acceder a él repetidamente se retorna el mismo objeto cada vez. Por otro lado, acceder a ella a través de un índice retorna un nuevo objeto cada vez:

```
>>> from ctypes import CDLL
>>> libc = CDLL("libc.so.6") # On Linux
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

Los siguientes atributos públicos están disponibles, su nombre comienza con un guión bajo para no chocar con los nombres de las funciones exportadas:

PyDLL._handle

El manejador del sistema usado para acceder a la biblioteca.

PyDLL._name

El nombre de la biblioteca pasado en el constructor.

Las bibliotecas compartidas también pueden ser cargadas usando uno de los objetos prefabricados, que son instancias de la clase *LibraryLoader*, ya sea llamando al método `LoadLibrary()`, o recuperando la biblioteca como atributo de la instancia de carga.

class ctypes.LibraryLoader (dlltype)

Clase que carga bibliotecas compartidas. *dlltype* debe ser uno de los tipos *CDLL*, *PyDLL*, *WinDLL*, o *OleDLL*.

`__getattr__()` tiene un comportamiento especial: Permite cargar una biblioteca compartida accediendo a ella como atributo de una instancia de carga de biblioteca. El resultado se almacena en caché, de modo que los accesos repetidos a los atributos retornan la misma biblioteca cada vez.

LoadLibrary (name)

Carga una biblioteca compartida en el proceso y la retorna. Este método siempre retorna una nueva instancia de la biblioteca.

Estos cargadores prefabricados de bibliotecas están disponibles:

ctypes.cdll

Crea instancias de *CDLL*.

ctypes.windll

Sólo Windows: Crea instancias de *WinDLL*.

ctypes.ole32

Sólo Windows: Crea instancias de *OleDLL*.

ctypes.pydll

Crea instancias de *PyDLL*.

Para acceder directamente a la API C de Python, se dispone de un objeto de biblioteca compartida de Python listo-para-usar:

ctypes.pythonapi

Una instancia de *PyDLL* que expone las funciones de la API C de Python como atributos. Ten en cuenta que se supone que todas estas funciones retornan `C int`, lo que por supuesto no siempre es cierto, así que tienes que asignar el atributo correcto `restype` para usar estas funciones.

Cargar una biblioteca a través de cualquiera de estos objetos lanza un *evento de auditoría* `ctypes.dlopen` con el argumento de cadena `name`, el nombre usado para cargar la biblioteca.

Al acceder a una función en una biblioteca cargada se lanza un evento de auditoría `ctypes.dlsym` con argumentos `library` (el objeto de la biblioteca) y `name` (el nombre del símbolo como cadena o entero).

En los casos en los que sólo está disponible el manejador de la biblioteca en lugar del objeto, al acceder a una función se produce un evento de auditoría `ctypes.dlsym/handle` con los argumentos `handle` (el manejador de la biblioteca en bruto) y `name`.

Funciones foráneas

Como se explicó en la sección anterior, se puede acceder a las funciones foráneas como atributos de las bibliotecas compartidas cargadas. Los objetos de función creados de esta forma aceptan por defecto cualquier número de argumentos, aceptan cualquier instancia de datos `ctypes` como argumentos y retornan el tipo de resultado por defecto especificado por el cargador de la biblioteca. Son instancias de una clase privada:

class `ctypes._FuncPtr`

Clase base para funciones foráneas C invocables.

Las instancias de funciones foráneas también son tipos de datos compatibles con C; representan punteros de funciones C.

Este comportamiento puede personalizarse asignando a los atributos especiales del objeto de la función foránea.

restype

Asigne un tipo de `ctypes` para especificar el tipo de resultado de la función externa. Usa `None` para `void`, una función que no retorna nada.

Es posible asignar un objeto Python invocable que no sea de tipo `ctypes`, en este caso se supone que la función retorna un `C int`, y el invocable se llamará con este entero, lo que permite un posterior procesamiento o comprobación de errores. El uso de esto está obsoleto, para un postprocesamiento más flexible o para la comprobación de errores utilice un tipo de datos `ctypes` como `restype` y asigne un invocable al atributo `errcheck`.

argtypes

Asigne una tupla de tipos `ctypes` para especificar los tipos de argumentos que acepta la función. Las funciones que utilizan la convención de llamada `stdcall` sólo pueden ser llamadas con el mismo número de argumentos que la longitud de esta tupla; las funciones que utilizan la convención de llamada C aceptan también argumentos adicionales no especificados.

Cuando se llama a una función foránea, cada argumento real se pasa al método de la clase `from_param()` de los elementos de la tupla `argtypes`, este método permite adaptar el argumento real a un objeto que la función externa acepta. Por ejemplo, un elemento `c_char_p` de la tupla `argtypes` convertirá una cadena pasada como argumento en un objeto de bytes utilizando reglas de conversión `ctypes`.

Nuevo: Ahora es posible poner en `argtypes` elementos que no son de tipo `ctypes`, pero cada elemento debe tener un método `from_param()` que retorne un valor utilizable como argumento (entero, cadena, instancia `ctypes`). Esto permite definir adaptadores que pueden adaptar objetos personalizados como parámetros de la función.

errcheck

Asigne una función Python u otra llamada a este atributo. El invocable será llamado con tres o más argumentos:

callable (*result*, *func*, *arguments*)

result es lo que retorna la función externa, como se especifica en el atributo `restype`.

func es el propio objeto de la función foránea, lo que permite reutilizar el mismo objeto invocable para comprobar o postprocesar los resultados de varias funciones.

arguments es una tupla que contiene los parámetros originalmente pasados a la llamada de la función, esto permite especializar el comportamiento en los argumentos utilizados.

El objeto que retorna esta función será retornado por la llamada de la función foránea, pero también puede comprobar el valor del resultado y hacer una excepción si la llamada de la función foránea ha fallado.

exception `ctypes.ArgumentError`

Esta excepción se lanza cuando una llamada a una función foránea no puede convertir uno de los argumentos pasados.

En Windows, cuando una llamada a una función foránea plantea una excepción de sistema (por ejemplo, debido a una violación de acceso), será capturada y sustituida por una excepción Python adecuada. Además, un evento de auditoría `ctypes.seh_exception` con el argumento `code` será levantado, permitiendo que un gancho de auditoría reemplace la excepción con la suya propia.

Algunas formas de invocar llamadas a funciones foráneas pueden lanzar un evento de auditoría `ctypes.call_function` con los argumentos `function pointer` y `arguments`.

Prototipos de funciones

Las funciones foráneas también pueden crearse mediante la instanciación de prototipos de funciones. Los prototipos de funciones son similares a los prototipos de funciones en C; describen una función (tipo de retorno, tipos de argumentos, convención de llamada) sin definir una implementación. Las funciones de fábrica deben ser llamadas con el tipo de resultado deseado y los tipos de argumento de la función, y pueden ser usadas como fábricas de decoradores, y como tales, ser aplicadas a las funciones a través de la sintaxis `@wrapper`. Ver [Funciones de retrollamadas \(callback\)](#) para ejemplos.

`ctypes.CFUNCTYPE` (*restype*, **argtypes*, *use_errno*=False, *use_last_error*=False)

El prototipo de función retornado crea funciones que usan la convención de llamada C estándar. La función liberará el GIL durante la llamada. Si *use_errno* se configura a true, la copia privada de `ctypes` de la variable del sistema `errno` se intercambia con el valor real `errno` antes y después de la llamada; *use_last_error* hace lo mismo con el código de error de Windows.

`ctypes.WINFUNCTYPE` (*restype*, **argtypes*, *use_errno*=False, *use_last_error*=False)

Sólo Windows: El prototipo de función retornado crea funciones que usan la convención de llamada `stdcall`, excepto en Windows CE donde `WINFUNCTYPE()` es lo mismo que `CFUNCTYPE()`. La función lanzará el GIL durante la llamada. *use_errno* y *use_last_error* tienen el mismo significado que arriba.

`ctypes.PYFUNCTYPE` (*restype*, **argtypes*)

El prototipo de función retornado crea funciones que usan la convención de llamadas de Python. La función *no* liberará el GIL durante la llamada.

Los prototipos de funciones creados por estas funciones de fábrica pueden ser instanciados de diferentes maneras, dependiendo del tipo y el número de los parámetros en la llamada:

prototype (*address*)

Retorna una función foránea en la dirección especificada que debe ser un número entero.

prototype (*callable*)

Crear una función de llamada C (una función de retrollamada) a partir de un *callable* Python.

prototype (*func_spec*[, *paramflags*])

Retorna una función foránea exportada por una biblioteca compartida. *func_spec* debe ser un 2-tupla (*name_or_ordinal*, *library*). El primer elemento es el nombre de la función exportada como cadena, o el ordinal de la función exportada como entero pequeño. El segundo elemento es la instancia de la biblioteca compartida.

prototype (*vtbl_index*, *name*[, *paramflags*[, *iid*]])

Retorna una función foránea que llamará a un método COM. *vtbl_index* es el índice de la tabla de funciones virtuales, un pequeño entero no negativo. *name* es el nombre del método COM. *iid* es un puntero opcional para el identificador de la interfaz que se utiliza en el informe de errores extendido.

Los métodos COM usan una convención especial de llamadas: Requieren un puntero a la interfaz COM como primer argumento, además de los parámetros que se especifican en la tupla *argtypes*.

El parámetro opcional *paramflags* crea envoltorios de funciones foráneas con mucha más funcionalidad que las características descritas anteriormente.

paramflags deben ser una tupla de la misma longitud que *argtypes*.

Cada elemento de esta tupla contiene más información sobre un parámetro, debe ser una tupla que contenga uno, dos o tres elementos.

El primer elemento es un entero que contiene una combinación de flags de dirección para el parámetro:

- 1 Especifica un parámetro de entrada a la función.
- 2 Parámetro de salida. La función foránea rellena un valor.
- 4 Parámetro de entrada que por defecto es el cero entero.

El segundo elemento opcional es el nombre del parámetro como cadena. Si se especifica esto, se puede llamar a la función foránea con parámetros con nombre.

El tercer elemento opcional es el valor por defecto de este parámetro.

Este ejemplo demuestra cómo envolver la función `MessageBoxW` de Windows para que soporte los parámetros por defecto y los argumentos con nombre. La declaración C del archivo de cabecera de Windows es esta:

```
WINUSERAPI int WINAPI
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

Aquí está el envoltorio con `ctypes`:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCWSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", "Hello from_
↳ctypes"), (1, "flags", 0)
>>> MessageBox = prototype(("MessageBoxW", windll.user32), paramflags)
```

La función foránea de `MessageBox` puede ser llamada de esta manera:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

Un segundo ejemplo demuestra los parámetros de salida. La función `GetWindowRect` de win32 retorna las dimensiones de una ventana especificada copiándolas en la estructura `RECT` que la persona que llama tiene que suministrar. Aquí está la declaración C:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
    HWND hWnd,
    LPRECT lpRect);
```

Aquí está el envoltorio con `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)
>>>
```

Las funciones con parámetros de salida retornarán automáticamente el valor del parámetro de salida si hay uno solo, o una tupla que contiene los valores del parámetro de salida cuando hay más de uno, por lo que la función `GetWindowRect` retorna ahora una instancia `RECT`, cuando se llama.

Los parámetros de salida pueden combinarse con el protocolo `errcheck` para hacer un mayor procesamiento de la salida y la comprobación de errores. La función api de win32 `GetWindowRect` retorna un `BOOL` para señalar el éxito o el fracaso, por lo que esta función podría hacer la comprobación de errores, y plantea una excepción cuando la llamada api ha fallado:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

Si la función `errcheck` retorna la tupla de argumentos que recibe sin cambios, `ctypes` continúa el procesamiento normal que hace en los parámetros de salida. Si quieres retornar una tupla de coordenadas de ventana en lugar de una instancia `RECT`, puedes recuperar los campos de la función y retornarlos en su lugar, el procesamiento normal ya no tendrá lugar:

```
>>> 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
>>>
```

Funciones de utilidad

`ctypes.addressof(obj)`

Retorna la dirección del buffer de memoria como un entero. *obj* debe ser una instancia de tipo `ctypes`.

Lanza un *evento de auditoría* `ctypes.addressof` con el argumento *obj*.

`ctypes.alignment(obj_or_type)`

Retorna los requerimientos de alineación de un tipo de `ctypes`. *obj_or_type* debe ser un tipo o instancia `ctypes`.

`ctypes.byref(obj[, offset])`

Retorna un puntero ligero a *obj*, que debe ser un ejemplo de un tipo de `ctypes`. *offset* es por defecto cero, y debe ser un entero que se añadirá al valor del puntero interno.

`byref(obj, offset)` corresponde a este código C:

```
((char *)&obj) + offset)
```

El objeto retornado sólo puede ser utilizado como un parámetro de llamada de función foránea. Se comporta de manera similar a `pointer(obj)`, pero la construcción es mucho más rápida.

`ctypes.cast(obj, type)`

Esta función es similar a la del operador de reparto en C. retorna una nueva instancia de *type* que apunta al mismo bloque de memoria que *obj*. *type* debe ser un tipo de puntero, y *obj* debe ser un objeto que pueda ser interpretado como un puntero.

`ctypes.create_string_buffer(init_or_size, size=None)`

Esta función crea un búfer de caracteres mutables. El objeto retornado es un arreglo de `ctypes` de `c_char`.

init_or_size debe ser un número entero que especifique el tamaño del arreglo, o un objeto de bytes que se utilizará para inicializar los elementos del arreglo.

Si se especifica un objeto bytes como primer argumento, el buffer se hace un elemento más grande que su longitud, de modo que el último elemento del arreglo es un carácter de terminación NUL. Se puede pasar un entero como segundo argumento que permite especificar el tamaño del arreglo si no se debe utilizar la longitud de los bytes.

Lanza un *evento de auditoría* `ctypes.create_string_buffer` con argumentos *init*, *size*.

`ctypes.create_unicode_buffer (init_or_size, size=None)`

Esta función crea un búfer de caracteres unicode mutable. El objeto retornado es un arreglo de ctypes de `c_wchar`.

`init_or_size` debe ser un entero que especifique el tamaño del arreglo, o una cadena que se utilizará para inicializar los elementos del arreglo.

Si se especifica una cadena como primer argumento, el búfer se hace un elemento más grande que la longitud de la cadena, de modo que el último elemento del arreglo es un carácter de terminación NUL. Se puede pasar un entero como segundo argumento que permite especificar el tamaño del arreglo si no se debe utilizar la longitud de la cadena.

Lanza un *evento de auditoría* `ctypes.create_unicode_buffer` con argumentos `init`, `size`.

`ctypes.DllCanUnloadNow ()`

Sólo Windows: Esta función es un gancho que permite implementar servidores COM en proceso con ctypes. Se llama desde la función `DllCanUnloadNow` que la extensión `_ctypes.dll` exporta.

`ctypes.DllGetClassObject ()`

Sólo Windows: Esta función es un gancho que permite implementar servidores COM en proceso con ctypes. Se llama desde la función `DllGetClassObject` que la extensión `_ctypes` exporta.

`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`.

La funcionalidad exacta depende del sistema.

`ctypes.util.find_msvcr ()`

Sólo Windows: retorna el nombre de archivo de la biblioteca de tiempo de ejecución de VC usada por Python, y por los módulos de extensión. Si no se puede determinar el nombre de la biblioteca, se retorna `None`.

Si necesita liberar memoria, por ejemplo, asignada por un módulo de extensión con una llamada al `free(void *)`, es importante que utilice la función en la misma biblioteca que asignó la memoria.

`ctypes.FormatError ([code])`

Sólo Windows: retorna una descripción textual del código de error *code*. Si no se especifica ningún código de error, se utiliza el último código de error llamando a la función de api de Windows `GetLastError`.

`ctypes.GetLastError ()`

Sólo Windows: retorna el último código de error establecido por Windows en el hilo de llamada. Esta función llama directamente a la función `GetLastError()` de Windows, no retorna la copia ctypes-private del código de error.

`ctypes.get_errno ()`

Retorna el valor actual de la copia ctypes-private de la variable de sistema *errno* en el hilo de llamada.

Lanza un *evento de auditoría* `ctypes.get_errno` sin argumentos.

`ctypes.get_last_error ()`

Sólo Windows: retorna el valor actual de la copia ctypes-private de la variable de sistema `LastError` en el hilo de llamada.

Lanza un *evento de auditoría* `ctypes.get_last_error` sin argumentos.

`ctypes.memmove (dst, src, count)`

Igual que la función de la biblioteca estándar de C *memmove*: copia *count* bytes de *src* a *dst*. *dst* y *src* deben ser enteros o instancias ctypes que pueden ser convertidos en punteros.

`ctypes.memset (dst, c, count)`

Igual que la función de la biblioteca estándar de C *memset*: llena el bloque de memoria en la dirección *dst* con *count* bytes de valor *c*. *dst* debe ser un número entero que especifique una dirección, o una instancia ctypes.

`ctypes.POINTER (type)`

Esta función de fábrica crea y retorna un nuevo tipo de puntero ctypes. Los tipos de puntero se almacenan en

caché y se reutilizan internamente, por lo que llamar a esta función repetidamente es barato. *type* debe ser un tipo ctypes.

ctypes.pointer(*obj*)

Esta función crea una nueva instancia de puntero, apuntando a *obj*. El objeto retornado es del tipo `POINTER(tipo(obj))`.

Nota: Si sólo quieres pasar un puntero a un objeto a una llamada de función foránea, deberías usar `byref(obj)` que es mucho más rápido.

ctypes.resize(*obj*, *size*)

Esta función redimensiona el búfer de memoria interna de *obj*, que debe ser una instancia de tipo ctypes. No es posible hacer el buffer más pequeño que el tamaño nativo del tipo de objetos, como lo indica `size of (type(obj))`, pero es posible agrandar el buffer.

ctypes.set_errno(*value*)

Poner el valor actual de la copia ctypes-private de la variable del sistema *errno* en el hilo de llamada a *valor* y retornar el valor anterior.

Lanza un *evento de auditoría* `ctypes.set_errno` con argumento *errno*.

ctypes.set_last_error(*value*)

Sólo para Windows: pone el valor actual de la copia ctypes-private de la variable del sistema `LastError` en el hilo de llamada a *valor* y retorna el valor anterior.

Lanza un *evento de auditoría* `ctypes.set_last_error` con argumento *error*.

ctypes.sizeof(*obj_or_type*)

Retorna el tamaño en bytes de un buffer de memoria tipo ctypes o instancia. Hace lo mismo que el operador `C sizeof`.

ctypes.string_at(*address*, *size*=-1)

Esta función retorna la cadena C que comienza en la dirección de memoria *address* como un objeto de bytes. Si se especifica el tamaño, se utiliza como tamaño, de lo contrario se asume que la cadena tiene un final cero.

Lanza un *evento de auditoría* `ctypes.string_at` con argumentos *address*, *size*.

ctypes.WinError(*code*=None, *descr*=None)

Sólo para Windows: esta función es probablemente la cosa peor nombrada de los ctypes. Crea una instancia de `OSError`. Si no se especifica el *code*, se llama a `GetLastError` para determinar el código de error. Si no se especifica *descr*, se llama a `FormatError`()` para obtener una descripción textual del error.

Distinto en la versión 3.3: Una instancia de *WindowsError* solía ser creada.

ctypes.wstring_at(*address*, *size*=-1)

Esta función retorna la cadena de caracteres anchos que comienza en la dirección de memoria *address* como una cadena. Si se especifica *size*, se utiliza como el número de caracteres de la cadena, de lo contrario se asume que la cadena tiene un final cero.

Lanza un *evento de auditoría* `ctypes.wstring_at` con argumentos *address*, *size*.

Tipos de datos

class ctypes._CData

Esta clase no pública es la clase de base común de todos los tipos de datos de los ctypes. Entre otras cosas, todas las instancias de tipo ctypes contienen un bloque de memoria que contiene datos compatibles con C; la dirección del bloque de memoria es retornada por la función de ayuda `addressof()`. Otra variable de instancia se expone como `_objetos`; ésta contiene otros objetos de Python que deben mantenerse vivos en caso de que el bloque de memoria contenga punteros.

Métodos comunes de tipos de datos ctypes, estos son todos métodos de clase (para ser exactos, son métodos del *metaclass*):

from_buffer(*source*[, *offset*])

Este método retorna una instancia ctypes que comparte el buffer del objeto *source*. El objeto *source* debe

soportar la interfaz del buffer de escritura. El parámetro opcional *offset* especifica un offset en el buffer de la fuente en bytes; el valor por defecto es cero. Si el buffer de la fuente no es lo suficientemente grande se lanza un *ValueError*.

Lanza un *evento de auditoría* `ctypes.cdata/buffer` con argumentos `pointer`, `size`, `offset`.

from_buffer_copy (*source*[, *offset*])

Este método crea una instancia `ctypes`, copiando el buffer del buffer de objetos *source* que debe ser legible. El parámetro opcional *offset* especifica un offset en el buffer de origen en bytes; el valor por defecto es cero. Si el buffer de fuente no es lo suficientemente grande se lanza un *ValueError*.

Lanza un *evento de auditoría* `ctypes.cdata/buffer` con argumentos `pointer`, `size`, `offset`.

from_address (*address*)

Este método retorna una instancia de tipo `ctypes` utilizando la memoria especificada por *address* que debe ser un entero.

Este método, y otros que indirectamente llaman a este método, lanzan un *evento de auditoría* `ctypes.cdata` con argumento *address*.

from_param (*obj*)

Este método adapta el *obj* a un tipo de `ctypes`. Se llama con el objeto real usado en una llamada a una función externa cuando el tipo está presente en la tupla `argtypes` de la función foránea; debe retornar un objeto que pueda ser usado como parámetro de llamada a la función.

Todos los tipos de datos `ctypes` tienen una implementación por defecto de este método de clase que normalmente retorna *obj* si es una instancia del tipo. Algunos tipos aceptan también otros objetos.

in_dll (*library*, *name*)

Este método retorna una instancia de tipo `ctypes` exportada por una biblioteca compartida. *name* es el nombre del símbolo que exporta los datos, *library* es la biblioteca compartida cargada.

Variables de instancia común de los tipos de datos de `ctypes`:

`_b_base_`

A veces, las instancias de datos `ctypes` no poseen el bloque de memoria que contienen, sino que comparten parte del bloque de memoria de un objeto base. El miembro de sólo lectura `_b_base_` es el objeto raíz `ctypes` que posee el bloque de memoria.

`_b_needsfree_`

Esta variable de sólo lectura es verdadera cuando la instancia de datos `ctypes` ha sido asignada a el propio bloque de memoria, falsa en caso contrario.

`_objects`

Este miembro es `None` o un diccionario que contiene objetos de Python que deben mantenerse vivos para que el contenido del bloque de memoria sea válido. Este objeto sólo se expone para su depuración; nunca modifique el contenido de este diccionario.

Tipos de datos fundamentales

class `ctypes._SimpleCData`

Esta clase no pública es la clase base de todos los tipos de datos de `ctypes` fundamentales. Se menciona aquí porque contiene los atributos comunes de los tipos de datos de `ctypes` fundamentales. `_SimpleCData` es una subclase de `_CData`, por lo que hereda sus métodos y atributos. Los tipos de datos `ctypes` que no son y no contienen punteros ahora pueden ser archivados.

Los instancias tienen un solo atributo:

`value`

Este atributo contiene el valor real de la instancia. Para los tipos enteros y punteros, es un entero, para los tipos de caracteres, es un objeto o cadena de bytes de un solo carácter, para los tipos de punteros de caracteres es un objeto o cadena de bytes de Python.

Cuando el atributo `value` se recupera de una instancia `ctypes`, normalmente se retorna un nuevo objeto cada vez. `ctypes` *no* implementa el retorno del objeto original, siempre se construye un nuevo objeto. Lo mismo ocurre con todas las demás instancias de objetos `ctypes`.

Los tipos de datos fundamentales, cuando se retornan como resultados de llamadas de funciones foráneas, o, por ejemplo, al recuperar miembros de campo de estructura o elementos de arreglos, se convierten de forma transparente a tipos nativos de Python. En otras palabras, si una función externa tiene un `restype` de `c_char_p`, siempre recibirá un objeto de bytes Python, *no* una instancia de `c_char_p`.

Las subclases de los tipos de datos fundamentales *no* heredan este comportamiento. Así, si una función externa `restype` es una subclase de `c_void_p`, recibirás una instancia de esta subclase desde la llamada a la función. Por supuesto, puedes obtener el valor del puntero accediendo al atributo `value`.

Estos son los tipos de datos fundamentales de `ctypes`:

class `ctypes.c_byte`

Representa el tipo de datos C `signed char`, e interpreta el valor como un entero pequeño. El constructor acepta un inicializador de entero opcional; no se hace ninguna comprobación de desbordamiento.

class `ctypes.c_char`

Representa el tipo de datos C `char`, e interpreta el valor como un solo carácter. El constructor acepta un inicializador de cadena opcional, la longitud de la cadena debe ser exactamente un carácter.

class `ctypes.c_char_p`

Representa el tipo de datos C `char *` cuando apunta a una cadena terminada en cero. Para un puntero de carácter general que también puede apuntar a datos binarios, se debe usar `POINTER(c_char)`. El constructor acepta una dirección entera, o un objeto de bytes.

class `ctypes.c_double`

Representa el tipo de datos C `double`. El constructor acepta un inicializador flotante opcional.

class `ctypes.c_longdouble`

Representa el tipo de datos C `long double`. El constructor acepta un inicializador flotante opcional. En las plataformas donde `sizeof(long double) == sizeof(double)` es un alias de `c_double`.

class `ctypes.c_float`

Representa el tipo de datos C `float`. El constructor acepta un inicializador flotante opcional.

class `ctypes.c_int`

Representa el tipo de datos C `signed int`. El constructor acepta un inicializador entero opcional; no se hace ninguna comprobación de desbordamiento. En plataformas donde `sizeof(int) == sizeof(long)` es un alias de `c_long`.

class `ctypes.c_int8`

Representa el tipo de datos C 8-bit `signed int`. Normalmente un alias para `c_byte`.

class `ctypes.c_int16`

Representa el tipo de datos C 16-bit `signed int`. Normalmente un alias para `c_short`.

class `ctypes.c_int32`

Representa el tipo de datos C 32-bit `signed int`. Normalmente un alias para `c_int`.

class `ctypes.c_int64`

Representa el tipo de datos C 64-bit `signed int`. Normalmente un alias para `c_longlong`.

class `ctypes.c_long`

Representa el tipo de datos C `signed long`. El constructor acepta un inicializador entero opcional; no se hace ninguna comprobación de desbordamiento.

class `ctypes.c_longlong`

Representa el tipo de datos C significado `long long`. El constructor acepta un inicializador entero opcional; no se hace ninguna comprobación de desbordamiento.

class `ctypes.c_short`

Representa el tipo de datos C `signed short`. El constructor acepta un inicializador entero opcional; no se hace ninguna comprobación de desbordamiento.

class `ctypes.c_size_t`
Representa el tipo de datos `C size_t`.

class `ctypes.c_ssize_t`
Representa el tipo de datos `C ssize_t`.
Nuevo en la versión 3.2.

class `ctypes.c_ubyte`
Representa el tipo de datos `C unsigned char`, interpreta el valor como un entero pequeño. El constructor acepta un inicializador entero opcional; no se hace ninguna comprobación de desbordamiento.

class `ctypes.c_uint`
Representa el tipo de datos `C unsigned int`. El constructor acepta un inicializador entero opcional; no se hace ninguna comprobación de desbordamiento. En plataformas donde `sizeof(int) == sizeof(long)` es un alias para `c_ulong`.

class `ctypes.c_uint8`
Representa el tipo de datos `C 8-bit unsigned int`. Normalmente un alias para `c_ubyte`.

class `ctypes.c_uint16`
Representa el tipo de datos `C 16-bit unsigned int`. Normalmente un alias para `c_ushort`.

class `ctypes.c_uint32`
Representa el tipo de datos `C 32-bit unsigned int`. Normalmente un alias para `c_uint`.

class `ctypes.c_uint64`
Representa el tipo de datos `C 64-bit unsigned int`. Normalmente un alias para `c_ulonglong`.

class `ctypes.c_ulong`
Representa el tipo de datos `C unsigned long`. El constructor acepta un inicializador entero opcional; no se hace ninguna comprobación de desbordamiento.

class `ctypes.c_ulonglong`
Representa el tipo de datos `C unsigned long long`. El constructor acepta un inicializador entero opcional; no se hace ninguna comprobación de desbordamiento.

class `ctypes.c_ushort`
Representa el tipo de datos `C unsigned short`. El constructor acepta un inicializador entero opcional; no se hace ninguna comprobación de desbordamiento.

class `ctypes.c_void_p`
Representa el tipo `C void *`. El valor se representa como un entero. El constructor acepta un inicializador entero opcional.

class `ctypes.c_wchar`
Representa el tipo de datos `C wchar_t`, e interpreta el valor como una cadena unicode de un solo carácter. El constructor acepta un inicializador de cadena opcional, la longitud de la cadena debe ser exactamente de un carácter.

class `ctypes.c_wchar_p`
Representa el tipo de datos `C wchar_t *`, que debe ser un puntero a una cadena de caracteres anchos con terminación cero. El constructor acepta una dirección entera, o una cadena.

class `ctypes.c_bool`
Representa el tipo de datos `C bool` (más exactamente, `_Bool` de C99). Su valor puede ser `True` o `False`, y el constructor acepta cualquier objeto que tenga un valor verdadero.

class `ctypes.HRESULT`
Sólo Windows: Representa un valor `HRESULT`, que contiene información de éxito o error para una llamada de función o método.

class `ctypes.py_object`
Representa el tipo de datos `C PyObject *`. Llamar esto sin un argumento crea un puntero `NULL PyObject *`.

El módulo `ctypes.wintypes` proporciona otros tipos de datos específicos de Windows, por ejemplo `HWND`, `LPARAM`, o `DWORD`. Algunas estructuras útiles como `MSG` o `RECT` también están definidas.

Tipos de datos estructurados

class `ctypes.Union` (*args, **kw)

Clase base abstracta para uniones en orden de bytes nativos.

class `ctypes.BigEndianStructure` (*args, **kw)

Clase base abstracta para estructuras en orden de bytes *big endian*.

class `ctypes.LittleEndianStructure` (*args, **kw)

Clase base abstracta para estructuras en orden de bytes *little endian*.

Las estructuras con un orden de bytes no nativo no pueden contener campos de tipo puntero, o cualquier otro tipo de datos que contenga campos de tipo puntero.

class `ctypes.Structure` (*args, **kw)

Clase base abstracta para estructuras en orden de bytes *native*.

La estructura concreta y los tipos de unión deben crearse subclasificando uno de estos tipos, y al menos definir una variable de clase `__fields__`. `ctypes` creará *descriptors* que permitan leer y escribir los campos por accesos directos de atributos. Estos son los

`__fields__`

Una secuencia que define los campos de estructura. Los elementos deben ser de 2 o 3 tuplas. El primer ítem es el nombre del campo, el segundo ítem especifica el tipo de campo; puede ser cualquier tipo de datos `ctypes`.

Para los campos de tipo entero como `c_int`, se puede dar un tercer elemento opcional. Debe ser un pequeño entero positivo que defina el ancho de bit del campo.

Los nombres de los campos deben ser únicos dentro de una estructura o unión. Esto no se comprueba, sólo se puede acceder a un campo cuando los nombres se repiten.

Es posible definir la variable de clase `__fields__` después de la sentencia de clase que define la subclase Estructura, esto permite crear tipos de datos que se refieren directa o indirectamente a sí mismos:

```
class List(Structure):
    pass
List.__fields__ = [("pNext", POINTER(List)),
                  ...
                  ]
```

Sin embargo, la variable de clase `__fields__` debe ser definida antes de que el tipo sea usado por primera vez (se crea una instancia, se llama a `sizeof()`, y así sucesivamente). Las asignaciones posteriores a la variable de clase `__fields__` lanzarán un `AttributeError`.

Es posible definir subclases de tipos de estructura, que heredan los campos de la clase base más el `__fields__` definido en la subclase, si existe.

`__pack__`

Un pequeño entero opcional que permite anular la alineación de los campos de estructura en la instancia. `__pack__` ya debe estar definido cuando se asigna `__fields__`, de lo contrario no tendrá ningún efecto.

`__anonymous__`

Una secuencia opcional que enumera los nombres de los campos sin nombre (anónimos). `__anonymous__` debe estar ya definida cuando se asigna `__fields__`, de lo contrario no tendrá ningún efecto.

Los campos listados en esta variable deben ser campos de tipo estructura o unión. `ctypes` creará *descriptores* en el tipo de estructura que permitan acceder a los campos anidados directamente, sin necesidad de crear el campo de estructura o unión.

Aquí hay un tipo de ejemplo (Windows):

```

class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]

```

La estructura `TYPEDESC` describe un tipo de datos COM, el campo `vt` especifica cuál de los campos de unión es válido. Como el campo `u` está definido como campo anónimo, ahora es posible acceder a los miembros directamente desde la instancia `TYPEDESC`. `td.lptdesc` y `td.u.lptdesc` son equivalentes, pero el primero es más rápido ya que no necesita crear una instancia de unión temporal:

```

td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)

```

Es posible definir subclases de estructuras, que heredan los campos de la clase base. Si la definición de la subclase tiene una variable `_fields_` separada, los campos especificados en ella se añaden a los campos de la clase base.

Los constructores de estructuras y uniones aceptan tanto argumentos posicionales como de palabras clave. Los argumentos posicionales se usan para inicializar los campos de los miembros en el mismo orden en que aparecen en `_fields_`. Los argumentos de palabras clave en el constructor se interpretan como asignaciones de atributos, por lo que inicializarán `_fields_` con el mismo nombre, o crearán nuevos atributos para nombres no presentes en `_fields_`.

Arreglos y punteros

class `ctypes.Array` (*args)

Clase base abstracta para arreglos.

La forma recomendada de crear tipos de arreglos concretos es multiplicando cualquier tipo de datos `ctypes` con un número entero positivo. Alternativamente, puedes subclasificar este tipo y definir las variables de clase `_length_` y `_type_`. Los elementos del arreglo pueden ser leídos y escritos usando subíndices estándar y accesos slice; para las lecturas slice, el objeto resultante *no es* en sí mismo un `Array`.

length

Un número entero positivo que especifica el número de elementos del conjunto. Los subíndices fuera de rango dan como resultado un `IndexError`. Será retornado por `len()`.

type

Especifica el tipo de cada elemento del arreglo.

Los constructores de subclases de arreglos aceptan argumentos posicionales, usados para inicializar los elementos en orden.

class `ctypes._Pointer`

Clase base, privada y abstracta para punteros.

Los tipos de punteros concretos se crean llamando a `POINTER()` con el tipo que será apuntado; esto se hace automáticamente por `pointer()`.

Si un puntero apunta a un arreglo, sus elementos pueden ser leídos y escritos usando accesos de subíndices y cortes estándar. Los objetos punteros no tienen tamaño, así que `len()` lanzará un `TypeError`. Los subíndices negativos se leerán de la memoria *antes* que el puntero (como en C), y los subíndices fuera de rango probablemente se bloqueen con una violación de acceso (si tienes suerte).

type

Especifica el tipo apuntado.

contents

Returns the object to which to pointer points. Assigning to this attribute changes the pointer to point to the assigned object.

Ejecución concurrente

Los módulos descritos en este capítulo proveen soporte para la ejecución concurrente de código. La elección de qué herramienta utilizar depende de la tarea a ejecutar (vinculada a CPU o vinculada a E/S) y del estilo preferido de desarrollo (multi-tarea cooperativa o multi-tarea apropiativa). A continuación se muestra un resumen:

17.1 `threading` — Paralelismo basado en hilos

Código fuente: [Lib/threading.py](#)

Este módulo construye interfaces de hilado de alto nivel sobre el módulo de más bajo nivel `_thread`. Ver también el módulo `queue`.

Distinto en la versión 3.7: Este módulo solía ser opcional, ahora está siempre disponible.

Nota: Aunque no están listados en lo que sigue, los nombres en `camelCase` usados para algunos de los métodos y funciones de la versión Python 2.x todavía son soportados por este módulo.

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` or `concurrent.futures.ProcessPoolExecutor`. However, `threading` is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

Este módulo define las siguientes funciones:

`threading.active_count()`

Retorna el número de objetos `Thread` actualmente con vida. La cuenta retornada es igual al largo de la lista retornada por `enumerate()`.

`threading.current_thread()`

Retorna el objeto `Thread` actual, correspondiente al hilo de control del invocador. Si el hilo de control del invocador no fue creado a través del módulo `threading`, se retorna un objeto hilo `dummy` con funcionalidad limitada.

`threading.excepthook (args, /)`

Gestiona una excepción lanzada por `Thread.run()`.

El argumento *args* posee los siguientes atributos:

- *exc_type*: Tipo de la excepción.
- *exc_value*: Valor de la excepción, puede ser `None`.
- *exc_traceback*: Rastreo de la excepción, puede ser `None`.
- *thread*: El hilo que ha lanzado la excepción, puede ser `None`.

Si *exc_type* es `SystemExit`, la excepción es silenciosamente ignorada. De otro modo, la excepción se imprime en `sys.stderr`.

Si esta función lanza una excepción, se llama a `sys.excepthook()` para manejarla.

`threading.excepthook()` se puede sobrescribir para controlar cómo se gestionan las excepciones levantadas por `Thread.run()`.

Guarda *exc_value* usando un *hook* personalizado puede crear un ciclo de referencias. Debe ser aclarado explícitamente que se rompa el ciclo de referencias cuando la excepción ya no se necesite.

Guarda *thread* usando un *hook* personalizado puede resucitarlo si se asigna a un objeto que esté siendo finalizado. Evítese que *thread* sea almacenado después de que el *hook* personalizado se complete para evitar resucitar objetos.

Ver también:

`sys.excepthook()` gestiona excepciones no capturadas.

Nuevo en la versión 3.8.

`threading.get_ident()`

Retorna el “identificador de hilo” del hilo actual. Éste es un entero distinto de cero. Su valor no tiene un significado directo; ha sido pensado como una *cookie* mágica para usarse, por ejemplo, en indexar un diccionario con datos específicos del hilo. Los identificadores de hilo pueden ser reciclados cuando se abandona un hilo y se crea otro hilo.

Nuevo en la versión 3.3.

`threading.get_native_id()`

Retorna la ID de Hilo (*Thread ID*) nativo integral del hilo actual asignado por el *kernel*. Ella es un entero distinto de cero. Su valor puede utilizarse para identificar de forma única a este hilo en particular a través de todo el sistema (hasta que el hilo termine, luego de lo cual el valor puede ser reciclado por el SO).

Disponibilidad: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX.

Nuevo en la versión 3.8.

`threading.enumerate()`

Retorna una lista de todos los objetos tipo `Thread` actualmente con vida. La lista incluye hilos demonio, objetos hilo *dummy* creados por `current_thread()`, y el hilo principal. Excluye hilos terminados e hilos que todavía no hayan sido iniciados.

`threading.main_thread()`

Retorna el objeto `Thread` principal. En condiciones normales, el hilo principal es el hilo desde el que fue inicializado el intérprete de Python.

Nuevo en la versión 3.4.

`threading.settrace (func)`

Establece una función de traza para todos los hilos iniciados desde el módulo `threading`. La *func* se pasará a `sys.settrace()` por cada hilo, antes de que su método `run()` sea llamado.

`threading.setprofile (func)`

Establece una función de perfil para todos los hilos iniciados desde el módulo `threading`. La *func* se pasará a `sys.setprofile()` por cada hilo, antes de que se llame a su método `run()`.

`threading.stack_size([size])`

Retorna el tamaño de pila usado para crear nuevos hilos. El argumento opcional *size* (tamaño) especifica el tamaño de pila a ser utilizado para hilos creados posteriormente, y debe ser 0 (usar el valor por defecto de la plataforma o el configurado) o un valor entero positivo de al menos 32.768 (32KiB). Si no se especifica *size*, se usará 0. Si no existe soporte para cambiar el tamaño de pila, se lanzará un `RuntimeError`. Si el tamaño de pila especificado es inválido, se lanzará un `ValueError` y el tamaño de pila no será modificado. El tamaño mínimo de pila actualmente soportado es de 32KiB para garantizar suficiente espacio de pila para el intérprete mismo. Nótese que algunas plataformas pueden tener restricciones particulares de valores para tamaños de pila, como requerir un tamaño de pila > 32KiB, o requerir una asignación en múltiplos del tamaño de página de la memoria del sistema. Debe consultarse la documentación de cada plataforma para mayor información (páginas de 4KiB son comunes; se recomienda el uso de múltiplos de 4096 para el tamaño de pila en ausencia de información más específica)

Disponibilidad: Windows, sistemas con hilos POSIX.

Este módulo también define la siguiente constante:

`threading.TIMEOUT_MAX`

El máximo valor permitido para el parámetro *timeout* de las funciones bloqueantes (`Lock.acquire()`, `RLock.acquire()`, `Condition.wait()`, etc.). La especificación de un tiempo de espera mayor a este valor lanzará un `OverflowError`.

Nuevo en la versión 3.2.

Este módulo define un número de clases, las cuales son detalladas en las siguientes secciones.

El diseño de este módulo está libremente basado en el modelo de *threading* de Java. Sin embargo, donde Java hace de *locks* y variables condicionales el comportamiento básico de cada objeto, éstos son objetos separados en Python. La clase de Python `Thread` soporta un subdominio del comportamiento de la clase `Thread` de Java; actualmente, no hay prioridades, ni grupos de hilos, y los hilos no pueden ser destruidos, detenidos, suspendidos, retomados o interrumpidos. Los métodos estáticos de la clase `Thread` de Java, cuando son implementados, son mapeados a funciones a nivel de módulo.

Todos los métodos descritos abajo son ejecutados de manera atómica.

17.1.1 Datos locales del hilo

Los datos locales de hilo son datos cuyos valores son específicos a cada hilo. Para manejar los datos locales de hilos, simplemente crear una instancia de `local` (o una subclase) y almacenar los atributos en ella:

```
mydata = threading.local()
mydata.x = 1
```

Los valores de instancia serán diferentes para hilos distintos.

class `threading.local`

Una clase que representa datos locales de hilo.

Para más detalles y ejemplos extensivos, véase la documentación del módulo `_threading_local`.

17.1.2 Objetos tipo hilo

La clase `Thread` representa una actividad que corre en un hilo de control separado. Hay dos manera de especificar la actividad: pasando un objeto invocable al constructor, o sobrescribiendo el método `run()` en una subclase. Ningún otro método (a excepción del constructor) deberá ser sobrescrito en una subclase. En otras palabras, *solo* sobrescribir los métodos `__init__()` y `run()` de esta clase.

Una vez que un objeto *thread* es creado, su actividad debe ser iniciada llamando al método `start()` del hilo. Ésto invoca el método `run()` en un hilo de control separado.

Una vez que la actividad del hilo ha sido iniciada, el hilo se considerará “vivo”. Deja de estar vivo cuando su método `run()` termina – ya sea normalmente, o por lanzar una excepción no manejada. El método `is_alive()` verifica si acaso el hilo está vivo.

Otros hilos pueden llamar al método `join()` de un hilo. Esto bloquea el hilo llamador hasta que el hilo cuyo método `join()` ha sido llamado termine.

Un hilo tiene un nombre. El nombre puede ser pasado al constructor y leído o cambiado a través del atributo `name`.

Si el método `run()` lanza una excepción, se llama a `threading.excepthook()` para gestionarla. Por defecto, `threading.excepthook()` ignora silenciosamente a `SystemExit`.

Un hilo puede ser marcado como un «hilo demonio». El significado de esta marca es que la totalidad del programa de Python finalizará cuando solo queden hilos demonio. El valor inicial es heredado del hilo creador. La marca puede ser establecida a través de la propiedad `daemon` o del argumento `daemon` en el constructor.

Nota: Los hilos demonio son detenidos abruptamente al momento del cierre. Sus recursos (tales como archivos abiertos, transacciones con bases de datos, etc.) pueden no ser liberados adecuadamente. Si se requiere que los hilos se detengan con gracia, háganse no-demoníacos y úsese un mecanismo de señalización adecuado tal como un `Event`.

Existe un objeto «hilo principal»; éste corresponde al hilo de control inicial del programa de Python. No es un hilo demonio.

Existe la posibilidad de crear «objetos de hilos *dummy*». Estos son objetos hilo correspondientes a «hilos extranjeros», que son hilos de control iniciados afuera del modulo `threading`, por ejemplo directamente de código en C. Los objetos de hilos *dummy* tienen funcionalidad limitada; siempre se consideran vivos y demoníacos, y no pueden ser les puede aplicar el método `join()`. Nunca son eliminados, ya que es imposible detectar la terminación de hilos extranjeros.

class `threading.Thread` (*group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None*)

Este constructor siempre debe ser llamado con argumentos de palabra clave. Los argumentos son:

group debe ser `None`; reservado para una futura extensión cuando se implemente una clase `ThreadGroup`.

target es el objeto invocable a ser invocado por el método `run()`. Por defecto es `None`, lo que significa que nada es llamado.

name es el nombre del hilo. Por defecto, se construye un nombre único con la forma «*Thread-N*» donde *N* es un número decimal pequeño.

args es la tupla de argumento para la invocación objetivo. Por defecto es `()`.

kwargs es un diccionario de argumentos de palabra clave para la invocación objetivo. Por defecto es `{}`.

Si no es `None`, *daemon* establece explícitamente si el hilo es demoníaco. Si es `None` (el valor por defecto), la propiedad demoníaca es heredada del hilo actual.

Si la subclase sobrescribe el constructor, debe asegurarse de invocar al constructor de la clase base (`Thread.__init__()`) antes de hacer cualquier otra cosa al hilo.

Distinto en la versión 3.3: Se agregó el argumento *daemon*.

start()

Inicia la actividad del hilo.

Debe ser llamada máximo una vez por objeto hilo. Se encarga de que el método `run()` del objeto sea invocado en un hilo de control separado.

Este método lanzará un `RuntimeError` si se llama más de una vez en el mismo objeto hilo.

run()

Método que representa la actividad del hilo.

Se puede sobrescribir este método en una subclase. El método estándar `run()` invoca el objeto invocable pasado al constructor del objeto como argumento *target*, si lo hay, con argumentos posicionales y de palabra clave tomados de los argumentos *args* y *kwargs*, respectivamente.

join (*timeout=None*)

Espera a que el hilo termine. Esto bloquea el hilo llamador hasta que el hilo cuyo método `join()` es llamado finalice – ya sea normalmente o a través de una excepción no gestionada – o hasta que el tiempo de espera opcional caduque.

Cuando se presenta un argumento *timeout* y no es `None`, debe ser un número de punto flotante que especifique un tiempo de espera en segundos (o en fracciones de segundo) para la operación. Ya que `join()` siempre retorna `None`, se debe llamar a `is_alive()` después de `join()` para decidir si acaso caducó el tiempo de espera – si el hilo todavía está vivo, la llamada a `join()` caducó.

Cuando el argumento *timeout* no se presenta o es `None`, la operación bloqueará hasta que el hilo termine.

A un hilo se le puede aplicar `join()` muchas veces.

`join()` lanza un `RuntimeError` si se intenta unir el hilo actual ya que ello generaría un punto muerto. También es un error aplicar `join()` a un hilo antes de que haya sido iniciado y los intentos de hacerlo lanzarán la misma excepción.

name

Un *string* utilizado con propósitos de identificación. No posee semántica. Se puede dar el mismo nombre a múltiples hilos. El nombre inicial es establecido por el constructor.

getName ()

setName ()

Antigua API *getter/setter* para *name*; úsese en cambio directamente como una propiedad.

ident

El “identificador de hilo” de este hilo o `None` si el hilo no ha sido iniciado. Es un entero distinto de cero. Ver la función `get_ident()`. Los identificadores de hilos pueden ser reciclados cuando un hilo finaliza y otro hilo es creado. El identificador está disponible incluso después de que el hilo ha abandonado.

native_id

La ID integral nativa de este hilo. Es un entero no negativo, o `None` si el hilo no ha sido iniciado. Ver la función `get_native_id()`. Ésta representa la *Thread ID* (TID) tal como haya sido asignada al hilo por el SO (*kernel*). Su valor puede ser utilizado para identificar específicamente a este hilo en particular a través de todo el sistema (hasta que el hilo termina, luego de lo cual el valor puede ser reciclado por el SO).

Nota: Similar a las *Process IDs*, las *Thread IDs* sólo son válidas (garantizadas como únicas a través de todo el sistema) desde el momento en que se crea el hilo hasta que el hilo es finalizado.

Disponibilidad: Requiere la función `get_native_id()`.

Nuevo en la versión 3.8.

is_alive ()

Retornar si acaso el hilo está vivo.

Este método retorna `True` desde justo antes de que el método `run()` inicie hasta justo antes de que el método `run()` termine. La función `enumerate()` del módulo retorna una lista de todos los hilos vivos.

daemon

Un valor booleano que indica si este hilo es un hilo demonio (*True*) o no (*False*). Debe ser establecido antes de que se llame a `start()`, de lo contrario se lanzará un `RuntimeError`. Su valor inicial se hereda del hilo creador; el hilo principal no es un hilo demonio y por lo tanto todos los hilos creados en el hilo principal tienen por defecto un valor `daemon=False`.

El programa de Python en su totalidad finaliza cuando no queda ningún hilo no-demonio vivo.

isDaemon ()

setDaemon ()

Antigua API *getter/setter* para *daemon*; úsese en cambio directamente como una propiedad.

17.1.3 Objetos tipo *lock*

Una primitiva *lock*, es una primitiva de sincronización que no pertenece a ningún hilo en particular cuando está cerrado. En Python, es la primitiva de sincronización de más bajo nivel actualmente disponible, implementado directamente por el módulo de extensión `_thread`.

Una primitiva *lock* está en uno de dos estados, «cerrado» o «abierto» (*locked/unlocked*). Se crea en estado abierto. Tiene dos métodos básicos, `acquire()` (adquirir) y `release()` (liberar). Cuando el estado es *abierto*, `acquire()` cambia el estado a cerrado y retorna inmediatamente. Cuando el estado es *cerrado*, `acquire()` bloquea hasta que una llamada a `release()` en otro hilo lo cambie a abierto, luego la llamada a `acquire()` lo restablece a cerrado y retorna. El método `release()` sólo debe ser llamado en el estado cerrado; cambia el estado a abierto y retorna inmediatamente. Si se realiza un intento de liberar un *lock* abierto, se lanzará un `RuntimeError`.

Los *locks* también soportan el *protocolo de gestión de contexto*.

Cuando más de un hilo está bloqueado en `acquire()` esperando que el estado sea abierto, sólo un hilo procederá cuando una llamada a `release()` restablezca el estado a abierto; cuál de los hilos en espera procederá no está definido, y puede variar a través de las implementaciones.

Todos los métodos se ejecutan de manera atómica.

class `threading.Lock`

La clase que implemente los objetos de la primitiva *lock*. Una vez que un hilo ha adquirido un *lock*, intentos subsiguientes por adquirirlo bloquearán, hasta que sea liberado; cualquier hilo puede liberarlo.

Nótese que `Lock` es una función de fábrica que retorna una instancia de la versión más eficiente de la clase `Lock` concreta soportada por la plataforma.

acquire (*blocking=True, timeout=-1*)

Adquirir un *lock*, bloqueante o no bloqueante.

Cuando se invoca con el argumento *blocking* establecido como `True` (el valor por defecto), bloquea hasta que el *lock* se abra, luego lo establece como cerrado y retorna `True`.

Cuando es invocado con el argumento *blocking* como `False`, no bloquea. Si una llamada con *blocking* establecido como `True` bloqueara, retorna `Falso` inmediatamente; de otro modo, cierra el *lock* y retorna `True`.

Cuando se invoca con el argumento de punto flotante *timeout* fijado a un valor positivo, bloquea por a lo más el número de segundos especificado en *timeout* y mientras el *lock* no pueda ser adquirido. Un argumento *timeout* de «-1» especifica una espera ilimitada. No está admitido especificar un *timeout* cuando *blocking* es falso.

El valor de retorno es `True` si el *lock* es adquirido con éxito, `Falso` si no (por ejemplo si *timeout* expiró).

Distinto en la versión 3.2: El parámetro *timeout* es nuevo.

Distinto en la versión 3.2: La adquisición de un *lock* ahora puede ser interrumpida por señales en POSIX si la implementación de hilado subyacente lo soporta.

release ()

Libera un *lock*. Puede ser llamado desde cualquier hilo, no solo el hilo que ha adquirido el *lock*.

Cuando el *lock* está cerrado, lo restablece a abierto, y retorna. Si cualquier otro hilo está bloqueado esperando que el *lock* se abra, permite que exactamente uno de ellos proceda.

Cuando se invoca en un *lock* abierto, se lanza un `RuntimeError`.

No hay valor de retorno.

locked ()

Retorna *true* si el *lock* ha sido adquirido.

17.1.4 Objetos *Rlock*

Un *lock* reentrante es una primitiva de sincronización que puede ser adquirido múltiples veces por el mismo hilo. Internamente, utiliza el concepto de «hilo dueño» y «nivel de recursividad» además del estado abierto/cerrado utilizado por las primitivas *locks*. Si está en estado cerrado, algún hilo es dueño del *lock*; si está en estado abierto, ningún hilo es dueño.

Para cerrar el *lock*, un hilo llama a su método `acquire()`; esto retorna una vez que el hilo se ha adueñado del *lock*. Para abrir el *lock*, un hilo llama a su método `release()`. Pares de llamadas `acquire()/release()` pueden anidarse; sólo el `release()` final (el `release()` del par más externo) restablece el *lock* a abierto y permite que otro hilo bloqueado en `acquire()` proceda.

Los *locks* reentrantes también soportan el *protocolo de manejo de contextos*.

class `threading.RLock`

Esta clase implementa objetos tipo *lock* reentrantes. Un *lock* reentrante debe ser liberado por el hilo que lo adquirió. Una vez que un hilo ha adquirido un *lock* reentrante, el mismo hilo puede adquirirlo otra vez sin bloquearse; el hilo debe liberarlo una vez por vez que lo adquiere.

Nótese que `RLock` en realidad es una función fábrica que retorna una instancia de la versión más eficiente de la clase `RLock` concreta que sea soportada por la plataforma.

acquire (*blocking=True, timeout=-1*)

Adquirir un *lock*, bloqueante o no bloqueante.

Cuando se invoca sin argumentos: si este hilo ya es dueño del *lock*, incrementa el nivel de recursividad en uno, y retorna inmediatamente. De otro modo, si otro hilo es dueño del *lock*, bloquea hasta que se abra el *lock*. Una vez que el *lock* se abra (ningún hilo sea su dueño), se adueña, establece el nivel de recursividad en uno, y retorna. Si más de un hilo está bloqueado esperando que sea abra el *lock*, solo uno a la vez podrá apoderarse del *lock*. No hay valor de retorno en este caso.

Cuando se invoca con el argumento *blocking* fijado en *true*, hace lo mismo que cuando se llama sin argumentos y retorna `True`.

Cuando se invoca con el argumento *blocking* fijado a falso, no bloquea. Si una llamada sin argumento bloquease, retorna `False` inmediatamente; de otro modo, hace lo mismo que al llamarse sin argumentos, y retorna `True`.

Cuando se invoca con el argumento de coma flotante *timeout* fijado a un valor positivo, bloquea por máximo el número de segundos especificado por *timeout* y mientras el *lock* no pueda ser adquirido. Retorna `True` si el *lock* ha sido adquirido, falso si el tiempo de espera *timeout* ha caducado.

Distinto en la versión 3.2: El parámetro *timeout* es nuevo.

release ()

Libera un *lock*, disminuyendo el nivel de recursividad. Si después de la disminución es cero, restablece el *lock* a abierto (no perteneciente a ningún hilo), y si cualquier otro hilo está bloqueado esperando que se abra el *lock*, permite que exactamente uno de ellos proceda. Si luego de la disminución el nivel de recursividad todavía no es cero, el *lock* permanece cerrado y perteneciente al hilo llamador.

Solo llámese este método cuando el hilo llamador sea dueño del *lock*. Se lanza un `RuntimeError` si se llama este método cuando el *lock* esta abierto.

No hay valor de retorno.

17.1.5 Objetos condicionales

Una condición variable siempre va asociada a algún tipo de *lock*. éste puede ser provisto o se creará uno por defecto. Proveer uno es útil cuando varias variables de condición deben compartir el mismo *lock*. El *lock* es parte del objeto condicional: no es necesario rastrearlo por separado.

Una condición variable obedece el *protocolo de gestión de contexto*: al usar la declaración `with` se adquiere el *lock* asociado por la duración del bloque contenido. Los métodos `acquire()` y `release()` también llaman los métodos correspondientes del *lock* asociado.

Otros métodos deben llamarse con el *lock* asociado conservado. El método `wait()` libera el *lock*, y luego bloquea hasta que otro hilo lo despierte llamando `notify()` o `notify_all()`. Una vez que ha sido despertado, `wait()` re-adquiere el *lock* y retorna. También es posible especificar un tiempo de espera.

El método `notify()` despierta a uno de los hilos que esperan a la condición variable, si es que alguno espera. El método `notify_all()` despierta a todos los hilos que estén esperando a la condición variable.

Nota: Los métodos `notify()` y `notify_all()` no liberan el *lock*; esto significa que el hilo o los hilos que han sido despertados no retornaran de su llamada de `wait()` inmediatamente, sino solo una vez que el hilo que haya llamado a `notify()` o `notify_all()` renuncie finalmente a la propiedad del *lock*.

El estilo típico de programación con variables condicionales utiliza el *lock* para sincronizar el acceso a algún estado compartido; hilos que estén interesados en un cambio de estado en particular llamarán a `wait()` reiteradamente hasta que vean el estado deseado, mientras que los hilos que modifiquen el estado llamarán a `notify()` o a `notify_all()` cuando cambien el estado de modo que pudiera ser que el estado sea el deseado por alguno de los hilos en espera. Por ejemplo, el siguiente código es una situación genérica de productor-consumidor con capacidad de búfer ilimitada:

```
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

El bucle `while` que verifica la condición de la aplicación es necesario porque `wait()` puede retornar después de una cantidad arbitraria de tiempo, y la condición que dio pie a la llamada de `notify()` puede ya no ser verdadera. Esto es inherente a la programación multi-hilo. El método `wait_for()` puede usarse para automatizar la revisión de condiciones, y facilita la computación de tiempos de espera:

```
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

Para elegir entre `notify()` y `notify_all()`, considérese si un cambio de estado puede ser interesante para uno o varios hilos en espera. Por ejemplo en una típica situación productor-consumidor, agregar un elemento al búfer sólo necesita despertar un hilo consumidor.

class `threading.Condition (lock=None)`

Esta clase implementa objetos de condición variable. Una condición variable permite que uno o más hilos esperen hasta que sean notificados por otro hilo.

Si se provee un argumento *lock* distinto de `None`, debe ser un objeto *Lock* o *RLock*, y se utiliza como el *lock* subyacente. De otro modo, se crea un nuevo objeto *RLock* y se utiliza como el *lock* subyacente.

Distinto en la versión 3.3: cambiado de función de fábrica a una clase.

acquire (*args)

Adquiere el *lock* subyacente. Este método llama al método correspondiente sobre el *lock* subyacente; el

valor de retorno es lo que retorne aquel método.

release()

Libera el *lock* subyacente. Este método llama al método correspondiente en el *lock* subyacente; no tiene valor de retorno.

wait(timeout=None)

Espera hasta ser notificado o hasta que el tiempo de espera caduque. Si el hilo invocador no ha adquirido el *lock* cuando este método es llamado, se lanza un *RuntimeError*.

Este método libera el *lock* subyacente, y luego bloquea hasta ser despertado por una llamada a *notify()* o *notify_all()* para la misma condición variable en otro hilo, o hasta que el tiempo de espera opcional se cumpla. Una vez que ha sido despertado o el tiempo de espera ha pasado, re-adquiere el *lock* y retorna.

Cuando haya un argumento *timeout* presente y no sea *None*, debe ser un número de punto flotante que especifique un tiempo de espera para la operación en segundos (o fracciones de segundo).

Cuando el *lock* subyacente es un *RLock*, no se libera utilizando su método *release()*, ya que esto podría no abrir realmente el *lock* cuando haya sido adquirido múltiples veces recursivamente. En cambio, se usa una interfaz interna de la clase *RLock*, que lo abre realmente incluso cuando haya sido adquirido múltiples veces recursivamente. Otra interfaz interna se usa luego para restablecer el nivel de recursividad cuando el *lock* es readquirido.

El valor de retorno es *True* a menos que un *timeout* dado haya expirado, en cuyo caso será *False*.

Distinto en la versión 3.2: Previamente, el método siempre retornaba *None*.

wait_for(predicate, timeout=None)

Espera a que una condición se evalúe como verdadera. *predicate* debe ser un invocable cuyo resultado se interpretará como un valor booleano. Se puede proveer un *timeout* que especifique el máximo tiempo de espera.

Este método utilitario puede llamar a *wait()* reiteradas veces hasta que se satisfaga el predicado, o hasta que la espera caduque. El valor de retorno es el último valor de retorno del predicado y se evaluará a *False* si el método ha caducado.

Al ignorar la propiedad *feature*, llamar a este método equivale vagamente a escribir:

```
while not predicate():
    cv.wait()
```

Por ende, aplican las mismas reglas que con *wait()*: El *lock* debe ser conservado cuando se llame y es re-adquirido al momento del retorno. El predicado se evalúa con el *lock* conservado.

Nuevo en la versión 3.2.

notify(n=1)

Por defecto, despierta a un hilo que esté esperando por esta condición, si lo existe. Si el hilo llamador no ha adquirido el *lock* cuando se llama este método, se lanza un *RuntimeError*.

Este método despierta como máximo *n* de los hilos que estén esperando por la condición variable; no es una opción si no hay hilos esperando.

La implementación actual despierta exactamente *n* hilos, si hay por lo menos *n* hilos esperando. Sin embargo, no es seguro apoyarse en este comportamiento. A futuro, una implementación optimizada podría ocasionalmente despertar a más de *n* hilos.

Nota: un hilo que ha sido despertado no retorna realmente de su llamada a *wait()* hasta que pueda readquirir el *lock*. Ya que *notify()* no libera el *lock*, su llamador debiera hacerlo.

notify_all()

Despierta a todos los hilos que esperen por esta condición. Este método actúa como *notify()*, pero despierta a todos los hilos en espera en vez de a uno. Si el hilo llamador no ha adquirido el *lock* cuando se llama a este método, se lanza un *RuntimeError*.

17.1.6 Objetos semáforo

Éste es uno de las primitivas de sincronización más antiguos en la historia de las ciencias de la computación, inventado por el pionero en ciencias de la computación holandés Edsger W. Dijkstra (él utilizó los nombres `P()` y `V()` en lugar de `acquire()` y `release()`)

Un semáforo administra un contador interno que se disminuye por cada llamada a `acquire()` y se incrementa por cada llamada a `release()`. El contador no puede bajar de cero; cuando `acquire()` lo encuentra en cero, bloquea, esperando hasta que otro hilo llame `release()`.

Los semáforos también tienen soporte para el *protocolo de gestión de contexto*.

class `threading.Semaphore` (*value=1*)

Esta clase implementa los objetos semáforo. Un semáforo gestiona un contador atómico que representa el número de llamadas a `release()` menos el número de llamadas a `acquire()`, más un valor inicial. El método `acquire()` bloquea si es necesario, hasta que pueda retornar sin volver el contador negativo. Si no es provisto, el valor por defecto de *value* será 1.

El argumento opcional da el *value* inicial al contador interno; por defecto es 1. Si el *value* provisto es menor a 0; se lanza un `ValueError`.

Distinto en la versión 3.3: cambiado de función de fábrica a una clase.

acquire (*blocking=True*, *timeout=None*)

Adquirir un semáforo.

Cuando se invoca sin argumentos:

- Si el contador interno es mayor a cero de entrada, lo disminuye en uno y retorna `True` inmediatamente.
- Si el contador interno es cero de entrada, bloquea hasta ser despertado por una llamada a `release()`. Una vez despierto (y el contador sea mayor a 0), disminuye el contador en 1 y retorna `True`. Se despertará exactamente un hilo por cada llamada a `release()`. No debiese confiarse en el orden en que los hilos sean despertados.

Cuando se invoca con *blocking* fijado en falso, no bloquea. Si una llamada sin un argumento bloquease, retorna `Falso` inmediatamente; de otro modo, hace lo mismo que cuando se llama sin argumentos, y retorna `True`.

Cuando se invoca con *timeout* distinto de `None`, bloqueará por un tiempo máximo en segundos fijados en *timeout*. Si `acquire` no se completa exitosamente en ese intervalo, retorna `False`. De otro modo retorna `True`.

Distinto en la versión 3.2: El parámetro *timeout* es nuevo.

release ()

Libera un semáforo, incrementando el contador interno en uno. Cuando éste haya sido cero de entrada y otro hilo esté esperando que sea nuevamente mayor a cero, despierta a ese hilo.

class `threading.BoundedSemaphore` (*value=1*)

Clase que implementa objetos de semáforo delimitados. Un semáforo delimitado verifica que su valor actual no exceda su valor inicial. Si lo hace, se lanza un `ValueError`. En la mayoría de las situaciones se utilizan los semáforos para cuidar recursos con capacidad limitada. Si se libera el semáforo demasiadas veces es signo de un *bug*. Si no se provee, el valor por defecto de *value* será 1.

Distinto en la versión 3.3: cambiado de función de fábrica a una clase.

Ejemplo de Semaphore

Los semáforos suelen utilizarse para cuidar recursos con capacidad limitada, por ejemplo, un servidor de base de datos. En cualquier situación en que el tamaño de los recursos sea fijo, se debe usar un semáforo delimitado. Antes de generar cualquier hilo de trabajo, tu hilo principal debe inicializar el semáforo:

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Una vez que han sido generados, los hilos de trabajo llaman a los métodos *acquire* y *release* cuando necesitan conectarse al servidor:

```
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

El uso de semáforos delimitados reduce la posibilidad de que pase inadvertido un error de programación que cause que el semáforo sea liberado más veces de las que sea adquirido.

17.1.7 Objetos de eventos

Éste es uno de los mecanismos más simples de comunicación entre hilos: un hilo señala un evento y otro hilo lo espera.

Un objeto de evento maneja una marca interna que puede ser establecida como verdadera mediante el método *set()* y restablecida a falsa mediante el método *clear()*. El método *wait()* bloquea hasta que la marca sea *true*.

class threading.Event

Clase que implementa los objetos de evento. Un evento gestiona un indicador que puede ser establecido a verdadero mediante el método *set()* y restablecido a falso con el método *clear()*. El método *wait()* bloquea hasta que el indicador sea verdadero. El indicador es inicialmente falso.

Distinto en la versión 3.3: cambiado de función de fábrica a una clase.

is_set()

Retorna *True* exclusivamente si el indicador interno es verdadero.

set()

Establece el indicador interno a verdadero. Todos los hilos que estén esperando que se vuelva verdadero serán despertados. Los hilos que llaman a *wait()* una vez que el indicador marca verdadero no bloquearán.

clear()

Restablece el indicador a falso. Posteriormente, los hilos que llamen a *wait()* bloquearán hasta que se llame a *set()* para establecer el indicador interno a verdadero nuevamente.

wait(timeout=None)

Bloquea hasta que el indicador interno sea verdadero. Si el indicador interno es verdadero de entrada, retorna inmediatamente. De otro modo, bloquea hasta que otro hilo llame a *set()* para establecer el indicador a verdadero, o hasta que el tiempo de espera opcional caduque.

Cuando se presenta un argumento para el tiempo de espera *timeout* distinto de *None*, debe ser un número de punto flotante que especifique un tiempo de espera para la operación en segundos (o fracciones en su defecto).

Este método retorna *True* exclusivamente si el indicador interno ha sido establecido a verdadero, ya sea antes de la llamada a la espera o después de que la espera inicie, por lo que siempre retorna *True* excepto si se provee un tiempo de espera máximo y la operación caduca.

Distinto en la versión 3.1: Previamente, el método siempre retornaba `None`.

17.1.8 Objetos temporizadores

Esta clase representa una acción que sólo debe ejecutarse luego de que una cierta cantidad de tiempo transcurra — un temporizador. `Timer` es una subclase de `Thread` y en tanto tal también funciona como un ejemplo de creación de hilos personalizados.

Los temporizadores son iniciados, tal como los hilos, al llamarse su método `start()`. El temporizador puede ser detenido (antes de que su acción haya comenzado) al llamar al método `cancel()`. El intervalo que el temporizador esperará antes de ejecutar su acción puede no ser exactamente el mismo que el intervalo especificado por el usuario.

Por ejemplo:

```
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=None, kwargs=None*)

Crear un temporizador que ejecutará *function* con los argumentos *args* y los argumentos de palabra clave *kwargs*, luego de que una cantidad *interval* de segundos hayan transcurrido. Si *args* es `None` (por defecto) se utilizará una lista vacía. Si *kwargs* es `None` (por defecto) se utilizará un *dict* vacío.

Distinto en la versión 3.3: cambiado de función de fábrica a una clase.

cancel()

Detiene el temporizador, y cancela la ejecución de la acción del temporizador. Esto sólo funcionará si el temporizador está en etapa de espera.

17.1.9 Objetos de barrera

Nuevo en la versión 3.2.

Esta clase provee una primitiva de sincronización simple para ser usado por un número fijo de hilos que necesitan esperarse entre ellos. Cada uno de los hilos intenta pasar la barrera llamando al método `wait()` y bloqueará hasta que todos los hilos hayan hecho sus respectivas llamadas a `wait()`. En este punto, los hilos son liberados simultáneamente.

La barrera puede ser reutilizada cualquier número de veces para el mismo número de hilos.

Como ejemplo, aquí hay una manera simple de sincronizar un hilo cliente con uno servidor:

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class `threading.Barrier` (*parties, action=None, timeout=None*)

Crear un objeto de barrera para un número *parties* de hilos. Una *action*, si es provista, es un invocable a ser

llamado por uno de los hilos cuando sean liberados. *timeout* es el valor de tiempo de espera máximo por defecto si no se especifica uno en el método `wait()`.

wait (*timeout=None*)

Pasa la barrera. Cuando todos los hilos involucrados en el objeto barrera han llamado esta función, se liberan todos simultáneamente. Si se provee un valor *timeout*, se utilizará con preferencia sobre cualquiera que haya sido suministrado al constructor de la clase.

El valor de retorno es un entero en el rango desde 0 hasta *parties* - 1, diferente para cada hilo. Puede ser utilizado para seleccionar a un hilo para que haga alguna limpieza especial, por ejemplo:

```
i = barrier.wait()
if i == 0:
    # Only one thread needs to print this
    print("passed the barrier")
```

Se se provee una *action* al constructor, uno de los hilos la habrá llamado antes de ser liberado. Si acaso esta llamada lanzara un error, la barrera entra en estado *broken* (roto).

Si la llamada caduca, la barrera entra en estado *broken*.

Este método podría lanzar una excepción *BrokenBarrierError* si la barrera está rota o si se reinicia mientras el hilo está esperando.

reset ()

Retorna la barrera al estado por defecto, vacío. Cualquier hilo que esté a su espera recibirá la excepción *BrokenBarrierError*.

Nótese que utilizar esta función podría requerir alguna sincronización externa si existen otros hilos cuyos estados sean desconocidos. Si una barrera se rompe puede ser mejor abandonarla y crear una nueva.

abort ()

Coloca la barrera en estado roto. Esto causa que cualquier llamada activa o futura a `wait()` falle con el error *BrokenBarrierError*. Úsese por ejemplo si uno de los hilos necesita abortar, para evitar que la aplicación quede en punto muerto.

Puede ser preferible simplemente crear la barrera con un valor *timeout* sensato para cuidarse automáticamente de que uno de los hilos falle.

parties

El número de hilos requeridos para pasar la barrera.

n_waiting

El número de hilos actualmente esperando en la barrera.

broken

Un valor booleano que será `True` si la barrera está en el estado roto.

exception `threading.BrokenBarrierError`

Esta excepción, una subclase de *RuntimeError*, se lanza cuando el objeto *Barrier* se restablece o se rompe.

17.1.10 Uso de *locks*, condiciones y semáforos en la declaración *with*

Todos los objetos provistos por este módulo que tienen métodos `acquire()` y `release()` pueden ser utilizados como administradores de contexto para una declaración *with*. El método `acquire()` será llamado cuando se ingresa al bloque y el método `release()` será llamado cuando se abandona el bloque. De ahí que, el siguiente fragmento:

```
with some_lock:
    # do something...
```

sea equivalente a:

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

Actualmente, los objetos *Lock*, *RLock*, *Condition*, *Semaphore*, y *BoundedSemaphore* pueden ser utilizados como gestores de contexto con declaraciones `with`.

17.2 multiprocessing — Paralelismo basado en procesos

Código fuente: [Lib/multiprocessing/](#)

17.2.1 Introducción

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.

El módulo *multiprocessing* también introduce API que no tienen análogos en el módulo *threading*. Un buen ejemplo de esto es el objeto *Pool* que ofrece un medio conveniente de paralelizar la ejecución de una función a través de múltiples valores de entrada, distribuyendo los datos de entrada a través de procesos (paralelismo de datos). El siguiente ejemplo demuestra la práctica común de definir tales funciones en un módulo para que los procesos secundarios puedan importar con éxito ese módulo. Este ejemplo básico de paralelismo de datos usando *Pool*,

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

imprimirá la salida estándar

```
[1, 4, 9]
```

La clase *Process*

En *multiprocessing*, los procesos se generan creando un objeto *Process* y luego llamando a su método `start()`. *Process* sigue la API de *threading.Thread*. Un ejemplo trivial de un programa multiproceso es

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Para mostrar las IDs individuales involucradas en el proceso, aquí hay un ejemplo ampliado:

```

from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    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()

```

Para obtener una explicación de por qué es necesaria la parte `if __name__ == '__main__':`, consulte [Pautas de programación](#).

Contextos y métodos de inicio

Dependiendo de la plataforma, `multiprocessing` admite tres formas de iniciar un proceso. Estos métodos de inicio *start methods* son

Generación (*spawn*) The parent process starts a fresh python interpreter process. The child process will only inherit those resources necessary to run the process object's `run()` method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using *fork* or *forkserver*.

Disponible en Unix y Windows. Por defecto en Windows y macOS.

fork El proceso parental usa `os.fork()` para bifurcar el intérprete de Python. El proceso hijo, cuando comienza, es efectivamente idéntico al proceso parental. Todos los recursos del proceso parental son heredados por el proceso hijo. Tenga en cuenta que bifurcar (*forking*) de forma segura un proceso multihilo es problemático.

Disponible solo en Unix. Por defecto en Unix.

forkserver Cuando el programa se inicia y selecciona el método de inicio *forkserver*, se inicia un proceso de servidor. A partir de ese momento, cada vez que se necesite un nuevo proceso, el proceso parental se conecta al servidor y solicita que bifurque un nuevo proceso. El proceso del servidor *fork* es de un solo hilo, por lo que es seguro usarlo `os.fork()`. No se heredan recursos innecesarios.

Disponible en plataformas Unix que admiten pasar descriptores de archivo a través de tuberías (*pipes*) Unix.

Distinto en la versión 3.8: En macOS, el método de inicio *spawn* ahora es el predeterminado. El método de inicio *fork* debe considerarse inseguro ya que puede provocar bloqueos del subprocesso. Consulte [bpo-33725](#).

Distinto en la versión 3.4: *spawn* fue añadido en todas las plataformas Unix, y *forkserver* fue agregado para algunas plataformas Unix. Los procesos hijos (*child processes*) ya no heredan todos los identificadores heredables de los procesos parentales en Windows.

En Unix, los métodos de inicio *spawn* o *forkserver* también iniciarán un proceso *resource tracker* que rastrea los recursos del sistema con nombre no vinculados (como los nombrados semáforos o objetos `SharedMemory`) creado por procesos del programa. Cuando todos los procesos han salido, el rastreador de recursos desvincula cualquier objeto rastreado restante. Por lo general, no debería haber ninguno, pero si un proceso fue eliminado por una señal, puede haber algunos recursos «filtrados». (Ni los semáforos filtrados ni los segmentos de memoria compartida se desvincularán automáticamente hasta el próximo reinicio. Esto es problemático para ambos objetos porque el sistema

solo permite un número limitado de semáforos con nombre, y los segmentos de memoria compartida ocupan algo de espacio en la memoria principal).

Para seleccionar un método de inicio, utilice la función `set_start_method()` en la cláusula `if __name__ == '__main__'` del módulo principal. Por ejemplo:

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

`set_start_method()` no debería ser usada más de una vez en el programa.

Alternativamente, se puede usar `get_context()` para obtener un objeto de contexto. Los objetos de contexto tienen la misma API que el módulo de multiprocesamiento, y permiten utilizar múltiples métodos de inicio en el mismo programa.

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

Tenga en cuenta que los objetos relacionados con un contexto pueden no ser compatibles con procesos para un contexto diferente. En particular, los *locks* creados con el contexto *fork* no se pueden pasar a los procesos iniciados con los métodos de inicio *spawn* o *forkserver*.

Una biblioteca que quiera usar un método de inicio particular probablemente debería usar `get_context()` para evitar interferir con la elección del usuario de la biblioteca.

Advertencia: Los métodos de inicio `'spawn'` y `'forkserver'` actualmente no pueden ser usados con ejecutables «congelados» (*frozen*) (es decir, binarios producidos por paquetes como **PyInstaller** y **cx_Freeze**) en Unix. El método de inicio `'fork'` funciona.

Intercambiando objetos entre procesos

`multiprocessing` admite dos tipos de canales de comunicación entre procesos:

Colas (*queues*)

La clase `Queue` es prácticamente un clon de `queue.Queue`. Por ejemplo:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])
```

(continué en la próxima página)

(proviene de la página anterior)

```

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())      # prints "[42, None, 'hello']"
    p.join()

```

Las colas (*queues*) son hilos y procesos seguro.

Tuberías (*Pipes*)

La función `Pipe()` retorna un par de objetos de conexión conectados por una tubería (*pipe*) que, por defecto, es un dúplex (bidireccional). Por ejemplo:

```

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()

```

Los dos objetos de conexión retornados por `Pipe()` representan los dos extremos de la tubería (*pipe*). Cada objeto de conexión tiene los métodos `send()` y `recv()` (entre otros). Tenga en cuenta que los datos en una tubería pueden corromperse si dos procesos (o hilos) intentan leer o escribir en el *mismo* (*same*) extremo de la tubería al mismo tiempo. Por supuesto, no hay riesgo de corrupción por procesos que utilizan diferentes extremos de la tubería (*pipe*) al mismo tiempo.

Sincronización entre procesos

`multiprocessing` contiene equivalentes de todas las sincronizaciones primitivas de `threading`. Por ejemplo, se puede usar un candado (*lock*) para garantizar que solo un proceso se imprima a la salida estándar a la vez:

```

from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()

```

Sin usar el candado (*lock*) de salida de los diferentes procesos, es probable que todo se mezcle.

Compartiendo estado entre procesos

Como se mencionó anteriormente, cuando se realiza una programación concurrente, generalmente es mejor evitar el uso del estado compartido en la medida de lo posible. Esto es particularmente cierto cuando se utilizan múltiples procesos.

Sin embargo, si usted realmente necesita usar algunos datos compartidos el *multiprocessing* proporciona un par de maneras de hacerlo.

Memoria compartida

Los datos se pueden almacenar en un mapa de memoria compartida usando *Value* o *Array*. Por ejemplo, el siguiente código

```
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[:])
```

imprimirá

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Los argumentos 'd' y 'i' utilizados al crear `num` y `arr` son códigos de tipo del tipo utilizado por *array* module: 'd' indica un flotador de doble precisión y 'i' indica un entero con signo. Estos objetos compartidos serán seguros para procesos y subprocessos.

Para una mayor flexibilidad en el uso de la memoria compartida, se puede usar el módulo *multiprocessing.sharedctypes* que admite la creación arbitraria de objetos *ctypes* asignados desde la memoria compartida.

Proceso servidor (*Server process*)

Un objeto de administrador retornado por `Manager()` controla un proceso de servidor que contiene objetos de Python y permite que otros procesos los manipulen usando proxies.

Un administrador retornado por `Manager()` soportará tipos de clases como *list*, *dict*, *Namespace*, *Lock*, *RLock*, *Semaphore*, *BoundedSemaphore*, *Condition*, *Event*, *Barrier*, *Queue*, *Value* y *Array*. Por ejemplo,

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
```

(continué en la próxima página)

(proviene de la página anterior)

```
d = manager.dict()
l = manager.list(range(10))

p = Process(target=f, args=(d, l))
p.start()
p.join()

print(d)
print(l)
```

imprimirá

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Los administradores de procesos del servidor son más flexibles que el uso de objetos de memoria compartida porque pueden hacerse para admitir tipos de objetos arbitrarios. Por lo tanto, un solo administrador puede ser compartido por procesos en diferentes ordenadores a través de una red. Sin embargo, son más lentos que usar memoria compartida.

Usando una piscina de trabajadores (*pool of workers*)

La clase `Pool` representa procesos de piscina de trabajadores (*pool of workers*). Tiene métodos que permiten que las tareas se descarguen a los procesos de trabajo de diferentes maneras.

Por ejemplo:

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # 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)

        # 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:
```

(continué en la próxima página)

(proviene de la página anterior)

```
print(res.get(timeout=1))
except TimeoutError:
    print("We lacked patience and got a multiprocessing.TimeoutError")

print("For the moment, the pool remains available for more work")

# exiting the 'with'-block has stopped the pool
print("Now the pool is closed and no longer available")
```

Tenga en cuenta que los métodos de una piscina (*pool*) solo deben ser utilizados por el proceso que lo creó.

Nota: La funcionalidad en este paquete requiere que los procesos hijos (*children*) puedan importar el módulo `__main__`. Esto está cubierto en *Pautas de programación* sin embargo, vale la pena señalarlo aquí. Esto significa que algunos ejemplos, como `multiprocessing.pool.Pool` no funcionarán en el intérprete interactivo. Por ejemplo:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> with p:
...     p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
Traceback (most recent call last):
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'
```

(Si intenta esto, en realidad generará tres trazas completas intercaladas de forma semialeatoria, y luego tendrá que detener el proceso principal de alguna manera)

17.2.2 Referencia

El paquete *multiprocessing* mayoritariamente replica la API del módulo *threading*.

Process y excepciones

class `multiprocessing.Process` (*group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None*)

Los objetos de proceso representan la actividad que se ejecuta en un proceso separado. La clase *Process* tiene equivalentes para todos los métodos *threading.Thread*.

El constructor siempre debe llamarse con argumentos de palabras clave. *group* siempre debe ser *None*; existe únicamente por compatibilidad con *threading.Thread*. *target* es el objeto invocable a ser llamado por el método *run()*. El valor predeterminado es *None*, lo que significa que nada es llamado. *name* es el nombre del proceso (consulte *name* para más detalles). *args* es la tupla de argumento para la invocación de destino. *kwargs* es un diccionario de argumentos de palabras clave para la invocación de destino. Si se proporciona, el argumento *daemon* solo de palabra clave establece el proceso *daemon* en *True* o *False*. Si *None* (el valor predeterminado), este indicador se heredará del proceso de creación.

Por defecto, ningún argumento es pasado a *target*.

Si una subclase anula al constructor, debe asegurarse de que invoca al constructor de la clase base (`Process.__init__()`) antes de hacer cualquier otra cosa al proceso.

Distinto en la versión 3.3: Añadido el argumento *daemon*.

run()

Método que representa la actividad del proceso.

Puede anular este método en una subclase. El método estándar `run()` invoca el objeto invocable pasado al constructor del objeto como argumento objetivo, si lo hay, con argumentos posicionares y de palabras clave tomados de los argumentos *args* y *kwargs*, respectivamente.

start()

Comienza la actividad del proceso.

Esto debe llamarse como máximo una vez por objeto de proceso. Organiza la invocación del método `run()` del objeto en un proceso separado.

join([timeout])

Si el argumento opcional *timeout* es `None` (el valor predeterminado), el método se bloquea hasta que el proceso cuyo método `join()` se llama termina. Si *timeout* es un número positivo, bloquea como máximo *timeout* segundos. Tenga en cuenta que el método retorna `None` si su proceso finaliza o si el método agota el tiempo de espera. Verifique el proceso *exitcode* para determinar si terminó.

Un proceso puede unirse muchas veces.

Un proceso no puede unirse a sí mismo porque esto provocaría un punto muerto. Es un error intentar unirse a un proceso antes de que se haya iniciado.

name

El nombre del proceso. El nombre es una cadena utilizada solo con fines de identificación. No tiene semántica. Múltiples procesos pueden tener el mismo nombre.

El nombre inicial es establecido por el constructor. Si no se proporciona un nombre explícito al constructor, se forma un nombre usando “*Process-N:sub:1:N:sub:2:...:N:sub:k*” se construye, donde cada *N:sub:k* es el *N*-ésimo hijo del parental.

is_alive()

Retorna si el proceso está vivo.

Aproximadamente, un objeto de proceso está vivo desde el momento en que el método `start()` retorna hasta que finaliza el proceso hijo.

daemon

La bandera del proceso *daemon*, es un valor booleano. Esto debe establecerse antes de que `start()` sea llamado.

El valor inicial se hereda del proceso de creación.

Cuando un proceso sale, intenta terminar todos sus procesos demoníacos (*daemonic*) hijos.

Tenga en cuenta que un proceso demoníaco (*daemonic*) no puede crear procesos hijos. De lo contrario, un proceso demoníaco (*daemonic*) dejaría a sus hijos huérfanos si se termina cuando finaliza su proceso parental. Además, estos **no** son *daemons* o servicios de Unix, son procesos normales que finalizarán (y no se unirán) si los procesos no demoníacos han salido.

Además de API `threading.Thread`, los objetos `Process` también admiten los siguientes atributos y métodos:

pid

Retorna el ID del proceso. Antes de que se genere el proceso, esto será `None`.

exitcode

El código de salida del hijo. Esto será `None` si el proceso aún no ha terminado. Un valor negativo *-N* indica que el hijo fue terminado por la señal *N*.

authkey

La clave de autenticación del proceso (una cadena de bytes).

Cuando el `multiprocessing` se inicializa el proceso principal se le asigna una cadena aleatoria usando `os.urandom()`.

Cuando se crea un objeto `Process`, este heredará la clave de autenticación de su proceso parental, aunque esto puede cambiarse configurando `authkey` a otra cadena de bytes.

Consulte *Llaves de autenticación*.

sentinel

Un identificador numérico de un objeto del sistema que estará «listo» cuando finalice el proceso.

Se puede usar este valor si desea esperar varios eventos a la vez usando `multiprocessing.connection.wait()`. De lo contrario, llamar a `join()` es más simple.

En Windows, este es un sistema operativo manejable con la familia de llamadas API `WaitForSingleObject` y `WaitForMultipleObjects`. En Unix, este es un descriptor de archivo utilizable con primitivas del módulo `select`.

Nuevo en la versión 3.3.

terminate()

Terminar el proceso. En Unix, esto se hace usando la señal `SIGTERM`; en Windows se utiliza `TerminateProcess()`. Tenga en cuenta que los manejadores de salida y finalmente las cláusulas, etc., no se ejecutarán.

Tenga en cuenta que los procesos descendientes del proceso *no* finalizarán – simplemente quedarán huérfanos.

Advertencia: Si este método se usa cuando el proceso asociado está usando una tubería (*pipe*) o una cola (*queue*), entonces la tubería o la cola pueden corromperse y pueden quedar inutilizables por otro proceso. Del mismo modo, si el proceso ha adquirido un *lock* o un semáforo, etc., su finalización puede provocar el bloqueo de otros procesos.

kill()

Siempre como `terminate()` pero utilizando la señal `SIGKILL` en Unix.

Nuevo en la versión 3.7.

close()

El cierre del objeto `Process`, libera todos los recursos asociados a él. `ValueError` se lanza si el proceso subyacente aún se está ejecutando. Una vez `close()` se retorna con éxito, la mayoría de los otros métodos y atributos del objeto `Process` lanzará `ValueError`.

Nuevo en la versión 3.7.

Tenga en cuenta que los métodos `start()`, `join()`, `is_alive()`, `terminate()` y `exitcode` deberían solo ser llamados por el proceso que creó el objeto del proceso.

Ejemplo de uso de algunos métodos de la `Process`:

```
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<Process ... initial> False
>>> p.start()
>>> print(p, p.is_alive())
<Process ... started> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<Process ... stopped exitcode=-SIGTERM> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.ProcessError

La clase base de todas las excepciones *multiprocessing*.

exception multiprocessing.BufferTooShort

La excepción `Connection.recv_bytes_into()` es lanzada cuando el objeto de búfer suministrado es demasiado pequeño para el mensaje leído.

Si `e` es una instancia de *BufferTooShort* entonces `e.args[0]` dará el mensaje como una cadena de *bytes*.

exception multiprocessing.AuthenticationError

Lanzada cuando hay un error de autenticación.

exception multiprocessing.TimeoutError

Lanzada por métodos con un tiempo de espera (*timeout*) cuando este expira.

Tuberías (Pipes) y Colas (Queues)

Cuando se usan múltiples procesos, uno generalmente usa el paso de mensajes para la comunicación entre procesos y evita tener que usar primitivas de sincronización como *locks*.

Para pasar mensajes se puede usar *Pipe()* (para una conexión entre dos procesos) o una cola (*queue*)(que permite múltiples productores y consumidores).

Los tipos *Queue*, *SimpleQueue* y *JoinableQueue* son colas multi-productor, multi-consumidor FIFO (primero en entrar, primero en salir) modeladas en *queue.Queue* en la biblioteca estándar. Se diferencian en que *Queue* carece de *task_done()* y *join()* métodos introducidos en Python 2.5 *queue.Queue* class.

Si usa *JoinableQueue*, entonces **debe** llamar a *JoinableQueue.task_done()* para cada tarea eliminada de la cola (*queue*) o de lo contrario el semáforo utilizado para contar el número de tareas sin terminar puede eventualmente desbordarse, lanzando un excepción.

Tenga en cuenta que también se puede crear una cola compartida mediante el uso de un objeto de administrador – consulte *Administradores (Managers)*.

Nota: *multiprocessing* utiliza las excepciones habituales *queue.Empty* y *queue.Full* para indicar un tiempo de espera. No están disponibles en el espacio de nombres *multiprocessing*, por lo que debe importarlos desde *queue*.

Nota: Cuando un objeto se coloca en una cola (*queue*), el objeto se serializa (*pickled*) y luego un subproceso de fondo vacía los datos serializados a una tubería (*pipe*) subyacente. Esto tiene algunas consecuencias que son un poco sorprendentes, pero no deberían causar dificultades prácticas: si realmente causan molestias, se puede usar una cola creada con un *manager*.

- (1) Después de poner un objeto en una cola vacía, puede haber un retraso infinitesimal antes de que el método de la cola *empty()* retorne *False* y *get_nowait()* puede retornar sin lanzar *queue.Empty*.
 - (2) Si varios procesos están poniendo en cola objetos, es posible que los objetos se reciban en el otro extremo fuera de orden. Sin embargo, los objetos en cola por el mismo proceso siempre estarán en el orden esperado entre sí.
-

Advertencia: Si se termina un proceso usando *Process.terminate()* o *os.kill()* mientras intenta usar una clase *Queue*, es probable que los datos de la cola(*queue*) se corrompan. Esto puede hacer que cualquier otro proceso obtenga una excepción cuando intente usar la cola más adelante.

Advertencia: Como se mencionó anteriormente, si un proceso hijo ha puesto elementos en una cola (*queue*) (y no ha utilizado `JoinableQueue.cancel_join_thread()`), entonces ese proceso no terminará hasta que todos los elementos almacenados en búfer hayan sido vaciados a la tubería (*pipe*).

Esto significa que si intenta unirse a ese proceso, puede obtener un punto muerto a menos que esté seguro de que todos los elementos que se han puesto en la cola (*queue*) se han consumido. Del mismo modo, si el proceso hijo no es *daemonic*, el proceso parental puede bloquearse en la salida cuando intenta unir todos sus elementos hijos no *daemonic*.

Tenga en cuenta que una cola (*queue*) creada con un administrador no tiene este problema. Consulte *Pautas de programación*.

Para ver un ejemplo del uso de colas para la comunicación entre procesos, consulte *Ejemplos*.

`multiprocessing.Pipe([duplex])`

Retorna un par de objetos “(*conn1*, *conn2*)” de la *Connection* que representan los extremos de una tubería (*pipe*).

Si *duplex* es `True` (el valor predeterminado), entonces la tubería (*pipe*) es bidireccional. Si *duplex* es `False`, entonces la tubería es unidireccional: *conn1* solo se puede usar para recibir mensajes y *conn2* solo se puede usar para enviar mensajes.

`class multiprocessing.Queue([maxsize])`

Retorna un proceso de cola (*queue*) compartida implementado utilizando una tubería (*pipe*) y algunos candados/semáforos (*locks/semaphores*). Cuando un proceso pone por primera vez un elemento en la cola, se inicia un hilo alimentador que transfiere objetos desde un búfer a la tubería.

Las excepciones habituales *queue.Empty* y *queue.Full* del módulo de la biblioteca estándar *queue* se generan para indicar tiempos de espera.

La *Queue* implementa todos los métodos de la *queue.Queue* excepto por *task_done()* y *join()*.

`qsize()`

Retorna el tamaño aproximado de la cola (*queue*). Debido a la semántica multiproceso/multiprocesamiento, este número no es confiable.

Tenga en cuenta que esto puede lanzar *NotImplementedError* en plataformas Unix como Mac OS X donde *sem_getvalue()* no está implementado.

`empty()`

Retorna `True` si la cola (*queue*) está vacía, de lo contrario retorna `False`. Debido a la semántica multiproceso/multiprocesamiento, esto no es confiable.

`full()`

Retorna `True` si la cola (*queue*) está llena, de lo contrario retorna `False`. Debido a la semántica multiproceso/multiprocesamiento, esto no es confiable.

`put(obj[, block[, timeout]])`

Pone *obj* en la cola. Si el argumento opcional *block* es `True` (el valor predeterminado) y *timeout* es `None` (el valor predeterminado), se bloquea si es necesario hasta que haya un espacio disponible. Si *timeout* es un número positivo, bloquea a lo sumo *timeout* segundos y genera la excepción *queue.Full* si no hay espacio libre disponible en ese tiempo. De lo contrario (*block* es `False`), y coloca un elemento en la cola si hay un espacio libre disponible de inmediato, de lo contrario, genera la excepción *queue.Full* (*timeout* se ignora en ese caso).

Distinto en la versión 3.8: Si la cola (*queue*) está cerrada, *ValueError* se lanza en lugar de *AssertionError*.

`put_nowait(obj)`

Equivalente a `put(obj, False)`.

`get([block[, timeout]])`

Elimina y retorna un artículo de la cola (*queue*). Si un argumento opcional *block* es `True` (el valor predeterminado) y el *timeout* es `None` (el valor predeterminado), es bloqueado si es necesario hasta que

un elemento esté disponible. Si el *timeout* es un número positivo, bloquea a lo sumo segundos y genera la excepción `queue.Empty` si no había ningún elemento disponible dentro de ese tiempo. De lo contrario (el bloque es `False`), retorna un elemento si hay uno disponible de inmediato, de lo contrario, levante la excepción `queue.Empty` (*timeout* se ignora en ese caso).

Distinto en la versión 3.8: Si la cola está cerrada, se lanza `ValueError` en lugar de `OSError`.

get_nowait()

Equivalente a `get(False)`.

La `multiprocessing.Queue` tiene algunos métodos adicionales que no se encuentran en `queue.Queue`. Estos métodos suelen ser innecesarios para la mayoría de los códigos:

close()

Indica que el proceso actual no colocará más datos en esta cola (*queue*). El subproceso en segundo plano se cerrará una vez que haya vaciado todos los datos almacenados en la tubería (*pipe*). Esto se llama automáticamente cuando la cola es recolectada por el recolector de basura.

join_thread()

Unifica al hilo de fondo. Esto solo se puede usar después de que se ha llamado a `close()`. Esto se bloquea hasta que salga el hilo de fondo, asegurando que todos los datos en el búfer se hayan vaciado a la tubería (*pipe*).

Por defecto, si un proceso no es el creador de la cola (*queue*), al salir intentará unirse al hilo de fondo de la cola. El proceso puede llamar a `cancel_join_thread()` para hacer que el `join_thread()` no haga nada.

cancel_join_thread()

Evita que `join_thread()` bloquee. En particular, esto evita que el subproceso en segundo plano se una automáticamente cuando finaliza el proceso; consulte `join_thread()`.

Un mejor nombre para este método podría ser `allow_exit_without_flush()`. Es probable que provoque la pérdida de datos en cola (*queue*), y es casi seguro que no necesitará usarlos. Realmente solo está allí si necesita que el proceso actual salga inmediatamente sin esperar a vaciar los datos en cola en la tubería (*pipe*) subyacente, y no le importan los datos perdidos.

Nota: La funcionalidad de esta clase requiere una implementación de semáforo compartido en funcionamiento en el sistema operativo que es huésped (*host*). Sin uno, la funcionalidad en esta clase se deshabilitará, y los intentos de instanciar a `Queue` resultarán en `ImportError`. Consulte [bpo-3770](#) para información adicional. Lo mismo es válido para cualquiera de los tipos de cola especializados que se enumeran a continuación.

class multiprocessing.SimpleQueue

Es un tipo simplificado `Queue`, muy similar a un *lock* de `Pipe`.

empty()

Retorna `True` si la cola (*queue*) está vacía, de otra manera retorna `False`.

get()

Eliminar y retornar un artículo de la cola (*queue*).

put(item)

Pone *item* en la cola.

class multiprocessing.JoinableQueue([maxsize])

`JoinableQueue`, una subclase `Queue`, es una cola (*queue*) que además tiene los métodos `task_done()` y `join()`.

task_done()

Indica que una tarea anteriormente en cola (*queue*) está completa. Usado por los consumidores de la cola. Por cada `get()` utilizado para recuperar una tarea, una llamada posterior a `task_done()` le dice a la cola que el procesamiento de la tarea se ha completado.

Si un `join()` se está bloqueando actualmente, se reanudará cuando se hayan procesado todos los elementos (lo que significa que `task_done()` es llamado para cada elemento que había sido puesto en

cola (*queue*) por `put()`.

Lanza un `ValueError` si es llamado más veces que elementos hay en una cola.

`join()`

Se bloquea hasta que todos los elementos en una cola han sido recibidos y procesados.

El recuento de tareas no finalizadas aumenta cada vez que se agrega un elemento a la cola. El recuento disminuye cada vez que un consumidor llama a `task_done()` para indicar que el artículo se recuperó y todo el trabajo en él está completo. Cuando el recuento de tareas inacabadas cae a cero, `join()` se desbloquea.

Miscelánea

`multiprocessing.active_children()`

Retorna una lista con todos los hijos del proceso actual.

Llamar a esto tiene el efecto secundario de «unir» (*joining*) cualquier proceso que ya haya finalizado.

`multiprocessing.cpu_count()`

Retorna el número de CPU en el sistema.

Este número no es equivalente al número de CPU que puede utilizar el proceso actual. El número de CPU utilizables se puede obtener con `len(os.sched_getaffinity(0))`

Puedo lanzar `NotImplementedError`.

Ver también:

`os.cpu_count()`

`multiprocessing.current_process()`

Retorna el objeto de la `Process` correspondiente al proceso actual.

Un análogo de la `threading.current_thread()`.

`multiprocessing.parent_process()`

Retorna el objeto de la `Process` correspondiente al proceso parental de `current_process()`. Para el proceso principal, `parent_process` será `None`.

Nuevo en la versión 3.8.

`multiprocessing.freeze_support()`

Agrega soporte para cuando un programa que utiliza `multiprocessing` se haya congelado para producir un ejecutable de Windows. (Ha sido probado con `py2exe`, `PyInstaller` y `cx_Freeze`.)

Es necesario llamar a esta función inmediatamente después de la línea principal del módulo `if __name__ == "__main__":`. Por ejemplo:

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

Si se omite la línea `freeze_support()` entonces se intenta comenzar el ejecutable congelado que lanzará `RuntimeError`.

La llamada de `freeze_support()` no tiene efecto cuando es invocada por cualquier sistema operativo que no sea Windows. Además, si el módulo ha sido ejecutado en un intérprete de Python en Windows (y el programa no se ha congelado) entonces `freeze_support()` no tiene efecto.

`multiprocessing.get_all_start_methods()`

Retorna una lista de los métodos de inicio admitidos, el primero de los cuales es el predeterminado. Los posibles métodos de inicio son 'fork', 'spawn' y 'forkserver'. En Windows solo está disponible 'spawn'. En Unix, 'fork' y 'spawn' siempre son compatibles, siendo 'fork' el valor predeterminado.

Nuevo en la versión 3.4.

`multiprocessing.get_context(method=None)`

Retorna un objeto de contexto que tiene los mismos atributos que el módulo `multiprocessing`.

Si el método `method` es `None` entonces el contexto predeterminado es retornado. Por lo contrario, `method` debería ser 'fork', 'spawn', 'forkserver'. Se lanza `ValueError` if el método de inicio no esta disponible.

Nuevo en la versión 3.4.

`multiprocessing.get_start_method(allow_none=False)`

Retorna el nombre del método de inicio que es utilizado para iniciar procesos.

Si el método de inicio no se ha solucionado y `allow_none` es falso, entonces el método de inicio se fija al predeterminado y se retorna el nombre. Si el método de inicio no se ha solucionado y `allow_none` es verdadero, se retorna `None`.

The return value can be 'fork', 'spawn', 'forkserver' or `None`. 'fork' is the default on Unix, while 'spawn' is the default on Windows and macOS.

Distinto en la versión 3.8: En macOS, el método de inicio `spawn` ahora es el predeterminado. El método de inicio `fork` debe considerarse inseguro ya que puede provocar bloqueos del subprocesso. Consulte [bpo-33725](#).

Nuevo en la versión 3.4.

`multiprocessing.set_executable()`

Establece la ruta del intérprete de Python para usar cuando se inicia un proceso secundario. (Por defecto se utiliza `sys.executable`). Los integradores probablemente necesiten hacer algo como

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

antes ellos pueden crear procesos hijos.

Distinto en la versión 3.4: Ahora es compatible con Unix cuando se usa el método de inicio 'spawn'.

`multiprocessing.set_start_method(method)`

Se establece el método que se debe usar para iniciar procesos secundarios. `method` puede ser 'fork', 'spawn' o 'forkserver'.

Tenga en cuenta que esto debería llamarse como máximo una vez, y debería protegerse dentro de la cláusula `if __name__ == '__main__':` del módulo principal.

Nuevo en la versión 3.4.

Nota: `multiprocessing` no contiene análogos de `threading.active_count()`, `threading.enumerate()`, `threading.settrace()`, `threading.setprofile()`, `threading.Timer`, o `threading.local`.

Objetos de conexión *Connection Objects*

Los objetos de conexión permiten el envío y la recepción de objetos serializables (*pickable*) o cadenas de caracteres seleccionables. Pueden considerarse como sockets conectados orientados a mensajes.

Los objetos de conexión usualmente son creados usando *Pipe* – ver también *Oyentes y Clientes (Listeners and Clients)*.

class multiprocessing.connection.**Connection**

send (*obj*)

Envía un objeto al otro extremo de la conexión que debe leerse usando *recv()*.

El objeto debe ser serializable (*pickable*). Los serializados (*pickable*) muy grandes (aproximadamente 32 MiB+, aunque depende del sistema operativo) pueden generar una excepción *ValueError*.

recv ()

Retorna un objeto enviado desde el otro extremo de la conexión usando *send()*. Se bloquea hasta que haya algo para recibir. Se lanza *EOFError* si no queda nada por recibir y el otro extremo está cerrado.

fileno ()

Retorna el descriptor de archivo o identificador utilizado por la conexión.

close ()

Cierra la conexión.

Esto se llama automáticamente cuando la conexión es basura recolectada.

poll ([*timeout*])

Retorna si hay datos disponibles para leer.

Si no se especifica *timeout*, se retornará de inmediato. Si *timeout* es un número, esto especifica el tiempo máximo en segundos para bloquear. Si *timeout* es *None*, se usa un tiempo de espera infinito.

Tenga en cuenta que se pueden sondear varios objetos de conexión a la vez utilizando *multiprocessing.connection.wait()*.

send_bytes (*buffer*[, *offset*[, *size*]])

Envía datos de *bytes* desde a *bytes-like object* como un mensaje completo.

Si se da *offset*, los datos se leen desde esa posición en *buffer*. Si se da *size* entonces se leerán muchos *bytes* del búfer. Los *buffers* muy grandes (aproximadamente 32 MiB+, aunque depende del sistema operativo) pueden generar una excepción *ValueError*.

recv_bytes ([*maxlength*])

Retorna un mensaje completo de datos de *bytes* enviados desde el otro extremo de la conexión como una cadena de caracteres. Se bloquea hasta que haya algo para recibir. Aumenta *EOFError* si no queda nada por recibir y el otro extremo se ha cerrado.

Si se especifica *maxlength* y el mensaje es más largo que *maxlength*, entonces se lanza un *OSError* y la conexión ya no será legible.

Distinto en la versión 3.3: Esta función solía lanzar un *IOError*, que ahora es un alias de *OSError*.

recv_bytes_into (*buffer*[, *offset*])

Lee en *buffer* un mensaje completo de datos de *bytes* enviados desde el otro extremo de la conexión y retorne el número de *bytes* en el mensaje. Se bloquea hasta que haya algo para recibir. Si no queda nada por recibir y el otro extremo está cerrándose se lanza *EOFError*.

buffer debe ser un escribible *bytes-like object*. Si se proporciona *offset* el mensaje se escribirá en el búfer desde esa posición. La compensación debe ser un número entero no negativo menor que la longitud de *buffer* (en *bytes*).

Si el búfer es demasiado corto, se genera una excepción *BufferTooShort* y el mensaje completo está disponible como *e.args[0]* donde *e* es la instancia de excepción.

Distinto en la versión 3.3: Los objetos de conexión ahora pueden transferirse entre procesos usando `Connection.send()` y `Connection.recv()`.

Nuevo en la versión 3.3: Los objetos de conexión ahora admiten el protocolo de administración de contexto – consulte *Tipos Gestores de Contexto*. `__enter__()` retorna el objeto de conexión, y `__exit__()` llama a `close()`.

Por ejemplo:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'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])
```

Advertencia: El método `Connection.recv()` desempaqueta automáticamente los datos que recibe, lo que puede ser un riesgo de seguridad a menos que pueda confiar en el proceso que envió el mensaje.

Por lo tanto, a menos que el objeto de conexión se haya producido usando `Pipe()` solo debe usar los métodos `recv()` y `send()` después de realizar algún tipo de autenticación. Consulte *Llaves de autenticación*.

Advertencia: Si se mata un proceso mientras intenta leer o escribir en una tubería (*pipe*), entonces es probable que los datos en la tubería se corrompan, porque puede ser imposible estar seguro de dónde se encuentran los límites del mensaje.

Primitivas de sincronización (*Synchronization primitives*)

En general, las primitivas de sincronización no son tan necesarias en un programa multiproceso como en un programa *multihilos* (*multithreaded*). Consulte la documentación para *threading*.

Tenga en cuenta que también se pueden crear primitivas de sincronización utilizando un objeto administrador – consulte *Administradores (Managers)*.

class `multiprocessing.Barrier` (`parties`[, `action`[, `timeout`]])

Un objeto de barrera: un clon de `threading.Barrier`.

Nuevo en la versión 3.3.

class `multiprocessing.BoundedSemaphore` ([`value`])

Un objeto semáforo (*semaphore object*) acotado: un análogo cercano de la `threading.BoundedSemaphore`.

Existe una diferencia solitaria de su análogo cercano: el primer argumento de su método `acquire` es nombrado *block*, es consistente con `Lock.acquire()`.

Nota: En Mac OS X, esto no se puede distinguir de `Semaphore` porque `sem_getvalue()` no está implementado en esa plataforma.

class multiprocessing.Condition ([lock])

Una variable de condición: un alias para la `threading.Condition`.

Si se especifica `lock`, entonces debería ser una `Lock` o `RLock` objeto de `multiprocessing`.

Distinto en la versión 3.3: El método `wait_for()` fue a añadido.

class multiprocessing.Event

Un clon de `threading.Event`.

class multiprocessing.Lock

Un objeto candado no recursivo: un análogo cercano de `threading.Lock`. Una vez que un proceso o subproceso ha adquirido un bloqueo, los intentos posteriores de adquirirlo de cualquier proceso o subproceso se bloquearán hasta que se libere; cualquier proceso o hilo puede liberarlo. Los conceptos y comportamientos de `threading.Lock` como se aplica a los subprocesos se replican aquí en `multiprocessing.Lock` como se aplica a los procesos o subprocesos, excepto como se indica.

Tenga en cuenta que `Lock` es en realidad una función de fábrica que retorna una instancia de `multiprocessing.synchronize.Lock` inicializada con un contexto predeterminado.

La `Lock` soporta el protocolo `context manager` y, por lo tanto, se puede usar en la declaración `with`.

acquire (block=True, timeout=None)

Adquiriendo un candado (`lock`), bloqueante o no bloqueante.

Con el argumento `block` establecido en `True` (el valor predeterminado), la llamada al método se bloqueará hasta que el bloqueo esté en un estado desbloqueado, luego configúrelo como bloqueado y retorne `True`. Tenga en cuenta que el nombre de este primer argumento difiere del que aparece en `threading.Lock.acquire()`.

Con el argumento `block` establecido en `False`, la llamada al método no se bloquea. Si el bloqueo está actualmente en un estado bloqueado, retorna `False`; de lo contrario, configure el bloqueo en un estado bloqueado y retorne `True`.

Cuando se invoca con un valor positivo de punto flotante para `timeout`, bloquea como máximo el número de segundos especificado por `timeout` siempre que no se pueda obtener el bloqueo. Las invocaciones con un valor negativo para `timeout` de cero. Las invocaciones con un valor `timeout` de `None` (el valor predeterminado) establecen el período de tiempo de espera en infinito. Tenga en cuenta que el tratamiento de valores negativos o `None` para `timeout` difiere del comportamiento implementado en `threading.Lock.acquire()`. El argumento `timeout` no tiene implicaciones prácticas si el argumento `block` se establece en `False` y, por lo tanto, se ignora. Retorna `True` si se ha adquirido el candado o `False` si ha transcurrido el tiempo de espera.

release ()

Suelta un candado. Esto se puede llamar desde cualquier proceso o subproceso, no solo desde el proceso o subproceso que originalmente adquirió el candado.

El comportamiento es el mismo que en `threading.Lock.release()` excepto que cuando se invoca en un bloqueo desbloqueado, se genera a `ValueError`.

class multiprocessing.RLock

Un objeto de candado recursivo: un análogo cercano a `threading.RLock`. El proceso o el hilo que lo adquirió debe liberar un candado recursivo. Una vez que un proceso o subproceso ha adquirido un candado recursivo, el mismo proceso o subproceso puede volver a adquirirlo sin bloquearlo; ese proceso o hilo debe liberarlo una vez por cada vez que se haya adquirido.

Tenga en cuenta que `RLock` es en realidad una función de fábrica que retorna una instancia de `multiprocessing.synchronize.RLock` inicializada con un contexto predeterminado.

La `RLock` admite el protocolo `context manager` y, por lo tanto, puede usarse en `with`.

acquire (block=True, timeout=None)

Adquiriendo un candado (`lock`), bloqueante o no bloqueante.

Cuando se invoca con el argumento `block` establecido en `True`, bloquea hasta que el candado esté en un estado desbloqueado (que no sea propiedad de ningún proceso o subproceso) a menos que el candado o el subproceso actual ya sea de su propiedad. El proceso o subproceso actual se apropia del candado (si aún no

lo tiene) y el nivel de recursión dentro de este aumenta en uno, lo que da como resultado un valor de retorno de `True`. Tenga en cuenta que hay varias diferencias en el comportamiento de este primer argumento en comparación con la implementación de `threading.RLock.acquire()`, comenzando con el nombre del argumento en sí.

Cuando se invoca con el argumento `block` establecido en `False`, no bloquea. Si el candado ya ha sido adquirido (y por lo tanto es propiedad) de otro proceso o subproceso, el proceso o subproceso actual no se apropia y el nivel de recursión dentro del candado no cambia, lo que resulta en un valor de retorno de `False`. Si el candado está en un estado desbloqueado, el proceso o subproceso actual toma posesión y el nivel de recurrencia se incrementa, lo que resulta en un valor de retorno de `True`.

El uso y los comportamientos del argumento `timeout` son los mismos que en `Lock.acquire()`. Tenga en cuenta que algunos de estos comportamientos de `timeout` difieren de los comportamientos implementados en `threading.RLock.acquire()`.

release()

Libera un candado, disminuyendo el nivel de recursión. Si después del decremento el nivel de recursión es cero, restablece el candado a desbloqueado (que no sea propiedad de ningún proceso o subproceso) y si se bloquean otros procesos o subprocesos esperando que el candado se desbloquee, permite que continúe exactamente uno de ellos. Si después del decremento el nivel de recursión sigue siendo distinto de cero, el candado permanece bloqueado y pertenece al proceso de llamada o subproceso.

Solo llame a este método cuando el proceso o subproceso de llamada sea el propietario del candado. Se lanza un `AssertionError` si se llama a este método mediante un proceso o subproceso que no sea el propietario o si el candado está en un estado desbloqueado (sin propietario). Tenga en cuenta que el tipo de excepción planteada en esta situación difiere del comportamiento implementado en `threading.RLock.release()`.

class multiprocessing.Semaphore([value])

Un objeto semáforo: un análogo cercano de `threading.Semaphore`.

Existe una diferencia solitaria de su análogo cercano: el primer argumento de su método `acquire` es nombrado `block`, es consistente con `Lock.acquire()`.

Nota: En Mac OS X, `sem_timedwait` no es compatible, por lo que llamar a `acquire()` con un tiempo de espera emulará el comportamiento de esa función utilizando un bucle inactivo.

Nota: Si la señal `SIGINT` generada por `Ctrl-C` llega mientras el hilo principal está bloqueado por una llamada a `BoundedSemaphore.acquire()`, `Lock.acquire()`, `RLock.acquire()`, `Semaphore.acquire()`, `Condition.acquire()` o `Condition.wait()`, la llamada se interrumpirá inmediatamente y `KeyboardInterrupt` se lanzará.

Esto difiere del comportamiento de `threading` donde `SIGINT` será ignorado mientras las llamadas de candado equivalentes están en progreso.

Nota: Parte de la funcionalidad de este paquete requiere una implementación de semáforo compartido que funcione en el sistema operativo. Sin uno, el módulo `multiprocessing.synchronize` se desactivará, y los intentos de importarlo darán como resultado `ImportError`. Consulte [bpo-3770](#) para información adicional.

Objetos compartidos `ctypes`

Es posible crear objetos compartidos utilizando memoria compartida que puede ser heredada por procesos secundarios.

`multiprocessing.Value` (*typecode_or_type*, **args*, *lock=True*)

Retorna un objeto `ctypes` asignado desde la memoria compartida. Por defecto, el valor de retorno es en realidad un contenedor sincronizado para el objeto. Se puede acceder al objeto en sí a través del atributo `value` de la `Value`.

typecode_or_type determina el tipo del objeto retornado: es un tipo `ctypes` o un código de tipo de un carácter del tipo utilizado por el módulo `array`. **args* se pasa al constructor para el tipo.

Si *lock* es `True` (el valor predeterminado), se crea un nuevo objeto de candado recursivo para sincronizar el acceso al valor. Si *lock* es un objeto `Lock` o `RLock`, se usará para sincronizar el acceso al valor. Si *lock* es `False`, entonces el acceso al objeto retornado no estará protegido automáticamente por un candado, por lo que no será necesariamente «proceso-seguro».

Operaciones como `+=` que implican una lectura y escritura no son atómicas. Entonces, si, por ejemplo, desea incrementar atómicamente un valor compartido, es insuficiente simplemente hacer:

```
counter.value += 1
```

Suponiendo que el candado asociado es recursivo (que es por defecto), puede hacer

```
with counter.get_lock():
    counter.value += 1
```

Véase que *lock* es un argumento de solo una palabra clave.

`multiprocessing.Array` (*typecode_or_type*, *size_or_initializer*, *, *lock=True*)

Retorna una matriz `ctypes` asignada desde la memoria compartida. Por defecto, el valor de retorno es en realidad un contenedor sincronizado para el arreglo.

typecode_or_type determina el tipo de los elementos de la matriz retornada: es un tipo de tipo `ctypes` o un código de tipo de un carácter del tipo utilizado por el módulo `array`. Si *size_or_initializer* es un número entero, entonces determina la longitud de la matriz, y la matriz se pondrá a cero inicialmente. De lo contrario, *size_or_initializer* es una secuencia que se utiliza para inicializar la matriz y cuya longitud determina la longitud de la matriz.

Si *lock* es `True` (el valor predeterminado), se crea un nuevo objeto de bloqueo para sincronizar el acceso al valor. Si *lock* es un objeto `Lock` o `RLock`, se usará para sincronizar el acceso al valor. Si *lock* es `False`, entonces el acceso al objeto retornado no estará protegido automáticamente por un candado, por lo que no será necesariamente «proceso seguro».

Véase que *lock* es un argumento de solo una palabra clave.

Tenga en cuenta que una matriz de `ctypes.c_char` tiene atributos `value` y `raw` que le permiten a uno usarlo para almacenar y recuperar cadenas de caracteres.

El módulo `multiprocessing.sharedctypes`

El módulo `multiprocessing.sharedctypes` proporciona funciones para asignar objetos `ctypes` de la memoria compartida que pueden ser heredados por procesos secundarios.

Nota: Aunque es posible almacenar un puntero en la memoria compartida, recuerde que esto se referirá a una ubicación en el espacio de direcciones de un proceso específico. Sin embargo, es muy probable que el puntero sea inválido en el contexto de un segundo proceso y tratar de desreferenciar el puntero del segundo proceso puede causar un bloqueo.

`multiprocessing.sharedctypes.RawArray` (*typecode_or_type*, *size_or_initializer*)

Retorna una matriz *ctypes* asignada desde la memoria compartida.

typecode_or_type determina el tipo de los elementos de la matriz retornada: es un tipo de tipo *ctypes* o un código de tipo de un carácter del tipo utilizado por el módulo `array`. Si *size_or_initializer* es un entero, entonces determina la longitud de la matriz, y la matriz se pondrá a cero inicialmente. De lo contrario, *size_or_initializer* es una secuencia que se usa para inicializar la matriz y cuya longitud determina la longitud del arreglo.

Tenga en cuenta que configurar y obtener un elemento es potencialmente no atómico – utiliza `Array()` en su lugar para asegurarse de que el acceso se sincronice automáticamente mediante un candado.

`multiprocessing.sharedctypes.RawValue` (*typecode_or_type*, **args*)

Retorna un objeto *ctypes* asignado desde la memoria compartida.

typecode_or_type determina el tipo del objeto retornado: es un tipo *ctypes* o un código de tipo de un carácter del tipo utilizado por el módulo `array`. **args* se pasa al constructor para el tipo.

Tenga en cuenta que configurar y obtener el valor es potencialmente no atómico – use `Value()` en su lugar para asegurarse de que el acceso se sincronice automáticamente mediante un candado.

Tenga en cuenta que una matriz de `ctypes.c_char` tiene atributos `value` y `raw` que le permiten a uno usarlo para almacenar y recuperar cadenas de caracteres – consulte la documentación para *ctypes*.

`multiprocessing.sharedctypes.Array` (*typecode_or_type*, *size_or_initializer*, ***, *lock=True*)

Lo mismo que `RawArray()`, excepto que, dependiendo del valor de *lock*, se puede retornar un contenedor de sincronización seguro para el proceso en lugar de un arreglo de tipos crudos.

Si *lock* es `True` (el valor predeterminado), se crea un nuevo objeto candado para sincronizar el acceso al valor. Si *lock* es un objeto `Lock` o `RLock`, se utilizará para sincronizar el acceso al valor. Si *lock* es `False`, entonces el acceso al objeto retornado no estará protegido automáticamente por un candado, por lo que no será necesariamente «seguro para el proceso».

Véase que *lock* es un argumento de solo una palabra clave.

`multiprocessing.sharedctypes.Value` (*typecode_or_type*, **args*, *lock=True*)

Lo mismo que `RawValue()` excepto que, dependiendo del valor de *lock*, se puede retornar una envoltura de sincronización segura para el proceso en lugar de un objeto *ctypes* sin procesar.

Si *lock* es `True` (el valor predeterminado), se crea un nuevo objeto candado para sincronizar el acceso al valor. Si *lock* es un objeto `Lock` o `RLock`, se utilizará para sincronizar el acceso al valor. Si *lock* es `False`, entonces el acceso al objeto retornado no estará protegido automáticamente por un candado, por lo que no será necesariamente «seguro para el proceso».

Véase que *lock* es un argumento de solo una palabra clave.

`multiprocessing.sharedctypes.copy` (*obj*)

Retorna un objeto *ctypes* asignado de la memoria compartida, que es una copia del objeto *ctypes obj*.

`multiprocessing.sharedctypes.synchronized` (*obj*, *lock*)

Retorna un objeto contenedor seguro para un objeto *ctypes* que usa *lock* para sincronizar el acceso. Si *lock* es `None` (el valor predeterminado), se crea automáticamente un objeto `multiprocessing.RLock`.

Un contenedor sincronizado tendrá dos métodos además de los del objeto que envuelve: `get_obj()` retorna el objeto envuelto y `get_lock()` retorna el objeto de bloqueo utilizado para la sincronización.

Tenga en cuenta que acceder al objeto *ctypes* a través del contenedor puede ser mucho más lento que acceder al objeto *ctypes* sin formato.

Distinto en la versión 3.5: Los objetos sincronizados admiten el protocolo: *context manager*.

La siguiente tabla compara la sintaxis para crear objetos *ctypes* compartidos desde la memoria compartida con la sintaxis *ctypes* normal. (En la tabla `MyStruct` hay alguna subclase de `ctypes.Structure`.)

<i>ctypes</i>	<i>sharedctypes</i> usando <i>type</i>	<i>sharedctypes</i> usando <i>typecode</i>
<code>c_double(2.4)</code>	<code>RawValue(c_double, 2.4)</code>	<code>RawValue("d", 2.4)</code>
<code>MyStruct(4, 6)</code>	<code>RawValue(MyStruct, 4, 6)</code>	
<code>(c_short * 7)()</code>	<code>RawArray(c_short, 7)</code>	<code>RawArray('h', 7)</code>
<code>(c_int * 3)(9, 2, 8)</code>	<code>RawArray(c_int, (9, 2, 8))</code>	<code>RawArray("i", (9, 2, 8))</code>

A continuación se muestra un ejemplo donde un número de objetos *ctypes* son modificados por un proceso hijo:

```
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__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'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])
```

Los resultados impresos son

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

Administradores (*Managers*)

Los administradores (*managers*) proporcionan una forma de crear datos que se pueden compartir entre diferentes procesos, incluido el intercambio en una red entre procesos que se ejecutan en diferentes máquinas. Un objeto administrador controla un proceso de servidor que gestiona *shared objects* (*objetos compartidos*). Otros procesos pueden acceder a los objetos compartidos mediante el uso de servidores proxy.

`multiprocessing.Manager()`

Retorna un objeto iniciado *SyncManager* que se puede usar para compartir objetos entre procesos. El objeto administrador retornado corresponde a un proceso hijo generado y tiene métodos que crearán objetos compartidos y retornarán los proxies correspondientes.

Los procesos del administrador se cerrarán tan pronto como se recolecte la basura o salga su proceso padre. Las clases de administrador se definen en el módulo `multiprocessing.managers`:

class `multiprocessing.managers.BaseManager` (`[address[, authkey]]`)

Crear un objeto *BaseManager*.

Una vez creado, debe llamar a `start()` o `get_server().serve_forever()` para asegurarse de que el objeto de administrador se refiera a un proceso de administrador iniciado.

address es la dirección en la que el proceso del administrador escucha las nuevas conexiones. Si *address* es `None`, se elige una arbitrariamente.

authkey es la clave de autenticación que se utilizará para verificar la validez de las conexiones entrantes al proceso del servidor. Si *authkey* es `None`, entonces se usa `current_process().authkey`. De lo contrario, se usa *authkey* y debe ser una cadena de *bytes*.

start (`[initializer[, initargs]]`)

Se inicia un subproceso para iniciar el administrador. Si *initializer* no es `None`, entonces el subproceso llamará `initializer(*initargs)` cuando se inicie.

get_server ()

Retorna un objeto `Server` que representa el servidor real bajo el control del Administrador. El objeto `Server` admite el método `serve_forever()`:

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=(' ', 50000), authkey=b'abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

`Server` tiene un atributo adicional *address*.

connect ()

Conecta un objeto de administrador (*manager*) local a un proceso de administrador remoto:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 50000), authkey=b'abc')
>>> m.connect()
```

shutdown ()

Detiene el proceso utilizado por el gerente (*manager*). Esto solo está disponible si `start()` se ha utilizado para iniciar el proceso del servidor.

Esto se puede llamar múltiples veces.

register (`typeid[, callable[, proxytype[, exposed[, method_to_typeid[, create_method]]]]`)

Un método de clase que puede usarse para registrar un tipo o invocarse con la clase de administrador (*manager*).

typeid es un «identificador de tipo» que se utiliza para identificar un tipo particular de objeto compartido. Esto debe ser una cadena de caracteres.

callable es un invocable utilizado para crear objetos para este identificador de tipo. Si una instancia de administrador se conectará al servidor utilizando el método `connect()`, o si el argumento *create_method* es `False`, esto se puede dejar como `None`.

proxytype es una subclase de `BaseProxy` que se usa para crear *proxies* para objetos compartidos con este *typeid*. Si `None`, se crea automáticamente una clase proxy.

Se utiliza *exposed* para especificar una secuencia de nombres de métodos a los que se debe permitir el acceso de los servidores proxy para este tipo de identificación utilizando `BaseProxy._callmethod()`. (Si *exposed* es `None`, entonces `proxytype._exposed` se usa en su lugar si existe). En el caso de que no se especifique una lista expuesta, todos los «métodos públicos» del objeto compartido serán accesibles. (Aquí un «método público» significa cualquier atributo que tenga un método `__call__()` y cuyo nombre no comience con `'_'`.)

El *method_to_typeid* es una asignación utilizada para especificar el tipo de retorno de los métodos expuestos que deberían retornar un proxy. Asigna nombres de métodos a cadenas *typeid*. (Si *method_to_typeid* es `None` entonces `proxytype._method_to_typeid` se usa en su lugar si existe). Si el nombre de un método no es una clave de esta asignación o si la asignación es `None` entonces el objeto retornado por el método se copiará por valor.

create_method determina si un método debe crearse con el nombre *typeid* que se puede usar para indicarle al proceso del servidor que cree un nuevo objeto compartido y retornando un proxy para él. Por defecto es `True`.

BaseManager las instancias también tienen una propiedad de solo lectura:

address

La dirección utilizada por el administrador.

Distinto en la versión 3.3: Los objetos de administrador admiten el protocolo de gestión de contexto; consulte *Tipos Gestores de Contexto*. `__enter__()` inicia el proceso del servidor (si aún no se ha iniciado) y luego retorna el objeto de administrador. `__exit__()` llama *shutdown()*.

En versiones anteriores `__enter__()` no iniciaba el proceso del servidor del administrador si aún no se había iniciado.

class multiprocessing.managers.SyncManager

Una subclase de *BaseManager* que se puede utilizar para la sincronización de procesos. Los objetos de este tipo son retornados por `multiprocessing.Manager()`.

Sus métodos crean y retornan *Objetos Proxy (Proxy Objects)* para varios tipos de datos de uso común que se sincronizarán entre procesos. Esto incluye notablemente listas compartidas y diccionarios.

Barrier (*parties* [, *action* [, *timeout*]])

threading.Barrier crea un objeto compartido y retorna un proxy para él.

Nuevo en la versión 3.3.

BoundedSemaphore ([*value*])

Crea un objeto compartido *threading.BoundedSemaphore* y retorna un proxy para él.

Condition ([*lock*])

Crea un objeto compartido *threading.Condition* y retorna un proxy para él.

Si se proporciona *lock*, debería ser un proxy para un objeto *threading.Lock* o *threading.RLock*.

Distinto en la versión 3.3: El método *wait_for()* fue a añadido.

Event ()

Crea un objeto compartido *threading.Event* y retorna un proxy para él.

Lock ()

Crea un objeto compartido *threading.Lock* y retorna un proxy para él.

Namespace ()

Crea un objeto compartido *Namespace* y retorna un proxy para él.

Queue ([*maxsize*])

Crea un objeto compartido *queue.Queue* y retorna un proxy para él.

RLock ()

Crea un objeto compartido *threading.RLock* y retorna un proxy para él.

Semaphore ([*value*])

Crea un objeto compartido *threading.Semaphore* y retorna un proxy para él.

Array (*typecode*, *sequence*)

Crea un arreglo y retorna un proxy para ello.

Value (*typecode*, *value*)

Crea un objeto con un atributo de escritura *value* y retorna un proxy para él.

dict()

dict(*mapping*)

dict(*sequence*)

Crea un objeto compartido *dict* y retorna un proxy para él.

list()

list(*sequence*)

Crea un objeto compartido *list* y retorna un proxy para él.

Distinto en la versión 3.6: Los objetos compartidos pueden anidarse. Por ejemplo, un objeto contenedor compartido, como una lista compartida, puede contener otros objetos compartidos que serán administrados y sincronizados por *SyncManager*.

class multiprocessing.managers.Namespace

Un tipo que puede registrarse con *SyncManager*.

Un objeto de espacio de nombres no tiene métodos públicos, pero tiene atributos de escritura. Su representación muestra los valores de sus atributos.

Sin embargo, cuando se usa un proxy para un objeto de espacio de nombres, un atributo que comience con '_' será un atributo del proxy y no un atributo del referente:

```
>>> 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')
```

Administradores customizables (*Customized managers*)

Para crear su propio administrador, uno crea una subclase de *BaseManager* y utiliza el método de clase *register()* para registrar nuevos tipos o llamadas con la clase de administrador. Por ejemplo:

```
from multiprocessing.managers import BaseManager

class MathsClass:
    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__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))      # prints 7
        print(maths.mul(7, 8))     # prints 56
```

Utilizando un administrador remoto

Es posible ejecutar un servidor administrador en una máquina y hacer que los clientes lo usen desde otras máquinas (suponiendo que los cortafuegos involucrados lo permitan).

La ejecución de los siguientes comandos crea un servidor para una única cola compartida a la que los clientes remotos pueden acceder:

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Un cliente puede tener accesos al servidor de la siguiente manera:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

Otro cliente puede también usarlo:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

Los procesos locales también pueden acceder a esa cola (*queue*), utilizando el código de arriba en el cliente para acceder de forma remota:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super().__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=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Objetos Proxy (*Proxy Objects*)

Un proxy es un objeto que *se refiere* a un objeto compartido que vive (presumiblemente) en un proceso diferente. Se dice que el objeto compartido es el *referente* del proxy. Varios objetos proxy pueden tener el mismo referente.

Un objeto proxy tiene métodos que invocan los métodos correspondientes de su referente (aunque no todos los métodos del referente estarán necesariamente disponibles a través del proxy). De esta manera, un proxy se puede usar al igual que su referente:

```
>>> 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...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

Tenga en cuenta que la aplicación `str()` a un proxy retornará la representación del referente, mientras que la aplicación `repr()` retornará la representación del proxy.

Una característica importante de los objetos proxy es que son seleccionables para que puedan pasarse entre procesos. Como tal, un referente puede contener *Objetos Proxy (Proxy Objects)*. Esto permite anidar estas listas administradas, dictados y otros *Objetos Proxy (Proxy Objects)*:

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)           # referent of a now contains referent of b
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...> []]
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

Del mismo modo, los proxies `dict` y `list` pueden estar anidados uno dentro del otro:

```
>>> l_outer = manager.list([ manager.dict() for i in range(2) ])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

Si los objetos estándar (no proxy) `list` or `dict` están contenidos en un referente, las modificaciones a esos valores mutables no se propagarán a través del administrador porque el proxy no tiene forma de saber cuándo los valores contenidos dentro son modificados. Sin embargo, almacenar un valor en un proxy de contenedor (que desencadena un `__setitem__` en el objeto proxy) se propaga a través del administrador y, por lo tanto, para modificar efectivamente dicho elemento, uno podría reasignar el valor modificado al proxy de contenedor:

```
# 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

Este enfoque es quizás menos conveniente que emplear anidado *Objetos Proxy (Proxy Objects)* para la mayoría de los casos de uso, pero también demuestra un nivel de control sobre la sincronización.

Nota: Los tipos de proxy en *multiprocessing* no hacen nada para admitir comparaciones por valor. Entonces, por ejemplo, tenemos:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

En su lugar, se debe usar una copia del referente al hacer comparaciones.

class multiprocessing.managers.**BaseProxy**

Los objetos proxy son instancias de subclases de *BaseProxy*.

_callmethod(*methodname*[, *args*[, *kwds*]])

Llama y retorna el resultado de un método del referente del proxy.

Si proxy es un proxy cuyo referente es obj entonces la expresión

```
proxy._callmethod(methodname, args, kwds)
```

evaluará la expresión

```
getattr(obj, methodname)(*args, **kwds)
```

en el proceso del administrador.

El valor retornado será una copia del resultado de la llamada o un proxy a un nuevo objeto compartido; consulte la documentación del argumento *method_to_typeid* de *BaseManager.register()*.

Si la llamada genera una excepción, entonces se vuelve a generar *_callmethod()*. Si se genera alguna otra excepción en el proceso del administrador, esto se convierte en una excepción *RemoteError* y se genera mediante *_callmethod()*.

Tenga en cuenta en particular que se generará una excepción si *methodname* no ha sido *exposed*.

Un ejemplo de uso de *_callmethod()*:

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) # equivalent to l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          # equivalent to l[20]
Traceback (most recent call last):
...
IndexError: list index out of range
```

_getvalue()

Retorna una copia del referente.

Si el referente no se puede deserializar (*unpicklable*), esto generará una excepción.

__repr__()

Retorna una representación de un objeto proxy.

__str__()

Retorna una representación del referente.

Limpieza (*Cleanup*)

Un objeto proxy utiliza una devolución de llamada (*callback*) de referencia débil (*weakref*) para que cuando sea recolectado por el recolector de basura se da de baja del administrador que posee su referente.

Un objeto compartido se elimina del proceso del administrador cuando ya no hay ningún proxy que se refiera a él.

Piscinas de procesos (*Process Pools*)

Se puede crear un grupo de procesos que llevarán a cabo las tareas que se le presenten con la `Pool` class.

```
class multiprocessing.pool.Pool ([processes[, initializer[, initargs[, maxtasksperchild[, context]]]])
```

Un objeto de grupo de procesos que controla un grupo de procesos de trabajo a los que se pueden enviar trabajos. Admite resultados asincrónicos con tiempos de espera y devoluciones de llamada y tiene una implementación de mapa paralelo.

`processes` es el número de procesos de trabajo a utilizar. Si `processes` es `None`, se utiliza el número retornado por `os.cpu_count()`.

Si `initializer` no es `None`, cada proceso de trabajo llamará `initializer(*initargs)` cuando se inicie.

`maxtasksperchild` es el número de tareas que un proceso de trabajo puede completar antes de salir y ser reemplazado por un proceso de trabajo nuevo, para permitir que se liberen los recursos no utilizados. El valor predeterminado `maxtasksperchild` es `None`, lo que significa que los procesos de trabajo vivirán tanto tiempo como el grupo.

`context` se puede utilizar para especificar el contexto utilizado para iniciar los procesos de trabajo. Por lo general, un grupo se crea utilizando la función `multiprocessing.Pool()` o el método de un objeto de contexto `Pool()`. En ambos casos, `context` se establece de manera adecuada.

Tenga en cuenta que los métodos del objeto de grupo solo deben ser invocados por el proceso que creó el grupo.

Advertencia: Los objetos `multiprocessing.pool` tienen recursos internos que necesitan ser administrados adecuadamente (como cualquier otro recurso) utilizando el grupo como administrador de contexto o llamando a `close()` y `terminate()` manualmente. De lo contrario, el proceso puede demorarse en la finalización.

Tenga en cuenta que **no es correcto** confiar en el recolector de basura para destruir el grupo ya que *CPython* no asegura que se llamará al finalizador del grupo (consulte `object.__del__()` para obtener más información).

Nuevo en la versión 3.2: `maxtasksperchild`

Nuevo en la versión 3.4: `context`

Nota: Los procesos de los trabajadores dentro de una `Pool` normalmente viven durante la duración completa de la cola de trabajo de la piscina. Un patrón frecuente que se encuentra en otros sistemas (como *Apache*, *mod_wsgi*, etc.) para liberar recursos en poder de los trabajadores es permitir que un trabajador dentro de un grupo complete solo una cantidad determinada de trabajo antes de salir, limpiarse y generar un nuevo proceso para reemplazar el viejo. El argumento `maxtasksperchild` para `Pool` expone esta capacidad al usuario final.

```
apply (func[, args[, kwds ] ] )
```

Llama a `func` con argumentos `args` y argumentos de palabras clave `kwds`. Se bloquea hasta que el resultado esté listo. Dados estos bloques, `apply_async()` es más adecuado para realizar trabajos en paralelo. Además, `func` solo se ejecuta en uno de los trabajadores de piscina.

```
apply_async (func[, args[, kwds[, callback[, error_callback ] ] ] ] )
```

A variant of the `apply()` method which returns a `AsyncResult` object.

Si se especifica *callback*, debería ser un invocable que acepte un único argumento. Cuando el resultado está listo, se le aplica *callback*, a menos que la llamada falle, en cuyo caso se aplica *error_callback*.

Si se especifica *error_callback*, debería ser un invocable que acepte un único argumento. Si la función de destino falla, se llama a *error_callback* con la instancia de excepción.

Las devoluciones de llamada deben completarse inmediatamente ya que de lo contrario el hilo que maneja los resultados se bloqueará.

map (*func*, *iterable*[, *chunksize*])

Un equivalente paralelo de la función incorporada *map()* (aunque solo admite un argumento *iterable*, para varios iterables consulte *starmap()*). Bloquea hasta que el resultado esté listo.

Este método corta el iterable en varios trozos que envía al grupo de procesos como tareas separadas. El tamaño (aproximado) de estos fragmentos se puede especificar estableciendo *chunksize* en un entero positivo.

Tenga en cuenta que puede causar un alto uso de memoria para iterables muy largos. Considere usar *imap()* o *imap_unordered()* con la opción explícita *chunksize* para una mejor eficiencia.

map_async (*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

A variant of the *map()* method which returns a *AsyncResult* object.

Si se especifica *callback*, debería ser un invocable que acepte un único argumento. Cuando el resultado está listo, se le aplica *callback*, a menos que la llamada falle, en cuyo caso se aplica *error_callback*.

Si se especifica *error_callback*, debería ser un invocable que acepte un único argumento. Si la función de destino falla, se llama a *error_callback* con la instancia de excepción.

Las devoluciones de llamada deben completarse inmediatamente ya que de lo contrario el hilo que maneja los resultados se bloqueará.

imap (*func*, *iterable*[, *chunksize*])

Una versión más perezosa (*lazier*) de *map()*.

El argumento *chunksize* es el mismo que el utilizado por el método *map()*. Para iterables muy largos, usar un valor grande para *chunksize* puede hacer que el trabajo se complete **much** (**mucho**) más rápido que usar el valor predeterminado de 1.

Además, si *chunksize* es 1, el método *next()* del iterador retornado por el método *imap()* tiene un parámetro opcional *timeout*: *next(timeout)* lanzará *multiprocessing.TimeoutError* si el resultado no puede retornarse dentro de *timeout* segundos.

imap_unordered (*func*, *iterable*[, *chunksize*])

Lo mismo que *imap()*, excepto que el orden de los resultados del iterador retornado debe considerarse arbitrario. (Solo cuando hay un solo proceso de trabajo se garantiza que el orden sea «correcto»).

starmap (*func*, *iterable*[, *chunksize*])

Como *map()* excepto que se espera que los elementos de *iterable* sean iterables que se desempaquetan como argumentos.

Por lo tanto, un *iterable* de [(1, 2), (3, 4)] da como resultado [*func*(1, 2), *func*(3, 4)].

Nuevo en la versión 3.3.

starmap_async (*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

Una combinación de *starmap()* y *map_async()* que itera sobre *iterable* de iterables y llama a *func* con los iterables desempaquetados. Como resultado se retorna un objeto.

Nuevo en la versión 3.3.

close()

Impide que se envíen más tareas a la piscina (*pool*). Una vez que se hayan completado todas las tareas, se cerrarán los procesos de trabajo.

terminate()

Detiene los procesos de trabajo inmediatamente sin completar el trabajo pendiente. Cuando el objeto del grupo es basura recolectada *terminate()* se llamará inmediatamente.

join()

Espera a que salgan los procesos de trabajo. Se debe llamar `close()` o `terminate()` antes de usar `join()`.

Nuevo en la versión 3.3: Los objetos de piscina (*pool*) ahora admiten el protocolo de administración de contexto; consulte *Tipos Gestores de Contexto*. `__enter__()` retorna el objeto de grupo, y `__exit__()` llama `terminate()`.

class multiprocessing.pool.AsyncResult

La clase del resultado retornado por `Pool.apply_async()` y `Pool.map_async()`.

get([timeout])

Retorna el resultado cuando llegue. Si `timeout` no es `None` y el resultado no llega dentro de `timeout` segundos, entonces se lanza `multiprocessing.TimeoutError`. Si la llamada remota generó una excepción, esa excepción se volverá a plantear mediante `get()`.

wait([timeout])

Espera hasta que el resultado esté disponible o hasta que pase `timeout` segundos.

ready()

Retorna si la llamada se ha completado.

successful()

Retorna si la llamada se completó sin generar una excepción. Lanzará `ValueError` si el resultado no está listo.

Distinto en la versión 3.7: Si el resultado no está listo `ValueError` aparece en lugar de `AssertionError`.

El siguiente ejemplo demuestra el uso de una piscina(*pool*):

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:          # 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(next(it))                    # prints "0"
        print(next(it))                    # 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
```

Oyentes y Clientes (*Listeners and Clients*)

Por lo general, el paso de mensajes entre procesos se realiza mediante colas o mediante objetos *Connection* retornados por *Pipe()*.

Sin embargo, el módulo *multiprocessing.connection* permite cierta flexibilidad adicional. Básicamente proporciona una API orientada a mensajes de alto nivel para tratar con sockets o canalizaciones con nombre de Windows. También tiene soporte para *digest authentication* usando el módulo *hmac*, y para sondear múltiples conexiones al mismo tiempo.

`multiprocessing.connection.deliver_challenge(connection, authkey)`

Envía un mensaje generado aleatoriamente al otro extremo de la conexión y espera una respuesta.

Si la respuesta coincide con el resumen del mensaje utilizando *authkey* como clave, se envía un mensaje de bienvenida al otro extremo de la conexión. De lo contrario se lanza *AuthenticationError*.

`multiprocessing.connection.answer_challenge(connection, authkey)`

Recibe un mensaje, calcula el resumen del mensaje usando *authkey* como la clave y luego envía el resumen de vuelta.

Si no se recibe un mensaje de bienvenida, se lanza *AuthenticationError*.

`multiprocessing.connection.Client(address[, family[, authkey]])`

Se intenta configurar una conexión con el oyente que utiliza la dirección *address*, retornando *Connection*.

El tipo de conexión está determinado por el argumento *family*, pero esto generalmente se puede omitir ya que generalmente se puede inferir del formato de *address*. (Consulte *Formatos de dirección (Address formats)*)

Si se proporciona *authkey* y no *None*, debe ser una cadena de *bytes* y se utilizará como clave secreta para un desafío de autenticación basado en HMAC. No se realiza la autenticación si *authkey* es *None*. Si falla la autenticación se lanza *AuthenticationError*. Consulte *Llaves de autenticación*.

`class multiprocessing.connection.Listener([address[, family[, backlog[, authkey]]])`

Un contenedor para un *socket* vinculado o una tubería (*pipe*) con nombre de Windows que está “escuchando” las conexiones.

address es la dirección que utilizará el *socket* vinculado o la conocida tubería (*pipe*) con nombre del objeto de escucha.

Nota: Si se usa una dirección de “0.0.0.0”, la dirección no será un punto final conectable en Windows. Si necesita un punto final conectable, debe usar “127.0.0.1”.

family es el tipo de socket (o tubería con nombre) a utilizar. Esta puede ser una de las cadenas de caracteres 'AF_INET' (para un socket TCP), 'AF_UNIX' (para un socket de dominio Unix) o 'AF_PIPE' (para una tubería con nombre de Windows). De estos, solo el primero está garantizado para estar disponible. Si *family* es *None*, *family* se deduce del formato de *address*. Si *address* también es *None*, se elige un valor predeterminado. Este valor predeterminado es *family* con la opción más rápida disponible. Consulte *Formatos de dirección (Address formats)*. Tenga en cuenta que si *family* es 'AF_UNIX' y la dirección es *None*, el *socket* se creará en un directorio temporal privado usando *tempfile.mkstemp()*.

Si el objeto de escucha utiliza un *socket*, entonces *backlog* (1 por defecto) se pasa al método *listen()* del *socket* una vez que se ha vinculado.

Si se proporciona *authkey* y no *None*, debe ser una cadena de *bytes* y se utilizará como clave secreta para un desafío de autenticación basado en HMAC. No se realiza la autenticación si *authkey* es *None*. Si falla la autenticación se lanza *AuthenticationError*. Consulte *Llaves de autenticación*.

`accept()`

Acepta una conexión en el *socket* vinculado o canalización con nombre del objeto de escucha y retorne un objeto *Connection*. Si se intenta la autenticación y falla, entonces se lanza una *AuthenticationError*.

close()

Cierra el socket vinculado o la tubería con nombre del objeto de escucha. Esto se llama automáticamente cuando el oyente es recolectado por el recolector de basura. Sin embargo, es aconsejable llamarlo explícitamente.

Los objetos de escucha tienen las siguientes propiedades de solo lectura:

address

La dirección que está utilizando el objeto *Listener*.

last_accepted

La dirección de donde vino la última conexión aceptada. Si esto no está disponible, entonces es *None*.

Nuevo en la versión 3.3: Los objetos de escucha ahora admiten el protocolo de gestión de contexto – consulte *Tipos Gestores de Contexto*. El objeto *LISTENER* retorna `__enter__()`, y `__exit__()` llama a `close()`.

`multiprocessing.connection.wait(object_list, timeout=None)`

Espera hasta que un objeto en *object_list* esté listo. Retorna la lista de esos objetos en *object_list* que están listos. Si *timeout* es flotante, la llamada se bloquea durante como máximo tantos segundos. Si *timeout* es `None`, se bloqueará por un período ilimitado. Un tiempo de espera negativo es equivalente a un tiempo de espera cero.

Tanto para Unix como para Windows, un objeto puede aparecer en *object_list* si este es

- un objeto legible de *Connection*;
- un objeto conectado y legible de `socket.socket`; o
- el atributo *sentinel* de un objeto *Process*.

Un objeto de conexión o *socket* está listo cuando hay datos disponibles para leer, o el otro extremo se ha cerrado.

Unix: `wait(object_list, timeout)` es casi equivalente a `select.select(object_list, [], [], timeout)`. La diferencia es que si se interrumpe `select.select()` por una señal, este lanza *OSError* con un número de error `EINTR`, a diferencia de `wait()`.

Windows: Un elemento en *object_list* debe ser un identificador de número entero que se pueda esperar (de acuerdo con la definición utilizada por la documentación de la función `Win32 WaitForMultipleObjects()`) o puede ser un objeto con un `fileno()` Método que retorna un manejador de tubo o manejador de tubería. (Tenga en cuenta que las manijas de las tuberías y las manijas de los zócalos son **no** manijas aptas)

Nuevo en la versión 3.3.

Ejemplos

El siguiente código de servidor crea un escucha que utiliza 'secret password' como clave de autenticación. Luego espera una conexión y envía algunos datos al cliente:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)      # family is deduced to be 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

El siguiente código se conecta al servidor y recibe algunos datos del servidor:

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    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])
```

El siguiente código utiliza `wait()` para esperar mensajes de múltiples procesos a la vez:

```
import time, random
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it. This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)
```

Formatos de dirección (*Address formats*)

- Una dirección 'AF_INET' es una tupla de la forma (hostname, port) donde *hostname* es una cadena de caracteres y *port* es un número entero.
- 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.

Llaves de autenticación

Cuando uno usa `Connection.recv()`, los datos recibidos se desbloquean automáticamente. Desafortunadamente, la eliminación de datos de una fuente no confiable es un riesgo de seguridad. Por lo tanto `Listener` y `Client()` usan el módulo `hmac` para proporcionar autenticación de resumen.

Una clave de autenticación es una cadena de bytes que se puede considerar como una contraseña: una vez que se establece una conexión, ambos extremos exigirán pruebas de que el otro conoce la clave de autenticación. (Demostrar que ambos extremos están usando la misma clave **no** implica enviar la clave a través de la conexión).

Si se solicita la autenticación pero no se especifica una clave de autenticación, se utiliza el valor de retorno de `current_process().authkey` (consulte `Process`). Este valor será heredado automáticamente por cualquier objeto `Process` que crea el proceso actual. Esto significa que (por defecto) todos los procesos de un programa multiproceso compartirán una única clave de autenticación que se puede usar al configurar conexiones entre ellos.

Las claves de autenticación adecuadas también se pueden generar utilizando `os.urandom()`.

Logging

Existe cierto soporte para el registro. Sin embargo, tenga en cuenta que el paquete `logging` no utiliza candados compartidos de proceso, por lo que es posible (dependiendo del tipo de controlador) que los mensajes de diferentes procesos se mezclen.

`multiprocessing.get_logger()`

Retorna el registrador utilizado por `multiprocessing`. Si es necesario, se creará uno nuevo.

Cuando se creó por primera vez, el registrador tiene nivel `logging.NOTSET` y no tiene un controlador predeterminado. Los mensajes enviados a este registrador no se propagarán por defecto al registrador raíz.

Tenga en cuenta que en Windows los procesos hijos solo heredarán el nivel del registrador del proceso parental – no se heredará ninguna otra personalización del registrador.

`multiprocessing.log_to_stderr()`

Esta función realiza una llamada a `get_logger()` pero además de retornar el registrador creado por `get_logger`, agrega un controlador que envía la salida a `sys.stderr` usando el formato `'[% (levelname)s/% (processName)s] %(message)s'`.

A continuación se muestra una sesión de ejemplo con el registro activado:

```
>>> 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 /.../pypm-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

Para obtener una tabla completa de niveles de registro, consulte el módulo `logging`.

El módulo `multiprocessing.dummy`

El `multiprocessing.dummy` replica la API de `multiprocessing` pero no es más que un contenedor alrededor del módulo `threading`.

In particular, the `Pool` function provided by `multiprocessing.dummy` returns an instance of `ThreadPool`, which is a subclass of `Pool` that supports all the same method calls but uses a pool of worker threads rather than worker processes.

class `multiprocessing.pool.ThreadPool` (`[processes[, initializer[, initargs]]]`)

A thread pool object which controls a pool of worker threads to which jobs can be submitted. `ThreadPool` instances are fully interface compatible with `Pool` instances, and their resources must also be properly managed, either by using the pool as a context manager or by calling `close()` and `terminate()` manually.

`processes` is the number of worker threads to use. If `processes` is `None` then the number returned by `os.cpu_count()` is used.

Si `initializer` no es `None`, cada proceso de trabajo llamará `initializer(*initargs)` cuando se inicie.

Unlike `Pool`, `maxtasksperchild` and `context` cannot be provided.

Nota: A `ThreadPool` shares the same interface as `Pool`, which is designed around a pool of processes and predates the introduction of the `concurrent.futures` module. As such, it inherits some operations that don't make sense for a pool backed by threads, and it has its own type for representing the status of asynchronous jobs, `AsyncResult`, that is not understood by any other libraries.

Users should generally prefer to use `concurrent.futures.ThreadPoolExecutor`, which has a simpler interface that was designed around threads from the start, and which returns `concurrent.futures.Future` instances that are compatible with many other libraries, including `asyncio`.

17.2.3 Pautas de programación

Hay ciertas pautas y expresiones idiomáticas que deben tenerse en cuenta al usar `multiprocessing`.

Todos los métodos de inicio

Lo siguiente se aplica a todos los métodos de inicio.

Evita estado compartido

En la medida de lo posible, se debe tratar de evitar el desplazamiento de grandes cantidades de datos entre procesos.

Probablemente sea mejor seguir usando colas (*queues*) o tuberías (*pipes*) para la comunicación entre procesos en lugar de usar las primitivas de sincronización de nivel inferior.

Serialización (*picklability*)

Asegúrese que todos los argumentos de los métodos de *proxies* son serializables (*pickable*)

Seguridad de hilos de *proxies*

No usa un objeto proxy de más de un hilo a menos que lo proteja con un candado (*lock*).

(Nunca hay un problema con diferentes procesos que usan el *mismo* proxy.)

Uniéndose a procesos zombies

En Unix, cuando un proceso finaliza pero no se ha unido, se convierte en un zombie. Nunca debería haber muchos porque cada vez que se inicia un nuevo proceso (o se llama `active_children()`) se unirán todos los procesos completados que aún no se hayan unido. También llamando a un proceso

terminado `Process.is_alive` se unirá al proceso. Aun así, probablemente sea una buena práctica unir explícitamente todos los procesos que comience.

Mejor heredar que serializar/deserializar (*pickle/unpickle*)

Cuando se usan los métodos de inicio *spawn* o *forkserver*, muchos tipos de multiprocesamiento deben ser seleccionables para que los procesos secundarios puedan usarlos. Sin embargo, generalmente se debe evitar enviar objetos compartidos a otros procesos mediante tuberías o colas. En su lugar, debe organizar el programa para que un proceso que necesita acceso a un recurso compartido creado en otro lugar pueda heredarlo de un proceso ancestro.

Evita procesos de finalización

El uso del método `Process.terminate` para detener un proceso puede causar que los recursos compartidos (como candados, semáforos, tuberías y colas) que el proceso utiliza actualmente se rompan o no disponible para otros procesos.

Por lo tanto, probablemente sea mejor considerar usar `Process.terminate` en procesos que nunca usan recursos compartidos.

Unirse a procesos que usan colas

Tenga en cuenta que un proceso que ha puesto elementos en una cola esperará antes de finalizar hasta que todos los elementos almacenados en búfer sean alimentados por el hilo «alimentador» a la tubería subyacente. (El proceso secundario puede llamar al método `Queue.cancel_join_thread` de la cola para evitar este comportamiento).

Esto significa que siempre que use una cola debe asegurarse de que todos los elementos que se hayan puesto en la cola se eliminarán antes de unirse al proceso. De lo contrario, no puede estar seguro de que los procesos que han puesto elementos en la cola finalizarán. Recuerde también que los procesos no demoníacos se unirán automáticamente.

Un ejemplo que de bloqueo mutuo (*deadlock*) es el siguiente

```
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()
```

Una solución aquí sería intercambiar las dos últimas líneas (o simplemente eliminar la línea `p.join()`).

Se pasan recursos explícitamente a procesos hijos

En Unix que utiliza el método de inicio *fork*, un proceso secundario puede hacer uso de un recurso compartido creado en un proceso primario utilizando un recurso global. Sin embargo, es mejor pasar el objeto como argumento al constructor para el proceso secundario.

Además de hacer que el código (potencialmente) sea compatible con Windows y los otros métodos de inicio, esto también garantiza que mientras el proceso secundario siga vivo, el objeto no se recolectará en el proceso primario. Esto podría ser importante si se libera algún recurso cuando el objeto es basura recolectada en el proceso padre.

Entonces por ejemplo

```
from multiprocessing import Process, Lock

def f():
```

(continué en la próxima página)

(proviene de la página anterior)

```

... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()

```

debería ser reescrito como

```

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()

```

Tenga cuidado de reemplazar `sys.stdin` con un «file like object»

`multiprocessing` original e incondicionalmente llamado:

```
os.close(sys.stdin.fileno())
```

en el método `multiprocessing.Process._bootstrap()` — Esto dio lugar a problemas con los procesos en proceso. Esto ha sido cambiado a:

```

sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY), closefd=False)

```

Lo que resuelve el problema fundamental de los procesos que chocan entre sí dando como resultado un error de descriptor de archivo incorrecto, pero presenta un peligro potencial para las aplicaciones que reemplazan `sys.stdin()` con un «objeto similar a un archivo» con almacenamiento en búfer de salida. Este peligro es que si varios procesos invocan `close()` en este objeto similar a un archivo, podría ocasionar que los mismos datos se vacíen al objeto varias veces, lo que provocaría corrupción.

Si escribe un objeto similar a un archivo e implementa su propio almacenamiento en caché, puede hacer que sea seguro para la bifurcación (*fork-safe*) almacenando el pid cada vez que se agrega al caché y descartando el caché cuando cambia el pid. Por ejemplo:

```

@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache

```

Para más información, consulte [bpo-5155](#), [bpo-5313](#) y [bpo-5331](#)

Los métodos de inicio *spawn* y *forkserver*

Hay algunas restricciones adicionales que no se aplican al método de inicio *fork*.

Más serialización (*pickability*)

Asegúrese de que todos los argumentos para `Process.__init__()` sean serializables (*pickable*). Además, si la subclase es *Process* asegúrese de que las instancias serán serializables cuando se llame al método *Process.start*.

Variables globales

Tenga en cuenta que si el código que se ejecuta en un proceso secundario intenta acceder a una variable global, entonces el valor que ve (si lo hay) puede no ser el mismo que el valor en el proceso primario en el momento en que fue llamado *Process.start*.

Sin embargo, las variables globales que son solo constantes de nivel de módulo no causan problemas.

Importando de manera segura el módulo principal

Asegúrese de que un nuevo intérprete de Python pueda importar de forma segura el módulo principal sin causar efectos secundarios no deseados (como comenzar un nuevo proceso).

Por ejemplo, usando el método de inicio *spawn* o *forkserver* ejecutando este módulo fallaría produciendo *RuntimeError*:

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

En su lugar, se debe proteger el «punto de entrada» («*entry point*») del programa utilizando como sigue `if __name__ == '__main__':`:

```
from multiprocessing import Process, freeze_support, set_start_method

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
    p = Process(target=foo)
    p.start()
```

(La línea `freeze_support()` puede omitirse si el programa se ejecuta normalmente en lugar de congelarse).

Esto permite que el intérprete de Python recién generado importe de forma segura el módulo y luego ejecute la función del módulo `foo()`.

Se aplican restricciones similares si se crea un grupo o administrador en el módulo principal.

17.2.4 Ejemplos

Demostración de cómo crear y usar gerentes y proxies personalizados:

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    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 range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
    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)

##

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
```

(continué en la próxima página)

(proviene de la página anterior)

```

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, end=' ')
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._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()

```

Usando *Pool*:

```

import multiprocessing
import time
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

```

(continué en la próxima página)

(proviene de la página anterior)

```

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)

    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # Tests
        #

        TASKS = [(mul, (i, 7)) for i in range(10)] + \
                [(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()

        #
        # 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')

```

(continué en la próxima página)

(proviene de la página anterior)

```

try:
    print(pool.map(f, list(range(10))))
except ZeroDivisionError:
    print('\tGot ZeroDivisionError as expected from pool.map()')
else:
    raise AssertionError('expected ZeroDivisionError')

try:
    print(list(pool.imap(f, list(range(10)))))
except ZeroDivisionError:
    print('\tGot ZeroDivisionError as expected from list(pool.imap())')
else:
    raise AssertionError('expected ZeroDivisionError')

it = pool.imap(f, list(range(10)))
for i in range(10):
    try:
        x = next(it)
    except ZeroDivisionError:
        if i == 5:
            pass
    except StopIteration:
        break
    else:
        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:', end=' ')
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:', end=' ')
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()

```

(continué en la próxima página)

(proviene de la página anterior)

```
if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

Un ejemplo que muestra cómo usar las colas para alimentar tareas a una colección de procesos de trabajo y recopilar los resultados:

```
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)

#
# 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
```

(continué en la próxima página)

(proviene de la página anterior)

```

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):
    task_queue.put('STOP')

if __name__ == '__main__':
    freeze_support()
    test()

```

17.3 multiprocessing.shared_memory — Proporciona memoria compartida para acceso directo a través de procesos

Código fuente: `Lib/multiprocessing/shared_memory.py`

Nuevo en la versión 3.8.

Este módulo proporciona una clase, *SharedMemory*, para la asignación y administración de memoria compartida entre uno o más procesos en una máquina con varios núcleos o varios procesadores simétrico (SMP). Para facilitar la gestión del ciclo de vida de la memoria compartida, especialmente entre múltiples procesos, el módulo `multiprocessing.managers` también proporciona la clase *SharedMemoryManager*, una subclase de *BaseManager*.

En este módulo, la memoria compartida se refiere a bloques de memoria compartida de «Sistema estilo V» (aunque no necesariamente se implementa explícitamente como tal) y no se refiere a «memoria compartida distribuida». Este tipo de memoria compartida permite que múltiples procesos lean y escriban en un área común (o compartida) de memoria volátil. Normalmente, los procesos solo tienen acceso a su propio espacio de memoria; la memoria compartida permite compartir datos entre procesos, lo que evita que tengan que enviar estos datos por mensaje. Compartir datos directamente a través de la memoria puede proporcionar importantes beneficios de rendimiento en comparación con compartir datos a través de un disco o socket u otras comunicaciones que requieren la serialización/deserialización y copia de datos.

class `multiprocessing.shared_memory.SharedMemory` (*name=None, create=False, size=0*)

Crea un nuevo bloque de memoria compartida o guarda un bloque ya existente. Se debe dar un nombre único a cada bloque de memoria compartida; por lo tanto, un proceso puede crear un nuevo bloque de memoria compartida con un nombre particular y un proceso diferente se puede conectar a ese mismo bloque de memoria compartida usando ese mismo nombre.

Como un recurso para compartir datos entre procesos, los bloques de memoria compartida pueden sobrevivir al proceso original que los creó. Cuando un proceso ya no necesita acceso a un bloque de memoria compartida que otros procesos aún podrían necesitar, se debe llamar al método `close()`. Cuando un proceso ya no

necesita un bloque de memoria compartida, se debe llamar al método `unlink()` para garantizar una limpieza adecuada.

`name` es el nombre único para la memoria compartida solicitada, especificada como una cadena de caracteres. Al crear un nuevo bloque de memoria compartida, si se proporciona `None` (valor por defecto) para el nombre, se generará un nombre nuevo.

`create` controla si se crea un nuevo bloque de memoria compartida (`True`) o si se adjunta un bloque de memoria compartida existente (`False`).

`size` especifica el número solicitado de bytes al crear un nuevo bloque de memoria compartida. Debido a que algunas plataformas eligen asignar fragmentos de memoria en función del tamaño de página de memoria de esa plataforma, el tamaño exacto del bloque de memoria compartida puede ser mayor o igual al tamaño solicitado. Cuando se conecta a un bloque de memoria compartida existente, se ignora el parámetro `size`.

close()

Cierra el acceso a la memoria compartida desde esta instancia. Para garantizar la limpieza adecuada de los recursos, todas las instancias deben llamar a `close()` una vez que la instancia ya no sea necesaria. Tenga en cuenta que llamar a `close()` no causa que el bloque de memoria compartida se destruya.

unlink()

Solicita que se destruya el bloque de memoria compartida subyacente. Para garantizar la limpieza adecuada de los recursos, se debe llamar a `unlink()` una vez (y solo una vez) en todos los procesos que necesitan el bloque de memoria compartida. Después de solicitar su destrucción, un bloque de memoria compartida puede o no destruirse de inmediato y este comportamiento puede diferir entre plataformas. Los intentos de acceder a los datos dentro del bloque de memoria compartida después de que se haya llamado a `unlink()` pueden provocar errores de acceso a la memoria. Nota: el último proceso para liberar el bloque de memoria compartida puede llamar a `unlink()` y `close()` en cualquier orden.

buf

Un *memoryview* del contenido del bloque de memoria compartida.

name

Acceso de solo lectura al nombre único del bloque de memoria compartida.

size

Acceso de solo lectura al tamaño en bytes del bloque de memoria compartida.

El siguiente ejemplo muestra el uso de bajo nivel de instancias de *SharedMemory*:

```
>>> from multiprocessing import shared_memory
>>> shm_a = shared_memory.SharedMemory(create=True, size=10)
>>> type(shm_a.buf)
<class 'memoryview'>
>>> buffer = shm_a.buf
>>> len(buffer)
10
>>> buffer[:4] = bytearray([22, 33, 44, 55]) # Modify multiple at once
>>> buffer[4] = 100 # Modify single byte at a time
>>> # Attach to an existing shared memory block
>>> shm_b = shared_memory.SharedMemory(shm_a.name)
>>> import array
>>> array.array('b', shm_b.buf[:5]) # Copy the data into a new array.array
array('b', [22, 33, 44, 55, 100])
>>> shm_b.buf[:5] = b'howdy' # Modify via shm_b using bytes
>>> bytes(shm_a.buf[:5]) # Access via shm_a
b'howdy'
>>> shm_b.close() # Close each SharedMemory instance
>>> shm_a.close()
>>> shm_a.unlink() # Call unlink only once to release the shared memory
```

El siguiente ejemplo muestra un uso práctico de la clase *SharedMemory* con NumPy arrays, accediendo al mismo `numpy.ndarray` desde dos shells de Python distintos:


```

>>> # In the first Python interactive shell
>>> import numpy as np
>>> a = np.array([1, 1, 2, 3, 5, 8]) # Start with an existing NumPy array
>>> from multiprocessing import shared_memory
>>> shm = shared_memory.SharedMemory(create=True, size=a.nbytes)
>>> # Now create a NumPy array backed by shared memory
>>> b = np.ndarray(a.shape, dtype=a.dtype, buffer=shm.buf)
>>> b[:] = a[:] # Copy the original data into shared memory
>>> b
array([1, 1, 2, 3, 5, 8])
>>> type(b)
<class 'numpy.ndarray'>
>>> type(a)
<class 'numpy.ndarray'>
>>> shm.name # We did not specify a name so one was chosen for us
'psm_21467_46075'

>>> # In either the same shell or a new Python shell on the same machine
>>> import numpy as np
>>> from multiprocessing import shared_memory
>>> # Attach to the existing shared memory block
>>> existing_shm = shared_memory.SharedMemory(name='psm_21467_46075')
>>> # Note that a.shape is (6,) and a.dtype is np.int64 in this example
>>> c = np.ndarray((6,), dtype=np.int64, buffer=existing_shm.buf)
>>> c
array([1, 1, 2, 3, 5, 8])
>>> c[-1] = 888
>>> c
array([ 1,  1,  2,  3,  5, 888])

>>> # Back in the first Python interactive shell, b reflects this change
>>> b
array([ 1,  1,  2,  3,  5, 888])

>>> # Clean up from within the second Python shell
>>> del c # Unnecessary; merely emphasizing the array is no longer used
>>> existing_shm.close()

>>> # Clean up from within the first Python shell
>>> del b # Unnecessary; merely emphasizing the array is no longer used
>>> shm.close()
>>> shm.unlink() # Free and release the shared memory block at the very end

```

class multiprocessing.managers.**SharedMemoryManager** ([*address*[, *authkey*]])

Una subclase de *BaseManager* que se puede utilizar para la gestión de bloques de memoria compartida en todos los procesos.

Una llamada al método *start()* en una instancia de *SharedMemoryManager* hace que se inicie un nuevo proceso. El único propósito de este nuevo proceso es administrar el ciclo de vida de todos los bloques de memoria compartida creados a través de él. Para activar la liberación de todos los bloques de memoria compartida administrados por ese proceso, llama al método *shutdown()* en la instancia. Esto desencadena una llamada al método *SharedMemory.unlink()* en todos los objetos de la clase *SharedMemory* administrados por ese proceso y luego detiene el proceso en sí. Al crear instancias de *SharedMemory* a través de un *SharedMemoryManager*, evitamos la necesidad de rastrear manualmente y activar la liberación de recursos de memoria compartida.

Esta clase proporciona métodos para crear y retornar instancias *SharedMemory* y para crear un objeto de tipo lista (*ShareableList*) basados en memoria compartida.

Consulte *multiprocessing.managers.BaseManager* para obtener una descripción de los argumentos heredados opcionales *address* y *authkey* y cómo se deben usar para registrar un servicio *SharedMemoryManager* desde otro proceso.

SharedMemory (*size*)

Crea y retorna un nuevo objeto *SharedMemory* con el tamaño *size* especificado en bytes.

ShareableList (*sequence*)

Crea y retorna un nuevo objeto *ShareableList*, inicializado por los valores de la entrada *sequence*.

El siguiente ejemplo muestra los mecanismos básicos de *SharedMemoryManager*:

```
>>> from multiprocessing.managers import SharedMemoryManager
>>> smm = SharedMemoryManager()
>>> smm.start() # Start the process that manages the shared memory blocks
>>> sl = smm.ShareableList(range(4))
>>> sl
ShareableList([0, 1, 2, 3], name='psm_6572_7512')
>>> raw_shm = smm.SharedMemory(size=128)
>>> another_sl = smm.ShareableList('alpha')
>>> another_sl
ShareableList(['a', 'l', 'p', 'h', 'a'], name='psm_6572_12221')
>>> smm.shutdown() # Calls unlink() on sl, raw_shm, and another_sl
```

El siguiente ejemplo muestra un patrón más conveniente para usar un objeto *SharedMemoryManager* con la sentencia *with* para asegurarse de que todos los bloques de memoria se liberen cuando ya no son necesarios. Esto suele ser más práctico que el ejemplo anterior:

```
>>> with SharedMemoryManager() as smm:
...     sl = smm.ShareableList(range(2000))
...     # Divide the work among two processes, storing partial results in sl
...     p1 = Process(target=do_work, args=(sl, 0, 1000))
...     p2 = Process(target=do_work, args=(sl, 1000, 2000))
...     p1.start()
...     p2.start() # A multiprocessing.Pool might be more efficient
...     p1.join()
...     p2.join() # Wait for all work to complete in both processes
...     total_result = sum(sl) # Consolidate the partial results now in sl
```

Cuando se utiliza un *SharedMemoryManager* en una sentencia *with*, los bloques de memoria compartida creados por ese administrador se liberan cuando la sentencia dentro del bloque de código *with* finaliza la ejecución.

class multiprocessing.shared_memory.*ShareableList* (*sequence=None*, *, *name=None*)

Construye un objeto mutable compatible con el tipo de lista cuyos valores se almacenan en un bloque de memoria compartida. Esto reduce los valores de tipo que se pueden almacenar solo a tipos de datos nativos *int*, *float*, *bool*, *str* (menos de 10 MB cada uno), *bytes* (menos de 10 MB cada uno) y *None*. Otra diferencia importante con una lista nativa es que es imposible cambiar el tamaño (es decir, sin adición al final de la lista, sin inserción, etc.) y que no es posible crear nuevas instancias de *ShareableList* mediante la división.

sequence se utiliza para completar una nueva *ShareableList* con valores. Establezca en *None* para registrar en su lugar una *ShareableList* ya existente por su nombre único de memoria compartida.

name es el nombre único para la memoria compartida solicitada, como se describe en la definición de *SharedMemory*. Al adjuntar a una *ShareableList* existente, especifique el nombre único de su bloque de memoria compartida mientras deja *sequence* establecida en *None*.

count (*value*)

Retorna el número de ocurrencias de *value*.

index (*value*)

Retorna la primera posición del índice de *value*. Lanza *ValueError* si *value* no está presente.

format

Atributo de solo lectura que contiene el formato de empaquetado *struct* utilizado por todos los valores almacenados actualmente.

shm

La instancia de *SharedMemory* donde se almacenan los valores.

El siguiente ejemplo muestra el uso básico de una instancia *ShareableList*:

```
>>> from multiprocessing import shared_memory
>>> a = shared_memory.ShareableList(['howdy', b'HoWdY', -273.154, 100, None, True, ↵
↵42])
>>> [ type(entry) for entry in a ]
[<class 'str'>, <class 'bytes'>, <class 'float'>, <class 'int'>, <class 'NoneType'>
↵, <class 'bool'>, <class 'int'>]
>>> a[2]
-273.154
>>> a[2] = -78.5
>>> a[2]
-78.5
>>> a[2] = 'dry ice' # Changing data types is supported as well
>>> a[2]
'dry ice'
>>> a[2] = 'larger than previously allocated storage space'
Traceback (most recent call last):
...
ValueError: exceeds available storage for existing str
>>> a[2]
'dry ice'
>>> len(a)
7
>>> a.index(42)
6
>>> a.count(b'howdy')
0
>>> a.count(b'HoWdY')
1
>>> a.shm.close()
>>> a.shm.unlink()
>>> del a # Use of a ShareableList after call to unlink() is unsupported
```

El siguiente ejemplo muestra cómo uno, dos o muchos procesos pueden acceder al mismo *ShareableList* al proporcionar el nombre del bloque de memoria compartida detrás de él:

```
>>> b = shared_memory.ShareableList(range(5)) # In a first process
>>> c = shared_memory.ShareableList(name=b.shm.name) # In a second process
>>> c
ShareableList([0, 1, 2, 3, 4], name='...')
>>> c[-1] = -999
>>> b[-1]
-999
>>> b.shm.close()
>>> c.shm.close()
>>> c.shm.unlink()
```

17.4 El paquete `concurrent`

Actualmente solo existe un módulo en este paquete:

- `concurrent.futures` – Lanzamiento de tareas paralelas

17.5 `concurrent.futures` — Lanzamiento de tareas paralelas

Nuevo en la versión 3.2.

Código fuente: `Lib/concurrent/futures/thread.py` y `Lib/concurrent/futures/process.py`

El módulo `concurrent.futures` provee una interfaz de alto nivel para ejecutar invocables de forma asincrónica.

La ejecución asincrónica se puede realizar mediante hilos, usando `ThreadPoolExecutor`, o procesos independientes, mediante `ProcessPoolExecutor`. Ambos implementan la misma interfaz, que se encuentra definida por la clase abstracta `Executor`.

17.5.1 Objetos Ejecutores

class `concurrent.futures.Executor`

Una clase abstracta que proporciona métodos para ejecutar llamadas de forma asincrónica. No debe ser utilizada directamente, sino a través de sus subclasses.

submit (*fn*, **args*, ***kwargs*)

Programa la invocación de *fn*, que será ejecutada como `fn(*args **kwargs)` y retorna un objeto `Future` que representa la ejecución del invocable.

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map (*func*, **iterables*, *timeout=None*, *chunksize=1*)

Similar a `map(func, *iterables)` excepto que:

- los *iterables* son recolectados inmediatamente, en lugar de perezosamente;
- *func* se ejecuta de forma asincrónica y se pueden realizar varias llamadas a *func* simultáneamente.

El iterador retornado lanza `concurrent.futures.TimeoutError` si `__next__()` es llamado y el resultado no está disponible luego de *timeout* segundos luego de la llamada original a `Executor.map()`. *timeout* puede ser un int o un float. Si no se provee un *timeout* o es `None`, no hay limite de espera.

Si una llamada a *func* lanza una excepción, dicha excepción va a ser lanzada cuando su valor sea retornado por el iterador.

When using `ProcessPoolExecutor`, this method chops *iterables* into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer. For very long iterables, using a large value for *chunksize* can significantly improve performance compared to the default size of 1. With `ThreadPoolExecutor`, *chunksize* has no effect.

Distinto en la versión 3.5: Se agregó el argumento *chunksize*.

shutdown (*wait=True*)

Indica al ejecutor que debe liberar todos los recursos que está utilizando cuando los futuros actualmente pendientes de ejecución finalicen. Las llamadas a `Executor.submit()` y `Executor.map()` realizadas después del apagado lanzarán `RuntimeError`.

Si `wait` es `True` este método no retornará hasta que todos los futuros pendientes hayan terminado su ejecución y los recursos asociados al ejecutor hayan sido liberados. Si `wait` es `False`, este método retornará de inmediato y los recursos asociados al ejecutor se liberarán cuando todos los futuros asociados hayan finalizado su ejecución. Independientemente del valor de `wait`, el programa Python entero no finalizará hasta que todos los futuros pendientes hayan finalizado su ejecución.

You can avoid having to call this method explicitly if you use the `with` statement, which will shutdown the `Executor` (waiting as if `Executor.shutdown()` were called with `wait` set to `True`):

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

17.5.2 ThreadPoolExecutor

`ThreadPoolExecutor` es una subclase de `Executor` que usa un grupo de hilos para ejecutar llamadas de forma asincrónica.

Pueden ocurrir bloqueos mutuos cuando la llamada asociada a un `Future` espera el resultado de otro `Future`. Por ejemplo:

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result()) # b will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result()) # a will never complete because it is waiting on b.
    return 6

executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

Y:

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class `concurrent.futures.ThreadPoolExecutor` (*max_workers=None*, *th-read_name_prefix="*, *initializer=None*, *initargs=()*)

Subclase de `Executor` que utiliza un grupo de hilos de `max_workers` como máximo para ejecutar llamadas de forma asincrónica.

`initializer` es un invocable opcional que es llamado al comienzo de cada hilo de trabajo; `initargs` es una tupla de argumentos pasados al inicializador. Si el `initializer` lanza una excepción, todos los trabajos actualmente pendientes lanzarán `BrokenThreadPool`, así como cualquier intento de enviar más trabajos al grupo.

Distinto en la versión 3.5: Si *max_workers* es *None* o no es especificado, se tomará por defecto el número de procesadores de la máquina, multiplicado por 5, asumiendo que *ThreadPoolExecutor* a menudo se utiliza para paralelizar E/S en lugar de trabajo de CPU y que el número de trabajadores debe ser mayor que el número de trabajadores para *ProcessPoolExecutor*.

Nuevo en la versión 3.6: El argumento *thread_name_prefix* fue añadido para permitir al usuario controlar los nombres asignados a los *threading.Thread* creados por el grupo para facilitar la depuración del programa.

Distinto en la versión 3.7: Se agregaron los argumentos *initializer* y *initargs*.

Distinto en la versión 3.8: El valor predeterminado de *max_workers* fue reemplazado por `min(32, os.cpu_count() + 4)`. Este valor predeterminado conserva al menos 5 trabajadores para las tareas vinculadas de E/S. Utiliza como máximo 32 núcleos de CPU para tareas vinculadas a la CPU que liberan el GIL. Y evita utilizar recursos muy grandes implícitamente en máquinas de muchos núcleos.

ThreadPoolExecutor ahora también reutiliza hilos inactivos antes de crear *max_workers* hilos de trabajo.

Ejemplo de ThreadPoolExecutor

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://nonexistant-subdomain.python.org/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

17.5.3 ProcessPoolExecutor

The *ProcessPoolExecutor* class is an *Executor* subclass that uses a pool of processes to execute calls asynchronously. *ProcessPoolExecutor* uses the *multiprocessing* module, which allows it to side-step the *Global Interpreter Lock* but also means that only picklable objects can be executed and returned.

El módulo `__main__` debe ser importable por los subprocesos trabajadores. Esto significa que *ProcessPoolExecutor* no funcionará en el intérprete interactivo.

Llamar a métodos de *Executor* o *Future* desde el invocable enviado a *ProcessPoolExecutor* resultará en bloqueos mutuos.

```
class concurrent.futures.ProcessPoolExecutor(max_workers=None, mp_context=None,
                                             initializer=None, initargs=())
```

An *Executor* subclass that executes calls asynchronously using a pool of at most *max_workers* processes. If

max_workers is `None` or not given, it will default to the number of processors on the machine. If *max_workers* is less than or equal to 0, then a `ValueError` will be raised. On Windows, *max_workers* must be less than or equal to 61. If it is not then `ValueError` will be raised. If *max_workers* is `None`, then the default chosen will be at most 61, even if more processors are available. *mp_context* can be a multiprocessing context or `None`. It will be used to launch the workers. If *mp_context* is `None` or not given, the default multiprocessing context is used.

initializer is an optional callable that is called at the start of each worker process; *initargs* is a tuple of arguments passed to the initializer. Should *initializer* raise an exception, all currently pending jobs will raise a `BrokenProcessPool`, as well as any attempt to submit more jobs to the pool.

Distinto en la versión 3.3: Cuando uno de los procesos finaliza abruptamente, se lanzará `BrokenProcessPool`. Anteriormente, el comportamiento no estaba definido, pero las operaciones en el ejecutor o sus futuros a menudo se detenían o bloqueaban mutuamente.

Distinto en la versión 3.7: El argumento *mp_context* se agregó para permitir a los usuarios controlar el método de iniciación para procesos de trabajo creados en el grupo.

Se agregaron los argumentos *initializer* y *initargs*.

Ejemplo de `ProcessPoolExecutor`

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

17.5.4 Objetos Futuro

La clase `Future` encapsula la ejecución asincrónica del invocable. Las instancias de `Future` son creadas por `Executor.submit()`.

class `concurrent.futures.Future`

Encapsula la ejecución asincrónica del invocable. Las instancias de `Future` son creadas por `Executor.submit()` y no deberían ser creadas directamente, excepto para pruebas.

cancel()

Intenta cancelar la llamada. Si el invocable está siendo ejecutado o ha finalizado su ejecución y no puede ser cancelado el método retornará `False`, de lo contrario la llamada será cancelada y el método retornará `True`.

cancelled()

Retorna `True` si la llamada fue cancelada exitosamente.

running()

Retorna `True` si la llamada está siendo ejecutada y no puede ser cancelada.

done()

Retorna `True` si la llamada fue cancelada exitosamente o terminó su ejecución.

result(timeout=None)

Retorna el valor retornado por la llamada. Si la llamada aún no ha finalizado, el método esperará un total de `timeout` segundos. Si la llamada no ha finalizado luego de `timeout` segundos, `concurrent.futures.TimeoutError` será lanzada. `timeout` puede ser un `int` o un `float`. Si `timeout` es `None` o no fue especificado, no hay limite de espera.

Si el futuro es cancelado antes de finalizar su ejecución, `CancelledError` será lanzada.

Si la llamada lanzó una excepción, este método lanzará la misma excepción.

exception(timeout=None)

Retorna la excepción lanzada por la llamada. Si la llamada aún no ha finalizado, el método esperará un máximo de `timeout` segundos. Si la llamada aún no ha finalizado luego de `timeout` segundos, entonces `concurrent.futures.TimeoutError` será lanzada. `timeout` puede ser un `int` o un `float`. Si `timeout` es `None` o no es especificado, no hay limite en el tiempo de espera.

Si el futuro es cancelado antes de finalizar su ejecución, `CancelledError` será lanzada.

Si la llamada es completada sin excepciones, se retornará `None`.

add_done_callback(fn)

Asocia el invocable `fn` al futuro. `fn` va a ser llamada, con el futuro como su único argumento, cuando el futuro sea cancelado o finalice su ejecución.

Added callables are called in the order that they were added and are always called in a thread belonging to the process that added them. If the callable raises an `Exception` subclass, it will be logged and ignored. If the callable raises a `BaseException` subclass, the behavior is undefined.

Si el futuro ya ha finalizado su ejecución o fue cancelado, `fn` retornará inmediatamente.

Los siguientes métodos de `Future` están pensados para ser usados en pruebas unitarias e implementaciones de `Executor`.

set_running_or_notify_cancel()

Este método sólo debe ser llamado en implementaciones de `Executor` antes de ejecutar el trabajo asociado al `Future` y por las pruebas unitarias.

Si el método retorna `False` entonces `Future` fue cancelado. i.e. `Future.cancel()` fue llamado y retornó `True`. Todos los hilos esperando la finalización del `Future` (i.e. a través de `as_completed()` o `wait()`) serán despertados.

Si el método retorna `True`, entonces el `Future` no fue cancelado y ha sido colocado en estado de ejecución, i.e. las llamadas a `Future.running()` retornarán `True`.

Este método solo puede ser llamado una sola vez y no puede ser llamado luego de haber llamado a `Future.set_result()` o a `Future.set_exception()`.

set_result (*result*)

Establece *result* como el resultado del trabajo asociado al `Future`.

Este método solo debe ser usado por implementaciones de `Executor` y pruebas unitarias.

Distinto en la versión 3.8: Este método lanza `concurrent.futures.InvalidStateError` si `Future` ya ha finalizado su ejecución.

set_exception (*exception*)

Establece *exception*, subclase de `Exception`, como el resultado del trabajo asociado al `Future`.

Este método solo debe ser usado por implementaciones de `Executor` y pruebas unitarias.

Distinto en la versión 3.8: Este método lanza `concurrent.futures.InvalidStateError` si `Future` ya ha finalizado su ejecución.

17.5.5 Funciones del Módulo

`concurrent.futures.wait` (*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)

Espera a la finalización de las instancias de `Future` (posiblemente creadas por distintas instancias de `Executor`) dadas por *fs*. Retorna una tupla nombrada de 2 conjuntos. El primer conjunto, llamado `done`, contiene los futuros que finalizaron su ejecución (producto de su finalización normal o su cancelación) antes del tiempo de espera especificado. El segundo conjunto, llamado `not_done`, contiene los futuros que no finalizaron su ejecución (pueden estar pendientes o ejecutándose en ese momento).

El argumento *timeout* puede ser usado para controlar la espera máxima en segundos antes de retornar. *timeout* puede ser un `int` o un `float`. Si *timeout* no es especificado o es `None`, no hay limite en el tiempo de espera.

return_when indica cuando debe retornar esta función. Debe ser alguna de las siguientes constantes:

Constante	Descripción
<code>FIRST_COMPLETED</code>	La función retornará cuando cualquier futuro finalice o sea cancelado.
<code>FIRST_EXCEPTION</code>	La función retornará cuando cualquier futuro finalice lanzando una excepción. Si ningún futuro lanza una excepción, esta opción es equivalente a <code>ALL_COMPLETED</code> .
<code>ALL_COMPLETED</code>	La función retornará cuando todos los futuros finalicen o sean cancelados.

`concurrent.futures.as_completed` (*fs*, *timeout=None*)

Returns an iterator over the `Future` instances (possibly created by different `Executor` instances) given by *fs* that yields futures as they complete (finished or cancelled futures). Any futures given by *fs* that are duplicated will be returned once. Any futures that completed before `as_completed()` is called will be yielded first. The returned iterator raises a `concurrent.futures.TimeoutError` if `__next__()` is called and the result isn't available after *timeout* seconds from the original call to `as_completed()`. *timeout* can be an `int` or `float`. If *timeout* is not specified or `None`, there is no limit to the wait time.

Ver también:

PEP 3148 – futuros - ejecutar cálculos asincrónicamente La propuesta que describe esta propuesta de inclusión en la biblioteca estándar de Python.

17.5.6 Clases de Excepciones

exception `concurrent.futures.CancelledError`

Lanzada cuando un futuro es cancelado.

exception `concurrent.futures.TimeoutError`

Lanzada cuando un futuro excede el tiempo de espera máximo.

exception `concurrent.futures.BrokenExecutor`

Derivada de `RuntimeError`, esta excepción es lanzada cuando un ejecutor se encuentra corrupto por algún motivo y no puede ser utilizado para enviar o ejecutar nuevas tareas.

Nuevo en la versión 3.7.

exception `concurrent.futures.InvalidStateError`

Lanzada cuando una operación es realizada sobre un futuro que no permite dicha operación en el estado actual.

Nuevo en la versión 3.8.

exception `concurrent.futures.thread.BrokenThreadPool`

Derivada de `BrokenExecutor`, esta excepción es lanzada cuando uno de los trabajadores de `ThreadPoolExecutor` ha fallado en su inicialización.

Nuevo en la versión 3.7.

exception `concurrent.futures.process.BrokenProcessPool`

Derivada de `BrokenExecutor` (previamente `RuntimeError`), esta excepción es lanzada cuando uno de los trabajadores de `ProcessPoolExecutor` ha finalizado de forma abrupta (por ejemplo, al ser terminado desde afuera del proceso).

Nuevo en la versión 3.3.

17.6 subprocess — Gestión de subprocessos

Código fuente: [Lib/subprocess.py](#)

El módulo `subprocess` permite lanzar nuevos procesos, conectarse a sus pipes de entrada/salida/error y obtener sus códigos de resultado. Este módulo está destinado a reemplazar múltiples módulos y funciones previamente existentes:

```
os.system
os.spawn*
```

Se puede obtener información sobre cómo utilizar el módulo `subprocess` para reemplazar estos módulos y funciones en las siguientes secciones.

Ver también:

PEP 324 – PEP de proposición del módulo `subprocess`

17.6.1 Uso del módulo `subprocess`

La opción recomendada para invocar subprocessos es utilizar la función `run()` para todos los casos al alcance de ésta. Para usos más avanzados, se puede utilizar la interfaz de más bajo nivel `Popen`.

La función `run()` se añadió en Python 3.5; si necesita mantener la compatibilidad con versiones anteriores, consulte la sección [Antigua API de alto nivel](#).

```
subprocess.run(args, *, stdin=None, input=None, stdout=None, stderr=None, capture_output=False,
               shell=False, cwd=None, timeout=None, check=False, encoding=None, errors=None,
               text=None, env=None, universal_newlines=None, **other_popen_kwargs)
```

Ejecuta la orden descrita por `args`. Espera a que termine y retorna una instancia de `CompletedProcess`.

Los argumentos mostrados en la definición superior son sólo los más comunes; que se describen posteriormente en *Argumentos frecuentemente empleados* (de ahí el uso de la notación por clave en la signatura abreviada). La signatura completa de la función es a grandes rasgos la misma que la del constructor de `Popen`; la mayoría de los argumentos de esta función se pasan a esa interfaz (no es el caso de `timeout`, `input`, `check` ni `capture_output`).

Si `capture_output` es verdadero, se capturarán `stdout` y `stderr`. En tal caso, el objeto `Popen` interno se crea automáticamente con `stdout=PIPE` y `stderr=PIPE`. No se pueden proporcionar los argumentos `stdout` y `stderr` a la vez que `capture_output`. Si se desea capturar y combinar los dos flujos, se ha de usar `stdout=PIPE` y `stderr=STDOUT` en lugar de `capture_output`.

El argumento `timeout` se pasa a `Popen.communicate()`. Si vence el plazo de ejecución, se matará el proceso hijo y se le esperará. Se relanzará la excepción `TimeoutExpired` cuando finalice el proceso hijo.

Se pasará el argumento `input` a `Popen.communicate()` y de ahí, a la entrada estándar del subprocesso. Si se usa, debe ser una secuencia de bytes o una cadena de texto si se especifican `encoding` o `errors` o `text` en verdadero. Cuando se usa, el objeto `Popen` interno se crea automáticamente con `stdin=PIPE` y no se puede usar el argumento `stdin` a la vez.

Si `check` es verdadero y el proceso retorna un resultado distinto de cero, se lanzará una excepción `CalledProcessError`. Los atributos de dicha excepción contendrán los argumentos, el código retornado y tanto `stdout` como `stderr` si se capturaron.

Si se especifican `encoding` o `errors` o `text` es verdadero, se abrirán los objetos fichero para `stdin`, `stdout` y `stderr` en modo texto, con los `encoding` y `errors` especificados o los valores predeterminados de `io.TextIOWrapper`. El argumento `universal_newlines` equivale a `text` y se admite por compatibilidad hacia atrás. Los objetos fichero se abren en modo binario por defecto.

Si `env` no es `None`, debe ser un mapeo que defina las variables de entorno para el nuevo proceso; se utilizarán éstas en lugar del comportamiento predeterminado de heredar el entorno del proceso actual. Se le pasa directamente a `Popen`.

Ejemplos:

```
>>> subprocess.run(["ls", "-l"]) # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n', stderr=b'')
```

Nuevo en la versión 3.5.

Distinto en la versión 3.6: Se añadieron los parámetros `encoding` y `errors`

Distinto en la versión 3.7: Se añadió el parámetro `text` como alias más comprensible de `universal_newlines`. Se añadió el parámetro `capture_output`.

Distinto en la versión 3.8.17: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

class subprocess.CompletedProcess

El valor retornado por `run()`, que representa un proceso ya terminado.

args

Los argumentos utilizados para lanzar el proceso. Pueden ser una lista o una cadena.

returncode

Estado de salida del proceso hijo. Típicamente, un estado de salida 0 indica que la ejecución tuvo éxito.

Un valor negativo `-N` indica que el hijo fue forzado a terminar con la señal `N` (solamente POSIX).

stdout

La salida estándar capturada del proceso hijo. Una secuencia de bytes, o una cadena si se llamó a `run()` con `encoding`, `errors`, o `text` establecidos a `True`. `None` si no se capturó el error estándar.

Si se ejecutó el proceso con `stderr=subprocess.STDOUT`, `stdout` y `stderr` se combinarán en este atributo, y `stderr` será `None`.

stderr

El error estándar capturado del proceso hijo. Una secuencia de bytes, o una cadena si se llamó a `run()` con `encoding`, `errors`, o `text` establecidos a `True`. `None` si no se capturó el error estándar.

check_returncode()

Si `returncode` no es cero, lanza un `CalledProcessError`.

Nuevo en la versión 3.5.

subprocess.DEVNULL

Valor especial que se puede usar como argumento `stdin`, `stdout` o `stderr` de `Popen` y que indica que se usará el fichero especial `os.devnull`.

Nuevo en la versión 3.3.

subprocess.PIPE

Valor especial que se puede usar como argumento `stdin`, `stdout` o `stderr` de `Popen` y que indica que se abrirá un pipe al flujo indicado. Es útil para usarlo con `Popen.communicate()`.

subprocess.STDOUT

Valor especial que se puede usar de argumento `stderr` a `Popen` y que indica que el error estándar debería ir al mismo gestor que la salida estándar.

exception subprocess.SubprocessError

Clase base para el resto de excepciones de este módulo.

Nuevo en la versión 3.3.

exception subprocess.TimeoutExpired

Subclase de `SubprocessError`, se lanza cuando expira un plazo de ejecución esperando a un proceso hijo.

cmd

Orden que se utilizó para lanzar el proceso hijo.

timeout

Plazo de ejecución en segundos.

output

Salida del proceso hijo si fue capturada por `run()` o `check_output()`. De otro modo, `None`.

stdout

Alias de `output`, por simetría con `stderr`.

stderr

Salida de `stderr` del proceso hijo si fue capturada por `run()`. De otro modo, `None`.

Nuevo en la versión 3.3.

Distinto en la versión 3.5: Se añadieron los atributos `stdout` y `stderr`

exception subprocess.CalledProcessError

Subclase de `SubprocessError`; se lanza cuando un proceso ejecutado con `check_call()` o `check_output()` retorna un estado distinto de cero.

returncode

Estado de salida del proceso hijo. Si el proceso terminó por causa de una señal, el estado será el número de la señal en negativo.

cmd

Orden que se utilizó para lanzar el proceso hijo.

output

Salida del proceso hijo si fue capturada por `run()` o `check_output()`. De otro modo, `None`.

stdout

Alias de `output`, por simetría con `stderr`.

stderr

Salida de `stderr` del proceso hijo si fue capturada por `run()`. De otro modo, `None`.

Distinto en la versión 3.5: Se añadieron los atributos `stdout` y `stderr`

Argumentos frecuentemente empleados

Para permitir una gran variedad de usos, el constructor de `Popen` (y las funciones asociadas) aceptan un gran número de argumentos opcionales. Para los usos más habituales, se pueden dejar de forma segura los valores por defecto. Los argumentos más frecuentemente necesarios son:

`args` se requiere en todas las llamadas; debe ser una cadena o una secuencia de argumentos al programa. En general, es mejor proporcionar una secuencia de argumentos porque permite que el módulo se ocupe de las secuencias de escape y los entrecomillados de los argumentos (por ejemplo, para permitir espacios en los nombres de fichero). Si se pasa una cadena simple, se ha de especificar `shell` como `True` (ver más adelante) o la cadena debe ser el nombre del programa a ejecutar sin especificar ningún argumento.

`stdin`, `stdout` y `stderr` especifican los flujos de la entrada estándar, la salida estándar y el error estándar, respectivamente. Los valores válidos son `PIPE`, `DEVNULL`, un descriptor de fichero existente (un entero positivo), un objeto fichero existente o `None`. `PIPE` indica que se ha de crear un nuevo pipe hacia el hijo. `DEVNULL` indica que se usará el fichero especial `os.devnull`. Con el valor por defecto `None`, no se realiza ninguna redirección y el hijo heredará los gestores de flujos del padre. Además, `stderr` puede ser `STDOUT`, que indica que los datos de `stderr` del proceso hijo serán capturados por el mismo gestor de flujo que `stdout`.

Si se especifican `encoding` o `errors`, o `text` (o su alias `universal_newlines`) es verdadero, se abrirán en modo texto los objetos fichero `stdin`, `stdout` y `stderr` usando los `encoding` y `errors` especificados en la llamada o los valores predeterminados de `io.TextIOWrapper`.

Para `stdin`, los saltos de línea `'\n'` de la entrada serán convertidos al separador de línea predeterminado `os.linesep`. Para `stdout` y `stderr`, todos los saltos de línea de la salida serán convertidos a `'\n'`. Hay más información en la documentación de la clase `io.TextIOWrapper` para el caso en que el argumento `newline` de su constructor es `None`.

Si no se usa el modo texto, `stdin`, `stdout` y `stderr` se abrirán como flujos binarios. No se realizará ninguna codificación ni conversión de salto de línea.

Nuevo en la versión 3.6: Se añadieron los parámetros `encoding` y `errors`.

Nuevo en la versión 3.7: Se añadió el parámetro `text` como alias de `universal_newlines`.

Nota: El atributo `newlines` de los objetos fichero `Popen.stdin`, `Popen.stdout` y `Popen.stderr` no es actualizado por el método `Popen.communicate()`.

Si `shell` es `True`, la orden especificada se ejecutará pasando por la shell. Esto tiene utilidad si se usa Python principalmente por el flujo de control mejorado sobre la mayoría de las shell de sistema, pero se desea también un acceso práctico a otras características de la shell, como pipes, nombres de fichero con comodines, expansión de variables de entorno o expansión de `~` al directorio `home` del usuario. Sin embargo, no se debe olvidar que el propio Python tiene implementaciones de muchas características tipo shell (en particular, `glob`, `fnmatch`, `os.walk()`, `os.path.expandvars()`, `os.path.expanduser()`, y `shutil`).

Distinto en la versión 3.3: Cuando `universal_newlines` es `True`, la clase usa la codificación `locale.getpreferredencoding(False)` en lugar de `locale.getpreferredencoding()`. Ver la clase `io.TextIOWrapper` para obtener más información sobre este cambio.

Nota: Leer la sección *Consideraciones sobre la seguridad* antes de usar `shell=True`.

Estas opciones y el resto se describen con más detalle en la documentación del constructor de `Popen`.

El constructor de `Popen`

El proceso interno de creación y gestión de este módulo lo gestiona la clase `Popen`. Proporciona una gran flexibilidad para que los desarrolladores sean capaces de gestionar los casos menos comunes que quedan sin cubrir por las funciones auxiliares.

class `subprocess.Popen` (*args*, *bufsize=-1*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=True*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=None*, *startupinfo=None*, *creationflags=0*, *restore_signals=True*, *start_new_session=False*, *pass_fds=()*, *, *encoding=None*, *errors=None*, *text=None*)

Ejecuta un programa hijo en un proceso nuevo. En POSIX, la clase se comporta como `os.execvp()` para lanzar el proceso hijo. En Windows, la clase usa la función `CreateProcess()` de Windows. Los argumentos de `Popen` son los siguientes.

args debe ser o una secuencia de argumentos de programa o una cadena simple o un *objeto tipo ruta*. Por omisión, el programa a ejecutar es el primer elemento de *args* si *args* es una secuencia. Si *args* es una cadena, la interpretación es dependiente de la plataforma, según se describe más abajo. Véase los argumentos *shell* y *executable* para más información sobre el comportamiento por defecto. Salvo que se indique, se recomienda pasar los *args* como una secuencia.

Un ejemplo del paso de argumentos a un programa externo mediante una secuencia:

```
Popen(["usr/bin/git", "commit", "-m", "Fixes a bug."])
```

En POSIX, si *args* es una cadena se interpreta como el nombre o la ruta del programa que ejecutar. Sin embargo, esto solamente funciona si no hay que pasar argumentos al programa.

Nota: Puede que no resulte evidente cómo descomponer una orden de la shell en una secuencia de argumentos, especialmente en casos complejos. `shlex.split()` puede aclarar cómo determinar la descomposición en tokens de *args*:

```
>>> import shlex, subprocess
>>> command_line = 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!
```

Hay que destacar en particular que las opciones (como *-input*) y los argumentos (como *eggs.txt*) que van separados por espacio en blanco en la shell van en elementos de lista separados, mientras los argumentos que necesitan entrecomillado o escapado de espacios cuando se usan en la shell (como los nombres de ficheros con espacios o la orden *echo* anteriormente mostrada) son elementos simples de la lista.

En Windows, si *args* es una secuencia, se convertirá a cadena del modo descrito en *Cómo convertir una secuencia de argumentos a una cadena en Windows*. Esto es así porque la función de bajo nivel `CreateProcess()` funciona sobre cadenas.

Distinto en la versión 3.6: El parámetro *args* toma un *objeto tipo ruta* si *shell* es `False` y una secuencia de objetos tipo fichero en POSIX.

Distinto en la versión 3.8: El parámetro *args* toma un *objeto tipo ruta* si *shell* es `False` y una secuencia de bytes y objetos tipo fichero en Windows.

El argumento *shell* (False por defecto) especifica si usar la shell como programa a ejecutar. Si **shell** es True, se recomienda pasar *args* como cadena mejor que como secuencia.

En POSIX con *shell=True*, la shell predeterminada es */bin/sh*. Si *args* es una cadena, ésta especifica la orden a ejecutar por la shell. Esto significa que la cadena tiene que tener el formato que tendría si se tecleara en la línea de órdenes. Esto incluye, por ejemplo, el entrecomillado y las secuencias de escape necesarias para los nombres de fichero que contengan espacios. Si *args* es una secuencia, el primer elemento especifica la cadena de la orden y cualquier otro elemento será tratado como argumentos adicionales a la propia shell. De este modo, *Popen* hace el equivalente de:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

En Windows con *shell=True*, la variable de entorno COMSPEC especifica la shell predeterminada. Solamente hace falta especificar *shell=True* en Windows cuando la orden que se desea ejecutar es interna a la shell (como. *dir* o *copy*). No hace falta especificar *shell=True* para ejecutar un fichero por lotes o un ejecutable de consola.

Nota: Leer la sección *Consideraciones sobre la seguridad* antes de usar *shell=True*.

Se proporcionará *bufsize* como el argumento correspondiente a la función *open()* cuando se creen los objetos fichero de los flujos *stdin/stdout/stderr*:

- 0 significa sin búfer (*read* y *write* son una llamada al sistema y pueden retornar datos parciales)
- 1 significa usar búfer de líneas (solamente se puede usar si *universal_newlines=True*, es decir, en modo texto)
- cualquier otro valor positivo indica que hay que usar un búfer de aproximadamente dicho tamaño
- *bufsize* negativo (el valor por defecto) indica que se use el valor predeterminado del sistema, *io.DEFAULT_BUFFER_SIZE*.

Distinto en la versión 3.3.1: *bufsize* es ahora -1 por defecto para permitir que el buffering por defecto se comporte como o que la mayoría del código espera. En versiones anteriores a Python 3.2.4 o 3.3.1 tomaba un valor por defecto de 0, sin búfer, lo que permitía lecturas demasiado cortas. Esto no era intencionado y no se correspondía con el comportamiento de Python 2 como la mayoría de códigos esperan.

El argumento *executable* especifica un programa de reemplazo que ejecutar. Esto es muy poco frecuente. Cuando *shell=False*, *executable* reemplaza al programa especificado por *args*. Sin embargo, se pasan los *args* originales al programa. La mayoría de los programas tratan el programa especificado en los *args* como el nombre del programa, que puede ser diferente del programa realmente ejecutado. En POSIX, el nombre en *args* funciona como nombre visible en utilidades como *ps*. Si *shell=True*, en POSIX el argumento *executable* especifica una shell de reemplazo de la predeterminada */bin/sh*.

Distinto en la versión 3.6: El parámetro *executable* acepta un *objeto tipo ruta* en POSIX.

Distinto en la versión 3.8: El parámetro *executable* acepta bytes y un *objeto tipo ruta* en POSIX.

Distinto en la versión 3.8.17: Changed Windows shell search order for *shell=True*. The current directory and %PATH% are replaced with %COMSPEC% and %SystemRoot%\System32\cmd.exe. As a result, dropping a malicious program named *cmd.exe* into a current directory no longer works.

stdin, *stdout* y *stderr* especifican los gestores de ficheros de entrada estándar, salida estándar y error estándar, respectivamente. Los valores válidos son *PIPE*, *DEVNULL*, un descriptor de fichero existente (un entero positivo), un *file object* existente, y *None*. *PIPE* indica que se ha de crear un nuevo pipe al hijo. *DEVNULL* indica que se usará el fichero especial *os.devnull*. Con los valores por omisión de *None*, no se llevará a cabo ninguna redirección; el proceso hijo heredará los gestores de fichero del proceso padre. Además, *stderr* puede ser *STDOUT*, que indica que los datos de *stderr* de las aplicaciones se capturarán sobre el mismo gestor de fichero de *stdout*.

Si se establece *preexec_fn* a un objeto invocable, se llamará a dicho objeto en el proceso hijo justo antes de que se ejecute el hijo. (solamente POSIX)

Advertencia: No es seguro utilizar el parámetro *preexec_fn* en presencia de hilos de ejecución en la aplicación. El proceso hijo podría bloquearse antes de llamar a *exec*. ¡Si es imprescindible, que sea tan simple como sea posible! Se ha de minimizar el número de librerías a las que se llama.

Nota: Si es necesario modificar el entorno para el proceso hijo, se debe utilizar el parámetro *env* en lugar de hacerlo en una *preexec_fn*. El parámetro *start_new_session* puede tomar el lugar de un uso anteriormente común de *preexec_fn* para llamar a *os.setsid* en el proceso hijo.

Distinto en la versión 3.8: Se ha abandonado el soporte de *preexec_fn* en subintérpretes. El uso de dicho parámetro en un subintérprete lanza *RuntimeError*. La nueva restricción puede afectar a aplicaciones desplegadas en *mod_wsgi*, *uWSGI* y otros entornos incrustados.

Si *close_fds* es verdadero, todos los descriptores de fichero salvo 0, 1 y 2 serán cerrados antes de ejecutar el proceso hijo. Por el contrario, cuando *close_fds* es falso, los descriptores de fichero obedecen su indicador de heredable según *Herencia de los descriptores de archivos*.

En Windows, si *close_fds* es verdadero el proceso hijo no heredará ningún gestor de fichero salvo que se le pasen explícitamente en el elemento *handle_list* de *STARTUPINFO.lpAttributeList*, o mediante redirección estándar.

Distinto en la versión 3.2: El valor predeterminado de *close_fds* se cambió de *False* a lo antes descrito.

Distinto en la versión 3.7: En Windows, el valor predeterminado de *close_fds* se cambió de *False* a *True* al redirigir los gestores estándar. Ahora es posible establecer *close_fds* a *True* cuando se redirigen los gestores estándar.

pass_fds es una secuencia de descriptor de ficheros opcional que han de mantenerse abiertos entre el proceso padre e hijo. Si se proporciona un valor a *pass_fds* se fuerza *close_fds* a *True*. (solamente POSIX)

Distinto en la versión 3.2: Se añadió el parámetro *pass_fds*.

Si *cwd* no es *None*, la función cambia el directorio de trabajo a *cwd* antes de ejecutar el proceso hijo. *cwd* puede ser una cadena, o un objeto bytes o *tipo ruta*. En particular, la función busca *executable* (o el primer elemento de *args*) relativo a *cwd* si la ruta del ejecutable es relativa.

Distinto en la versión 3.6: El parámetro *cwd* acepta un *objeto tipo ruta* en POSIX.

Distinto en la versión 3.7: El parámetro *cwd* acepta un *objeto tipo ruta* en Windows.

Distinto en la versión 3.8: El parámetro *cwd* acepta un objeto bytes en Windows.

Si *restore_signals* es verdadero (el valor por defecto) todas las señales que Python ha establecido a *SIG_IGN* se restauran a *SIG_DFL* en el proceso hijo antes del *exec*. En la actualidad, esto incluye las señales *SIGPIPE*, *SIGXFZ* y *SIGXFSZ* (solamente POSIX).

Distinto en la versión 3.2: Se añadió *restore_signals*.

Si *start_new_session* es verdadero la llamada al sistema *setsid* se hará en el proceso hijo antes de la ejecución del subprocesso (solamente POSIX).

Distinto en la versión 3.2: Se añadió *start_new_session*.

Si *env* no es *None*, debe ser un mapeo que defina las variables de entorno del nuevo proceso; se utilizarán éstas en lugar del comportamiento por defecto de heredar el entorno del proceso en curso.

Nota: Si se especifica, *env* debe proporcionar las variables necesarias para que el programa se ejecute. En Windows, para ejecutar una *side-by-side assembly* el *env* especificado **debe** incluir un *SystemRoot* válido.

Si se especifica *encoding* o *errors*, o *text* verdadero, los objetos fichero *stdin*, *stdout* y *stderr* se abren en modo texto con la codificación y *errors* especificados, según se describió en *Argumentos frecuentemente empleados*. El argumento *universal_newlines* es equivalente a *text* y se admite por compatibilidad hacia atrás. Por omisión, los ficheros se abren en modo binario.

Nuevo en la versión 3.6: Se añadieron *encoding* y *errors*.

Nuevo en la versión 3.7: Se añadió *text* como alias más legible de *universal_newlines*.

Si se proporciona, *startupinfo* será un objeto *STARTUPINFO*, que se pasa a la función de más bajo nivel *CreateProcess*. *creationflags*, si está presente, puede ser uno o más de los siguientes indicadores:

- *CREATE_NEW_CONSOLE*
- *CREATE_NEW_PROCESS_GROUP*
- *ABOVE_NORMAL_PRIORITY_CLASS*
- *BELOW_NORMAL_PRIORITY_CLASS*
- *HIGH_PRIORITY_CLASS*
- *IDLE_PRIORITY_CLASS*
- *NORMAL_PRIORITY_CLASS*
- *REALTIME_PRIORITY_CLASS*
- *CREATE_NO_WINDOW*
- *DETACHED_PROCESS*
- *CREATE_DEFAULT_ERROR_MODE*
- *CREATE_BREAKAWAY_FROM_JOB*

Se puede usar los objetos *Popen* como gestores de contexto mediante sentencia *with*: a la salida, los descriptores de flujo se cierran y se espera a que acabe el proceso.

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

Lanza un *evento de auditoría* *subprocess.Popen* con argumentos *executable*, *args*, *cwd*, *env*.

Distinto en la versión 3.2: Se añadió la funcionalidad de gestor de contexto.

Distinto en la versión 3.6: El destructor de *Popen* ahora emite una advertencia *ResourceWarning* si el proceso hijo todavía se está ejecutando.

Distinto en la versión 3.8: *Popen* puede usar *os.posix_spawn()* en algunos casos para obtener mejor rendimiento. En el Subsistema de Windows para Linux (WSL) y en la emulación del modo usuario de QEMU, el constructor de *Popen* que use *os.posix_spawn()* ya no lanzará una excepción cuando se den errores tales como que un programa no esté, sino que el proceso hijo fracasará con un *returncode* distinto de cero.

Excepciones

Las excepciones lanzadas en el proceso hijo, antes de que el nuevo programa haya empezado a ejecutarse, se relanzarán en el padre.

La excepción más comúnmente lanzada es *OSError*. Esto ocurre, por ejemplo, cuando se intenta ejecutar un fichero no existente. Las aplicaciones deben estar preparadas para gestionar las excepciones *OSError*.

Se lanzará un *ValueError* si se llama a *Popen* con argumentos no válidos.

check_call() y *check_output()* lanzarán un *CalledProcessError* si el proceso invocado retorna un código de retorno distinto de cero.

Todas las funciones y métodos que admiten un parámetro *timeout*, tales como *call()* y *Popen.communicate()* lanzarán *TimeoutExpired* si se vence el plazo de ejecución antes de que finalice el proceso hijo.

Todas las excepciones definidas en este módulo heredan de *SubprocessError*.

Nuevo en la versión 3.3: Se añadió la clase base *SubprocessError*.

17.6.2 Consideraciones sobre seguridad

Unlike some other `popen` functions, this library will not implicitly choose to call a system shell. This means that all characters, including shell metacharacters, can safely be passed to child processes. If the shell is invoked explicitly, via `shell=True`, it is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid [shell injection](#) vulnerabilities.

Cuando se usa `shell=True`, se puede usar la función `shlex.quote()` para escapar correctamente el espaciado y los metacaracteres de la shell en las cadenas que se vayan a utilizar para construir órdenes de la shell.

On Windows, batch files (`*.bat` or `*.cmd`) may be launched by the operating system in a system shell regardless of the arguments passed to this library. This could result in arguments being parsed according to shell rules, but without any escaping added by Python. If you are intentionally launching a batch file with arguments from untrusted sources, consider passing `shell=True` to allow Python to escape special characters. See [gh-114539](#) for additional discussion.

17.6.3 Objetos Popen

Las instancias de la clase `Popen` cuentan con los siguientes métodos:

`Popen.poll()`

Comprueba si el proceso hijo ha finalizado. Establece y retorna el atributo `returncode`. De lo contrario, retorna `None`.

`Popen.wait(timeout=None)`

Espera a que termine el proceso hijo. Establece y retorna el atributo `returncode`.

Si el proceso no finaliza tras `timeout` segundos, lanza una excepción `TimeoutExpired`. Se puede capturar esta excepción para reintentar la espera.

Nota: Esto causará un bloqueo cuando se use `stdout=PIPE` o `stderr=PIPE` y el proceso hijo genere suficiente salida hacia un pipe como para bloquear esperando que el búfer del pipe del SO acepte más datos. Se debe usar `Popen.communicate()` cuando se usen pipes para evitar esto.

Nota: Esta función se implementa mediante una espera activa (llamada no bloqueante y breves llamadas a `sleep`). Se debe usar el módulo `asyncio` para hacer una espera asíncrona: ver `asyncio.create_subprocess_exec`.

Distinto en la versión 3.3: Se añadió `timeout`.

`Popen.communicate(input=None, timeout=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 and set the `returncode` attribute. The optional `input` argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. If streams were opened in text mode, `input` must be a string. Otherwise, it must be bytes.

`communicate()` retorna una tupla (`stdout_data`, `stderr_data`). Los datos serán cadenas si se abrieron los flujos en modo texto, en caso contrario serán bytes.

Adviértase que si se desea enviar datos al stdin del proceso, se ha de crear el objeto `Popen` con `stdin=PIPE`. Análogamente, para obtener algo diferente de `None` en la tupla del resultado, hay que suministrar `stdout=PIPE` o `stderr=PIPE` también.

Si el proceso no termina tras `timeout` segundos, se lanza una excepción `TimeoutExpired`. Si se captura dicha excepción y se reintenta la comunicación, no se perderán datos de salida.

No se matará el proceso si vence el plazo de ejecución, así que para hacer limpieza, una aplicación correcta debería matar el proceso y terminar la comunicación:

```

proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()

```

Nota: Los datos leídos pasan por un búfer en memoria, así que no se ha de usar este método para un tamaño de datos grande o ilimitado.

Distinto en la versión 3.3: Se añadió *timeout*.

`Popen.send_signal(signal)`

Envía la señal *signal* al proceso hijo.

Nota: En Windows, SIGTERM es un alias de *terminate()*. Se puede enviar CTRL_C_EVENT y CTRL_BREAK_EVENT a los procesos creados con un parámetro *creationflags* que incluya CREAT_NEW_PROCESS_GROUP.

`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.

`Popen.kill()`

Kills the child. On POSIX OSs the function sends SIGKILL to the child. On Windows *kill()* is an alias for *terminate()*.

También están disponibles los siguientes atributos:

`Popen.args`

El argumento *args* según se pasó a *Popen*: o una secuencia de argumentos del programa o una cadena sencilla.

Nuevo en la versión 3.3.

`Popen.stdin`

Si el argumento *stdin* fue *PIPE*, este atributo es un objeto flujo escribible según lo retorna *open()*. Si se especificaron argumentos *encoding* o *errors* o el argumento *universal_newlines* fue True, el flujo es de texto, de lo contrario, es de bytes. Si el argumento *stdin* no fue *PIPE*, este atributo es None.

`Popen.stdout`

Si el argumento *stdout* fue *PIPE*, este atributo es un objeto de flujo legible según lo retorna *open()*. Leer del flujo proporciona salida del proceso hijo. Si se especificaron argumentos *encoding* o *errors* o el argumento *universal_newlines* fue True, el flujo es de texto, de lo contrario, es de bytes. Si el argumento *stdout* no fue *PIPE*, este atributo es None.

`Popen.stderr`

Si el argumento *stderr* fue *PIPE*, este atributo es un objeto de flujo legible según lo retorna *open()*. Leer del flujo proporciona salida del proceso hijo. Si se especificaron argumentos *encoding* o *errors* o el argumento *universal_newlines* fue True, el flujo es de texto, de lo contrario, es de bytes. Si el argumento *stderr* no fue *PIPE*, este atributo es None.

Advertencia: Se ha de usar *communicate()* en lugar de *.stdin.write*, *.stdout.read* o *.stderr.read* para evitar bloqueos por búfer de pipes del SO llenos que puedan bloquear el proceso hijo.

`Popen.pid`

El ID de proceso del hijo.

Adviértase que si se establece el argumento *shell* a True, éste es el ID de proceso de la shell generada.

Popen.returncode

El código de retorno del hijo, establecido por `poll()` y `wait()` (e indirectamente por `communicate()`). Un valor None indica que el proceso no ha terminado aún.

Un valor negativo `-N` indica que el hijo fue forzado a terminar con la señal `N` (solamente POSIX).

17.6.4 Elementos auxiliares de Popen en Windows

La clase `STARTUPINFO` y las siguientes constantes sólo están disponibles en Windows.

class `subprocess.STARTUPINFO` (*, `dwFlags=0`, `hStdInput=None`, `hStdOutput=None`, `hStdError=None`, `wShowWindow=0`, `lpAttributeList=None`)

Se utiliza un soporte parcial de la estructura `STARTUPINFO` de Windows para la creación de `Popen`. Se pueden establecer los siguientes atributos pasándolos como argumentos sólo por clave.

Distinto en la versión 3.7: Se añadió el soporte de argumentos sólo por clave.

dwFlags

Un campo bit que determina si se usan ciertos atributos de `STARTUPINFO` cuando el proceso crea una ventana.

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_
↳ USESHOWWINDOW
```

hStdInput

Si `dwFlags` especifica `STARTF_USESTDHANDLES`, este atributo es el gestor de entrada estándar del proceso. Si no se especifica `STARTF_USESTDHANDLES`, el valor predeterminado de entrada estándar es el búfer de teclado.

hStdOutput

Si `dwFlags` especifica `STARTF_USESTDHANDLES`, este atributo es el gestor de salida estándar del proceso. En caso contrario, se hace caso omiso del atributo y el valor predeterminado de salida estándar es el búfer de ventana.

hStdError

Si `dwFlags` especifica `STARTF_USESTDHANDLES`, este atributo es el gestor de error estándar del proceso. En caso contrario, se hace caso omiso del atributo y el valor predeterminado de error estándar es el búfer de ventana.

wShowWindow

Si `dwFlags` especifica `STARTF_USESHOWWINDOW`, este atributo puede ser cualquiera de los valores que pueden especificarse en el parámetro `nCmdShow` para la función `ShowWindow`, salvo `SW_SHOWDEFAULT`. De otro modo, se hace caso omiso del atributo.

Se proporciona `SW_HIDE` para este atributo. Se usa cuando se llama a `Popen` con `shell=True`.

lpAttributeList

Un diccionario de atributos adicionales para la creación del proceso, según `STARTUPINFOEX`, véase `UpdateProcThreadAttribute`.

Atributos admitidos:

handle_list Una secuencia de gestores que se heredará. `close_fds` debe ser verdadero si no viene vacío.

Los gestores deben hacerse temporalmente heredables por `os.set_handle_inheritable()` cuando se pasan al constructor de `Popen` o de lo contrario, se lanzará `OSError` con el error de Windows `ERROR_INVALID_PARAMETER` (87).

Advertencia: En un proceso multihilo, hay que evitar filtrar gestores no heredables cuando se combina esta característica con llamadas concurrentes a otras funciones de creación de procesos

que heredan todos los gestores, como `os.system()`. Esto también rige para la redirección de gestores estándar, que crea temporalmente gestores heredables.

Nuevo en la versión 3.7.

Constantes de Windows

El módulo `subprocess` expone las siguientes constantes.

`subprocess.STD_INPUT_HANDLE`

El dispositivo de entrada estándar. Inicialmente, es el búfer de entrada de la consola, `CONIN$`.

`subprocess.STD_OUTPUT_HANDLE`

El dispositivo de salida estándar. Inicialmente, es el búfer de pantalla de la consola activa, `CONOUT$`.

`subprocess.STD_ERROR_HANDLE`

El dispositivo de error estándar. Inicialmente, es el búfer de pantalla de la consola activa, `CONOUT$`.

`subprocess.SW_HIDE`

Ocultar la ventana. Se activará otra ventana.

`subprocess.STARTF_USESTDHANDLES`

Especifica que los atributos `STARTUPINFO.hStdInput`, `STARTUPINFO.hStdOutput`, y `STARTUPINFO.hStdError` contienen información adicional.

`subprocess.STARTF_USESHOWWINDOW`

Especifica que el atributo `STARTUPINFO.wShowWindow` contiene información adicional.

`subprocess.CREATE_NEW_CONSOLE`

El nuevo proceso obtiene una nueva consola, en lugar de heredar la consola del padre (que es el comportamiento predeterminado).

`subprocess.CREATE_NEW_PROCESS_GROUP`

Un parámetro `Popen creationflags` para especificar que se cree un nuevo grupo de procesos. Este indicador es necesario para usar `os.kill()` sobre el subprocesso.

Este indicador no se tiene en cuenta si se especifica `CREATE_NEW_CONSOLE`.

`subprocess.ABOVE_NORMAL_PRIORITY_CLASS`

Parámetro `Popen creationflags` para especificar que el nuevo proceso tendrá una prioridad superior a la media.

Nuevo en la versión 3.7.

`subprocess.BELOW_NORMAL_PRIORITY_CLASS`

Parámetro `Popen creationflags` para especificar que el nuevo proceso tendrá una prioridad inferior a la media.

Nuevo en la versión 3.7.

`subprocess.HIGH_PRIORITY_CLASS`

Parámetro `Popen creationflags` para especificar que el nuevo proceso tendrá una prioridad elevada.

Nuevo en la versión 3.7.

`subprocess.IDLE_PRIORITY_CLASS`

Parámetro `Popen creationflags` para especificar que el nuevo proceso tendrá una la mínima prioridad.

Nuevo en la versión 3.7.

`subprocess.NORMAL_PRIORITY_CLASS`

Parámetro `Popen creationflags` para especificar que el nuevo proceso tendrá una prioridad normal (éste es el valor predeterminado).

Nuevo en la versión 3.7.

subprocess.REALTIME_PRIORITY_CLASS

Parámetro *Popen* *creationflags* para especificar que el nuevo proceso tendrá una prioridad de tiempo real. Raramente se debería usar `REALTIME_PRIORITY_CLASS`, porque esto interrumpe los hilos que gestionan la entrada del ratón y del teclado, así como la gestión del volcado de búfer del disco. Esta clase es apropiada para aplicaciones que se comunican directamente con el hardware o que llevan a cabo tareas breves que no admitan interrupciones.

Nuevo en la versión 3.7.

subprocess.CREATE_NO_WINDOW

Parámetro *Popen* *creationflags* para especificar que el nuevo proceso no creará una ventana.

Nuevo en la versión 3.7.

subprocess.DETACHED_PROCESS

Parámetro *Popen* *creationflags* para especificar que el nuevo proceso no heredará la consola del padre. Este valor es incompatible con `CREATE_NEW_CONSOLE`.

Nuevo en la versión 3.7.

subprocess.CREATE_DEFAULT_ERROR_MODE

Parámetro *Popen* *creationflags* para especificar que el nuevo proceso no hereda el modo de error del proceso padre. En su lugar, el proceso parte del modo de error predeterminado. Esta característica es particularmente útil para aplicaciones de shell multihilo que se ejecutan con los errores “duros” desactivados.

Nuevo en la versión 3.7.

subprocess.CREATE_BREAKAWAY_FROM_JOB

Parámetro *Popen* *creationflags* para especificar que el nuevo proceso no está asociado a la tarea.

Nuevo en la versión 3.7.

17.6.5 Antigua API de alto nivel

Antes de Python 3.5, estas tres funciones conformaban la API de alto nivel para subprocessos. Ahora se puede usar *run()* en muchos casos, pero hay mucho código escrito con estas funciones.

subprocess.call(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, *meout=None*, ***other_popen_kwargs*)

Ejecutar la orden descrita por *args*. Esperar que la orden se complete y retornar al atributo *returncode*.

El código que requiera capturar *stdout* o *stderr* debería usar *run()* en su lugar:

```
run(...).returncode
```

Para suprimir *stdout* o *stderr* se ha de proporcionar un valor de *DEVNULL*.

Se muestran algunos argumentos comunes. La signatura completa de la función es la misma que la del constructor de *Popen*; esta función pasa todos los argumentos proporcionados (salvo *timeout*) directamente a esa interfaz.

Nota: No usar *stdout=PIPE* ni *stderr=PIPE* con esta función. El proceso hijo se bloqueará si genera suficiente salida a un pipe como para saturar el búfer del pipe del sistema operativo mientras no se lee de los pipes.

Distinto en la versión 3.3: Se añadió *timeout*.

Distinto en la versión 3.8.17: Changed Windows shell search order for *shell=True*. The current directory and *%PATH%* are replaced with *%COMSPEC%* and *%SystemRoot%\System32\cmd.exe*. As a result, dropping a malicious program named *cmd.exe* into a current directory no longer works.

subprocess.check_call(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, *meout=None*, ***other_popen_kwargs*)

Ejecuta la instrucción con argumentos. Espera a que la instrucción se complete. Si el código de retorno es cero,

retornar; en caso contrario, lanzar `CalledProcessError`. El objeto `CalledProcessError` tendrá el código de retorno en el atributo `returncode`.

El código que requiera capturar stdout o stderr debería usar `run()` en su lugar:

```
run(..., check=True)
```

Para suprimir stdout o stderr se ha de proporcionar un valor de `DEVNULL`.

Se muestran algunos argumentos comunes. La signatura completa de la función es la misma que la del constructor de `Popen`; esta función pasa todos los argumentos proporcionados (salvo `timeout`) directamente a esa interfaz.

Nota: No usar `stdout=PIPE` ni `stderr=PIPE` con esta función. El proceso hijo se bloqueará si genera suficiente salida a un pipe como para saturar el búfer del pipe del sistema operativo mientras no se lee de los pipes.

Distinto en la versión 3.3: Se añadió `timeout`.

Distinto en la versión 3.8.17: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

```
subprocess.check_output(args, *, stdin=None, stderr=None, shell=False, cwd=None, encoding=None, errors=None, universal_newlines=None, timeout=None, text=None, **other_popen_kwargs)
```

Ejecuta orden con argumentos y retorna su salida.

Si el código de retorno fue diferente de cero lanza un `CalledProcessError`. El objeto `CalledProcessError` tendrá el código de retorno en el atributo `returncode` y los datos de salida en el atributo `output`.

Esto equivale a:

```
run(..., check=True, stdout=PIPE).stdout
```

The arguments shown above are merely some common ones. The full function signature is largely the same as that of `run()` - most arguments are passed directly through to that interface. One API deviation from `run()` behavior exists: passing `input=None` will behave the same as `input=b''` (or `input=''`, depending on other arguments) rather than using the parent's standard input file handle.

Por omisión, esta función retornará los datos como bytes codificados. La codificación real de los datos podría depender de la orden invocada, por lo que la decodificación a texto se deberá hacer al nivel de la aplicación.

Este comportamiento se puede modificar estableciendo `text`, `encoding`, `errors`, o `universal_newlines` a `True`, como se describe en *Argumentos frecuentemente empleados* y `run()`.

Para capturar también el error estándar del resultado se debe usar `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'
```

Nuevo en la versión 3.1.

Distinto en la versión 3.3: Se añadió `timeout`.

Distinto en la versión 3.4: Se añadió soporte para el argumento por clave `input`.

Distinto en la versión 3.6: Se añadieron `encoding` y `errors`. Ver `run()` para más detalles.

Nuevo en la versión 3.7: Se añadió `text` como alias más legible de `universal_newlines`.

Distinto en la versión 3.8.17: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

17.6.6 Cómo reemplazar anteriores funciones con el módulo `subprocess`

En esta sección, «a se convierte en b» significa que b se puede usar en lugar de.

Nota: Todas las funciones «a» de esta sección fracasan silenciosamente (o casi) si no se halla el programa ejecutado; las funciones de reemplazo «b» lanzan `OSError` en lugar de esto.

Además, las funciones de reemplazo que usan `check_output()` fracasarán con un error `CalledProcessError` si la operación solicitada produce un código de retorno diferente de cero. La salida queda disponible en el atributo `output` de la excepción lanzada.

En los siguientes ejemplos, se asume que las funciones relevantes ya han sido importadas del módulo `subprocess`.

Cómo reemplazar la sustitución de órdenes de `/bin/sh`

```
output=$(mycmd myarg)
```

se convierte en:

```
output = check_output(["mycmd", "myarg"])
```

Cómo reemplazar los flujos de la shell

```
output=$(dmesg | grep hda)
```

se convierte en:

```
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]
```

La llamada a `p1.stdout.close()` tras lanzar `p2` es importante para que `p1` reciba un `SIGPIPE` si `p2` retorna antes que `p1`.

Alternativamente, para entrada de confianza, se puede usar el propio soporte de pipeline de la shell directamente:

```
output=$(dmesg | grep hda)
```

se convierte en:

```
output=check_output("dmesg | grep hda", shell=True)
```


Cómo reemplazar `os.system()`

```
sts = os.system("mycmd" + " myarg")
# becomes
sts = call("mycmd" + " myarg", shell=True)
```

Notas:

- No suele hacer falta llamar al programa a través de la shell.

Un ejemplo más creíble:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

Cómo reemplazar la familia `os.spawn`

Ejemplo de `P_NOWAIT`:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

Ejemplo de `P_WAIT`:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Ejemplo de vector:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Ejemplo de entorno:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

Cómo reemplazar `os.popen()`, `os.popen2()`, `os.popen3()`

```
(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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
==>
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)
```

La gestión del código de retorno se traduce así:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

Cómo reemplazar las funciones del módulo `popen2`

Nota: Si el argumento *cmd* de las funciones `popen2` es una cadena, la orden se ejecuta a través de */bin/sh*. Si es una lista, la orden se ejecuta directamente.

```
(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)
```

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

`popen2.Popen3` y `popen2.Popen4` funcionan a grandes rasgos como *subprocess.Popen*, salvo:

- *Popen* lanza una excepción si falla la ejecución.
- El argumento *capturestderr* se sustituye por el argumento *stderr*.
- Se ha de especificar *stdin=PIPE* y *stdout=PIPE*.
- `popen2` cierra todos los descriptores de fichero por omisión, pero se ha de especificar *close_fds=True* con *Popen* para garantizar este comportamiento en todas las plataformas o en versiones anteriores de Python.

17.6.7 Funciones de llamada a la shell de retrocompatibilidad

Este módulo también proporciona las siguientes funciones de compatibilidad del módulo `commands` de las versiones 2.X. Estas operaciones invocan implícitamente a la shell del sistema y no se les aplican ninguna de las garantías descritas anteriormente respecto a seguridad o consistencia en la gestión de excepciones.

`subprocess.getstatusoutput(cmd)`

Retorna `(exitcode, output)` de ejecutar `cmd` en una shell.

Ejecuta la cadena `cmd` en una shell con `Popen.check_output()` y retorna una tupla `(exitcode, output)`. Se utiliza la codificación de localización activa; consultar las notas sobre [Argumentos frecuentemente empleados](#) para más información.

Se elimina un salto de línea final de la salida. El código de salida de la orden se puede interpretar como el código de retorno del subproceso. Por ejemplo:

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

Disponibilidad: POSIX y Windows.

Distinto en la versión 3.3.4: Se añadió soporte de Windows.

La función ahora retorna `(exitcode, output)` en lugar de `(status, output)` como en Python 3.3.3 y anteriores. `exitcode` tiene el mismo valor que `returncode`.

`subprocess.getoutput(cmd)`

Retorna la salida (`stdout` y `stderr`) de ejecutar `cmd` en una shell.

Como `getstatusoutput()`, salvo que se ignora el código de salida y el valor retornado es una cadena que contiene la salida del comando. Por ejemplo:

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

Disponibilidad: POSIX y Windows.

Distinto en la versión 3.3.4: Se añadió soporte de Windows

17.6.8 Notas

Cómo convertir una secuencia de argumentos a una cadena en Windows

En Windows, para convertir una secuencia `args` en una cadena que puede ser analizada con las siguientes reglas (correspondientes a las reglas que usa la biblioteca de ejecución de MS C):

1. Los argumentos se separan por espacio en blanco, que debe ser un espacio o un tabulador.
2. Una cadena entre comillas dobles se interpreta como un argumento simple, sin importar los espacios en blanco que contenga. Se puede incrustar una cadena entre comillas en un argumento.
3. Una comilla doble precedida de una barra invertida se interpreta literalmente como una comilla doble.
4. Las barras invertidas se interpretan literalmente, salvo que precedan a una comilla doble.
5. Si las barras invertidas preceden inmediatamente a una doble comilla, cada par de barras invertidas se interpreta como una barra invertida literal. Si el número de barras invertidas es impar, la última barra invertida escapa la siguiente comilla doble según se describe en la regla 3.

Ver también:

`shlex` Módulo que proporciona una función para analizar y escapar líneas de órdenes.

17.7 sched — Eventos del planificador

Código fuente: [Lib/sched.py](#)

El módulo `sched` define una clase que implementa un planificador de eventos de propósito general:

class `sched.scheduler` (*timefunc=time.monotonic, delayfunc=time.sleep*)

La clase `scheduler` define una interfaz genérica para planificar eventos. Necesita dos funciones para tratar con el «mundo exterior» — *timefunc* debe poder llamarse sin argumentos y retornar un número (el «time», en cualquier unidad). La función *delayfunc* debería ser invocable con un argumento, compatible con la salida de *timefunc*, y debería retrasar tantas unidades de tiempo. *delayfunc* también se llamará con el argumento 0 después de que se ejecute cada evento para permitir que otros hilos tengan la oportunidad de ejecutarse en aplicaciones multihilo.

Distinto en la versión 3.3: Los argumentos *timefunc* and *delayfunc* son opcionales.

Distinto en la versión 3.3: `scheduler` La clase se puede usar de forma segura en entornos multihilo.

Ejemplo:

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.run()
...     print(time.time())
...
>>> print_some_times()
930343690.257
From print_time 930343695.274 positional
From print_time 930343695.275 keyword
From print_time 930343700.273 default
930343700.276
```

17.7.1 Objetos de Scheduler

`scheduler` Las sentencias tienen los siguientes métodos y atributos:

`scheduler.enterabs` (*time, priority, action, argument=(), kwargs={}*)

Planifica un nuevo evento. El argumento *time* debe ser un tipo numérico compatible con el valor de retorno de la función *timefunc* que se pasa al constructor. Los eventos planificados para la misma *time* se ejecutarán en el orden de su *priority*. Un número más bajo representa una prioridad más alta.

Ejecutar el evento significa ejecutar `action(*argument, **kwargs)`. *argument* es una secuencia que contiene los argumentos posicionales para *action*. *kwargs* es un diccionario que contiene los argumentos de palabras clave para *action*.

El valor de retorno es un evento que puede usarse para una cancelación posterior del evento (ver `cancel()`).

Distinto en la versión 3.3: El argumento *argument* es opcional.

Distinto en la versión 3.3: Se agregó el argumento *kwargs*.

`scheduler.enter(delay, priority, action, argument=(), kwargs={})`

Planifica un evento *delay* para más unidades de tiempo. Aparte del tiempo relativo, los otros argumentos, el efecto y el valor de retorno son los mismos que para `enterabs()`.

Distinto en la versión 3.3: El argumento *argument* es opcional.

Distinto en la versión 3.3: Se agregó el argumento *kwargs*.

`scheduler.cancel(event)`

Elimina el evento de la cola. Si *event* no es un evento actualmente en la cola, este método generará un `ValueError`.

`scheduler.empty()`

Retorna `True` si la cola de eventos está vacía.

`scheduler.run(blocking=True)`

Ejecuta todos los eventos programados. Este método esperará (usando la función `delayfunc()` enviada al constructor) el próximo evento, luego lo ejecutará y así sucesivamente hasta que no haya más eventos programados.

Si *blocking* es falso, se ejecutan los eventos planificados que expiran mas pronto (si corresponde) y luego retorna la fecha límite de la próxima llamada programada en el planificador (si corresponde).

action o *delayfunc* pueden generar una excepción. En cualquier caso, el planificador mantendrá un estado consistente y propagará la excepción. Si *action* genera una excepción, el evento no se intentará en futuras llamadas a `run()`.

Si una secuencia de eventos tarda más en ejecutarse que el tiempo disponible antes del próximo evento, el planificador simplemente se retrasará. No se descartarán eventos; el código de llamada es responsable de cancelar eventos que ya no son oportunos.

Distinto en la versión 3.3: Se agregó el argumento *blocking*.

`scheduler.queue`

Atributo de solo lectura que retorna una lista de los próximos eventos en el orden en que se ejecutarán. Cada evento se muestra como un *named tuple* con los siguientes campos : *time*, *priority*, *action*, *argument*, *kwargs*.

17.8 queue — clase de cola sincronizada

Código fuente: `Lib/queue.py`

El módulo `queue` implementa colas multi-productor y multi-consumidor. Es especialmente útil en la programación en hilo cuando la información debe intercambiarse de forma segura entre varios subprocesos. La clase `Queue` de este módulo implementa toda la semántica de bloqueo necesaria.

El módulo implementa tres tipos de cola, que difieren sólo en el orden en que se recuperan las entradas. En una cola FIFO, las primeras tareas añadidas son las primeras recuperadas. En una cola LIFO, la última entrada añadida es la primera recuperada (operando como una pila). En una cola de prioridad, las entradas se mantienen ordenadas (usando el módulo `heapq`) y la entrada de menor valor se recupera primero.

Internamente, estos tres tipos de colas utilizan bloqueos para bloquear temporalmente los hilos que compiten entre sí; sin embargo, no están diseñadas para manejar la reposición dentro de un hilo.

Además, el módulo implementa un tipo de cola «simple» FIFO, `SimpleQueue`, cuya implementación específica proporciona garantías adicionales a cambio de una funcionalidad menor.

El módulo `queue` define las siguientes clases y excepciones:

class `queue.Queue(maxsize=0)`

Constructor para una cola FIFO. *maxsize* es un número entero que establece el límite superior del número de elementos que pueden ser colocados en la cola. La inserción se bloqueará una vez que se haya alcanzado este

tamaño, hasta que se consuman los elementos de la cola. Si *maxsize* es menor o igual a cero, el tamaño de la cola es infinito.

class `queue.LifoQueue (maxsize=0)`

Constructor para una cola LIFO. *maxsize* es un número entero que establece el límite superior del número de elementos que pueden ser colocados en la cola. La inserción se bloqueará una vez que se haya alcanzado este tamaño, hasta que se consuman los elementos de la cola. Si *maxsize* es menor o igual a cero, el tamaño de la cola es infinito.

class `queue.PriorityQueue (maxsize=0)`

Constructor para una cola de prioridad. *maxsize* es un número entero que establece el límite superior del número de elementos que pueden ser colocados en la cola. La inserción se bloqueará una vez que se haya alcanzado este tamaño, hasta que se consuman los elementos de la cola. Si *maxsize* es menor o igual a cero, el tamaño de la cola es infinito.

Las entradas de menor valor se recuperan primero (la entrada de menor valor es la retornada por `sorted(list(entries))[0]`). Un patrón típico para las entradas es una tupla en la forma: (número_de_prioridad, datos).

Si los elementos de *datos* no son comparables, los datos pueden envolverse en una clase que ignore el elemento de datos y sólo compare el número de prioridad:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

class `queue.SimpleQueue`

Constructor de una cola sin límites FIFO. Las colas simples carecen de funcionalidad avanzada como el seguimiento de tareas.

Nuevo en la versión 3.7.

exception `queue.Empty`

Excepción lanzada cuando el objeto `get()` (o `get_nowait()`) que no se bloquea es llamado en un objeto `Queue` que está vacío.

exception `queue.Full`

Excepción lanzada cuando el objeto `put()` (o `put_nowait()`) que no se bloquea es llamado en un objeto `Queue` que está lleno.

17.8.1 Objetos de la cola

Los objetos de la cola (`Queue`, `LifoQueue`, o `PriorityQueue`) proporcionan los métodos públicos descritos a continuación.

`Queue.qsize()`

Retorna el tamaño aproximado de la cola. Nota, `qsize() > 0` no garantiza que un `get()` posterior no se bloquee, ni `qsize() < maxsize` garantiza que `put()` no se bloquee.

`Queue.empty()`

Retorna `True` si la cola está vacía, `False` si no. Si `empty()` retorna `True` no garantiza que una llamada posterior a `put()` no se bloquee. Del mismo modo, si `empty()` retorna `False` no garantiza que una llamada posterior a `get()` no se bloquee.

`Queue.full()`

Retorna `True` si la cola está llena, `False` si no. Si `full()` retorna `True` no garantiza que una llamada posterior a `get()` no se bloquee. Del mismo modo, si `full()` retorna `False` no garantiza que una llamada posterior a `put()` no se bloquee.

`Queue.put(item, block=True, timeout=None)`

Pone el *item* en la cola. Si el argumento opcional *block* es verdadero y *timeout* es `None` (el predeterminado), bloquea si es necesario hasta que un espacio libre esté disponible. Si *timeout* es un número positivo, bloquea como máximo *timeout* segundos y aumenta la excepción `Full` si no había ningún espacio libre disponible en ese tiempo. De lo contrario (*block* es falso), pone un elemento en la cola si un espacio libre está disponible inmediatamente, o bien levanta la excepción `Full` (*timeout* es ignorado en ese caso).

`Queue.put_nowait(item)`

Equivalente a `put(item, False)`.

`Queue.get(block=True, timeout=None)`

Retira y retorna un elemento de la cola. Si el argumento opcional *block* es verdadero y *timeout* es `None` (el predeterminado), bloquea si es necesario hasta que un elemento esté disponible. Si *timeout* es un número positivo, bloquea como máximo *timeout* segundos y aumenta la excepción `Empty` si no había ningún elemento disponible en ese tiempo. De lo contrario (*block* es falso), retorna un elemento si uno está disponible inmediatamente, o bien lanza la excepción `Empty` (*timeout* es ignorado en ese caso).

Antes de la 3.0 en los sistemas POSIX, y para todas las versiones en Windows, si *block* es verdadero y *timeout* es `None`, esta operación entra en una espera ininterrumpida en una cerradura subyacente. Esto significa que no puede haber excepciones, y en particular una `SIGINT` no disparará una `KeyboardInterrupt`.

`Queue.get_nowait()`

Equivalente a `get(False)`.

Se ofrecen dos métodos para apoyar el seguimiento si las tareas en cola han sido completamente procesadas por hilos *daemon* de consumo.

`Queue.task_done()`

Indica que una tarea anteriormente en cola está completa. Utilizado por los hilos de la cola de consumo. Por cada `get()` usado para recuperar una tarea, una llamada posterior a `task_done()` le dice a la cola que el procesamiento de la tarea está completo.

Si un `join()` se está bloqueando actualmente, se reanudará cuando todos los ítems hayan sido procesados (lo que significa que se recibió una llamada `task_done()` por cada ítem que había sido `put()` en la cola).

Lanza un `ValueError` si se llama más veces de las que hay elementos colocados en la cola.

`Queue.join()`

Bloquea hasta que todos los artículos de la cola se hayan obtenido y procesado.

El conteo de tareas sin terminar sube cada vez que se añade un elemento a la cola. El conteo baja cuando un hilo de consumidor llama `task_done()` para indicar que el elemento fue recuperado y todo el trabajo en él está completo. Cuando el conteo de tareas sin terminar cae a cero, `join()` se desbloquea.

Ejemplo de cómo esperar a que se completen las tareas en cola:

```
import threading, queue

q = queue.Queue()

def worker():
    while True:
        item = q.get()
        print(f'Working on {item}')
        print(f'Finished {item}')
        q.task_done()

# turn-on the worker thread
threading.Thread(target=worker, daemon=True).start()

# send thirty task requests to the worker
for item in range(30):
    q.put(item)
print('All task requests sent\n', end='')
```

(continué en la próxima página)

(proviene de la página anterior)

```
# block until all tasks are done
q.join()
print('All work completed')
```

17.8.2 Objetos de cola simple

Los objetos `SimpleQueue` proporcionan los métodos públicos descritos a continuación.

`SimpleQueue.qsize()`

Retorna el tamaño aproximado de la cola. Nota, `qsize() > 0` no garantiza que un `get()` posterior no se bloquee.

`SimpleQueue.empty()`

Retorna `True` si la cola está vacía, `False` si no. Si `empty()` retorna `False` no garantiza que una llamada posterior a `get()` no se bloquee.

`SimpleQueue.put(item, block=True, timeout=None)`

Pone el elemento en la cola. El método nunca se bloquea y siempre tiene éxito (excepto por posibles errores de bajo nivel como la falta de asignación de memoria). Los argumentos opcionales `block` y `timeout` son ignorados y sólo se proporcionan por compatibilidad con `Queue.put()`.

CPython implementation detail: This method has a C implementation which is reentrant. That is, a `put()` or `get()` call can be interrupted by another `put()` call in the same thread without deadlocking or corrupting internal state inside the queue. This makes it appropriate for use in destructors such as `__del__` methods or `weakref` callbacks.

`SimpleQueue.put_nowait(item)`

Equivalente a `put(item)`, siempre y cuando sea compatible con `Queue.put_nowait()`.

`SimpleQueue.get(block=True, timeout=None)`

Retira y retorna un elemento de la cola. Si los argumentos opcionales `block` son `true` y `timeout` es `None` (el pre-determinado), bloquea si es necesario hasta que un elemento esté disponible. Si `timeout` es un número positivo, bloquea como máximo `timeout` segundos y lanza la excepción `Empty` si no había ningún elemento disponible en ese tiempo. De lo contrario (`block` es falso), retorna un elemento si uno está disponible inmediatamente, o bien lanza la excepción `Empty` (`timeout` es ignorado en ese caso).

`SimpleQueue.get_nowait()`

Equivalente a `get(False)`.

Ver también:

Clase `multiprocessing.Queue` Una clase de cola para su uso en un contexto de multiprocesamiento (en lugar de multihilo).

`collections.deque` es una implementación alternativa de colas sin límites con operaciones atómicas rápidas `append()` y `popleft()` que no requieren bloqueo y también soportan indexación.

17.9 contextvars — Variables de Contexto

Este módulo proporciona APIs para gestionar, almacenar y acceder a estados en el contexto local. La clase `ContextVar` se utiliza para declarar y trabajar con Variables de Contexto (*Context Variables*). La función `copy_context()` y la clase `Context` deberían ser utilizadas para gestionar el contexto actual en frameworks asíncronos.

Los gestores de contexto que tienen un estado establecido deberían utilizar Variables de Contexto en lugar de `threading.local()`, para así evitar que este estado se inyecte inesperadamente a otro código, cuando se utilice en código concurrente.

Ver **PEP 567** para más detalles.

Nuevo en la versión 3.7.

17.9.1 Variables de Contexto

class `contextvars.ContextVar` (*name*[, *, *default*])

Esta clase se utiliza para declarar una nueva Variable de Contexto, por ejemplo:

```
var: ContextVar[int] = ContextVar('var', default=42)
```

El parámetro obligatorio *name* se utiliza para introspección y depuración.

El parámetro opcional de sólo palabra clave *default* es utilizado por `ContextVar.get()`, cuando en el contexto actual no se encuentra ningún valor para la variable.

Importante: las Variables de Contexto deberían ser creadas en lo más alto a nivel de módulo y nunca en clausura. Los objetos `Context` mantienen referencias a variables de contexto, lo cual no permitiría que estas variables de contexto sean limpiadas por el recolector de basura.

name

El nombre de la variable. Propiedad de sólo lectura.

Nuevo en la versión 3.7.1.

get ([*default*])

Retorna un valor para la variable de contexto en el contexto actual.

Si la variable no tiene ningún valor en el contexto actual, el método:

- retornará el valor del argumento *default* del método, si alguno fue dado; o
- retornará el valor por defecto de la variable de contexto, si ésta fue creada con alguno; o
- lanzará `LookupError`.

set (*value*)

Establece un nuevo valor para la variable de contexto en el contexto actual.

El argumento obligatorio *value* es el nuevo valor de la variable de contexto.

Retorna un objeto `Token` que puede utilizarse para restaurar la variable a su valor anterior, utilizando el método `ContextVar.reset()`.

reset (*token*)

Restablece la variable de contexto al valor que tenía antes de llamar al método `ContextVar.set()`, que creó el *token* utilizado.

Por ejemplo:

```
var = ContextVar('var')

token = var.set('new value')
# code that uses 'var'; var.get() returns 'new value'.
var.reset(token)

# After the reset call the var has no value again, so
# var.get() would raise a LookupError.
```

class `contextvars.Token`

Los objetos *token* son retornados por el método `ContextVar.set()`. Se le pueden dar al método `ContextVar.reset()` para restablecer el valor de la variable al que estuviese dado antes del *set* correspondiente.

var

Propiedad de sólo lectura. Apunta al objeto `ContextVar` que creó el *token*.

old_value

Propiedad de sólo lectura. Es el valor que la variable tenía antes de llamar al método `ContextVar.set()` que creó el `token`. Apunta a `Token.MISSING` si la variable no estaba establecida antes de la llamada.

MISSING

Marcador utilizado por `Token.old_value`.

17.9.2 Gestión de Contexto Manual

`contextvars.copy_context()`

Retorna una copia del objeto `Context` actual.

El siguiente código obtiene una copia del contexto actual e imprime todas las variables y sus valores establecidos en el contexto:

```
ctx: Context = copy_context()
print(list(ctx.items()))
```

La función tiene una complejidad de $O(1)$; es decir, trabaja a la misma velocidad en contextos con pocas o con muchas variables de contexto.

class contextvars.Context

Maapeo de `ContextVars` con sus valores.

`Context()` crea un contexto vacío sin valores. Para obtener una copia del contexto actual, se puede utilizar la función `copy_context()`.

`Context` implementa la interfaz `collections.abc.Mapping`.

run (*callable*, *args, **kwargs)

Ejecuta el código de `callable(*args, **kwargs)` en el objeto de contexto del cual se llama al método `run`. Retorna el resultado de la ejecución, o propaga una excepción si alguna ocurre.

Cualquier cambio realizado por `callable` sobre cualquier variable de contexto será contenido en el objeto de contexto:

```
var = ContextVar('var')
var.set('spam')

def main():
    # 'var' was set to 'spam' before
    # calling 'copy_context()' and 'ctx.run(main)', so:
    # var.get() == ctx[var] == 'spam'

    var.set('ham')

    # Now, after setting 'var' to 'ham':
    # var.get() == ctx[var] == 'ham'

ctx = copy_context()

# Any changes that the 'main' function makes to 'var'
# will be contained in 'ctx'.
ctx.run(main)

# The 'main()' function was run in the 'ctx' context,
# so changes to 'var' are contained in it:
# ctx[var] == 'ham'

# However, outside of 'ctx', 'var' is still set to 'spam':
# var.get() == 'spam'
```

El método lanzará `RuntimeError` cuando es llamado desde el mismo objeto de contexto desde más de un hilo del sistema operativo, o si se llama recursivamente.

copy()

Return a shallow copy of the context object.

var in context

Retorna `True` si `context` tiene un valor establecido para `var`; de lo contrario, retorna `False`.

context[var]

Retorna el valor de la variable `ContextVar` `var`. Si la variable no está establecida en el contexto actual, se lanzará `KeyError`.

get(var[, default])

Retorna el valor de `var`, si `var` tiene el valor en el objeto de contexto; de lo contrario, retorna `default`. Si `default` no es dado, retorna `None`.

iter(context)

Retorna un iterador de las variables almacenadas en el objeto de contexto.

len(proxy)

Retorna el número de variables establecidas en el objeto de contexto.

keys()

Retorna un listado de todas las variables en el objeto de contexto.

values()

Retorna un listado de los valores de todas las variables en el objeto de contexto.

items()

Retorna un listado de dos tuplas que contienen todas las variables y sus valores en el contexto actual.

17.9.3 Soporte *asyncio*

Las variables de contexto están soportadas de forma nativa en `asyncio` y se pueden utilizar sin ninguna configuración adicional. Por ejemplo, el siguiente código crea un servidor simple de respuesta, que utiliza una variable de contexto que hace que la dirección del cliente remoto esté disponible en la `Task` que gestiona al cliente:

```
import asyncio
import contextvars

client_addr_var = contextvars.ContextVar('client_addr')

def render_goodbye():
    # The address of the currently handled client can be accessed
    # without passing it explicitly to this function.

    client_addr = client_addr_var.get()
    return f'Good bye, client @ {client_addr}\n'.encode()

async def handle_request(reader, writer):
    addr = writer.transport.get_extra_info('socket').getpeername()
    client_addr_var.set(addr)

    # In any code that we call is now possible to get
    # client's address by calling 'client_addr_var.get()'.

    while True:
        line = await reader.readline()
        print(line)
        if not line.strip():
            break
        writer.write(line)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
writer.write(render_goodbye())
writer.close()

async def main():
    srv = await asyncio.start_server(
        handle_request, '127.0.0.1', 8081)

    async with srv:
        await srv.serve_forever()

asyncio.run(main())

# To test it you can use telnet:
#     telnet 127.0.0.1 8081
```

He aquí módulos de apoyo para algunos de los servicios mencionados:

17.10 `_thread` — API de bajo nivel para manejo de hilos

Este módulo ofrece primitivas de bajo nivel para trabajar con múltiples *threads* o hilos (también llamados *light-weight processes* o *tasks*) – múltiples hilos de control compartiendo su espacio de datos global. Para sincronizar, provee «candados» simples (también llamados *mutexes* o *binary semaphores*). El módulo *threading* provee una API de manejo de hilos más fácil de usar y de más alto nivel, construida sobre este módulo.

Distinto en la versión 3.7: Este módulo solía ser opcional, pero ahora está siempre disponible.

Este módulo define las siguientes constantes y funciones:

exception `_thread.error`

Lanzado ante errores específicos de un hilo.

Distinto en la versión 3.3: Ahora es un sinónimo de la excepción incorporada *RuntimeError*.

`_thread.LockType`

Este es el tipo de los objetos candado (*lock objects*).

`_thread.start_new_thread` (*function*, *args* [, *kwargs*])

Inicia un nuevo hilo y retorna su identificador. El hilo ejecuta la función *function* con la lista de argumentos *args* (que debe ser una tupla). El argumento opcional *kwargs* especifica un diccionario de argumentos por palabras clave.

Cuando la función retorna, el hilo finaliza silenciosamente.

Cuando la función termina con una excepción no gestionada, se invoca a *sys.unraisablehook()* para que gestione la excepción. El atributo *object* del argumento gancho (*hook*), es *function*. Por defecto, se muestra un seguimiento de pila y luego el hilo sale (pero los otros hilos continúan funcionando).

Cuando la función lanza una excepción *SystemExit*, se ignora silenciosamente.

Distinto en la versión 3.8: Ahora se utiliza *sys.unraisablehook()* para gestionar las excepciones no gestionadas.

`_thread.interrupt_main()`

Simular el efecto de una señal *signal.SIGINT* que llega al hilo principal. Un hilo puede usar esta función para interrumpir el hilo principal.

Si *signal.SIGINT* no está gestionada por Python (se definió *signal.SIG_DFL* o *signal.SIG_IGN*), esta función no hace nada.

`_thread.exit()`

Lanza la excepción `SystemExit`. Cuando no es gestionada, causa que el hilo salga silenciosamente.

`_thread.allocate_lock()`

Retorna un nuevo objeto candado (*lock object*). Los métodos de los candados se describen más abajo. El candado está abierto al inicio.

`_thread.get_ident()`

Retorna el “identificador de hilo” (*thread identifier*) del hilo actual. Es un entero distinto de cero. Su valor no tiene un significado directo, tiene la intención de ser utilizada como una *cookie* mágica para, por ejemplo, indexar un diccionario con datos específicos del hilo. Los identificadores de hilo pueden reciclarse cuando un hilo sale y otro se crea.

`_thread.get_native_id()`

Retorna el ID de hilo nativo integral del hilo asignado por el kernel. Es un entero no-negativo. Su valor puede utilizarse para identificar inequívocamente este hilo en particular en todo el sistema (hasta que el hilo termine, luego de lo cual el valor puede ser reciclado por el Sistema Operativo).

Disponibilidad: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX.

Nuevo en la versión 3.8.

`_thread.stack_size([size])`

Retorna el tamaño de la pila del hilo (*thread stack*) utilizada al crear nuevos hilos. El argumento opcional *size* especifica el tamaño de la pila a utilizar en los hilos que se creen a continuación, y debe ser 0 (utiliza el valor por defecto de la plataforma o el configurado) o un entero positivo de al menos 32768 (32KiB). Si *size* no se especifica, se utiliza 0. Si no está soportado el cambio del tamaño de pila del hilo, se lanza una excepción `RuntimeError`. Si la pila especificada es inválida se lanza un `ValueError` y el tamaño de la pila no se modifica. 32KiB es actualmente el menor valor soportado para el tamaño de la pila, para garantizar suficiente espacio en la misma para que quepa el propio intérprete. Tenga en cuenta que alguna plataformas pueden tener restricciones particulares en los valores para el tamaño de la pila, como requerir un mínimo que supere los 32KiB, o requerir una asignación en múltiplos del tamaño de página de memoria del sistema. Es necesario consultar la documentación de la plataforma para mayor información (son habituales las páginas de 4KiB; usar múltiplos de 4096 para el tamaño de pila es la estrategia sugerida si no se cuenta con información más específica).

Disponibilidad: Sistemas Windows, con hilos POSIX.

`_thread.TIMEOUT_MAX`

El máximo valor permitido para el parámetro *timeout* de `Lock.acquire()`. Especificar un tiempo de espera (*timeout*) mayor que este valor lanzará una excepción `OverflowError`.

Nuevo en la versión 3.2.

Los objetos candado (*lock objects*) tienen los siguientes métodos:

`lock.acquire(waitflag=1, timeout=-1)`

Sin ningún argumento opcional, este método adquiere el candado incondicionalmente, si es necesario esperando que éste sea liberado por otro hilo (solamente un hilo por vez puede adquirir un candado; para eso existen).

Si el argumento entero *waitflag* está presente, la acción depende de su valor: si es cero, el candado solamente es adquirido si está disponible de forma inmediata, sin esperas. Mientras que si es distinto de cero, el candado es adquirido sin condiciones, como en el caso anterior.

Si el argumento de punto flotante *timeout* está presente y es positivo, especifica el tiempo máximo de espera en segundos antes de retornar. Un argumento *timeout* negativo, especifica una espera ilimitada. No se puede especificar un *timeout* si *waitflag* es cero.

El valor de retorno es `True` si el candado (*lock*) se adquirió exitosamente, `False` de lo contrario.

Distinto en la versión 3.2: El parámetro *timeout* es nuevo.

Distinto en la versión 3.2: La adquisición de candados ahora puede ser interrumpida por señales en POSIX.

`lock.release()`

Libera el candado. El candado debe haber sido adquirido previamente, pero no necesariamente por el mismo hilo.

`lock.locked()`

Retorna el estado del candado: `True` si ha sido adquirido por algún hilo, `False` de lo contrario.

Además de estos métodos, los objetos candado pueden ser utilizados mediante la declaración `with`, por ejemplo:

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

Salvedades:

- Los hilos interactúan de manera extraña con interrupciones: la excepción `KeyboardInterrupt` va a ser recibida por un hilo cualquiera. (Cuando el módulo `signal` está disponible, la interrupción siempre se dirige al hilo principal.
- Invocar a `sys.exit()` o lanzar la excepción `SystemExit` equivale a invocar `_thread.exit()`.
- No es posible interrumpir el método `acquire()` en un candado. La excepción `KeyboardInterrupt` tendrá lugar después de que el candado haya sido adquirido.
- Cuando el hilo principal sale, ¿sobreviven los otros hilos? Depende de cómo esté definido por el sistema. En la mayoría de los sistemas, los hilos se cierran inmediatamente (*killed*), sin ejecutar las cláusulas `try ... finally` o los destructores del objeto.
- Cuando el hilo principal sale, no hace ninguna de las tareas de limpieza habituales (excepto que se haga honor a las cláusulas `try ... finally`), y los archivos de E/S estándar no son liberados.

17.11 `_dummy_thread`— Reemplazo directo para el módulo `_thread`

Código fuente: `Lib/_dummy_thread.py`

Obsoleto desde la versión 3.7: Ahora Python siempre tiene el hilado (`threading`) activado. Por favor, utilice en su lugar `_thread` (o, mejor, `threading`).

Este módulo proporciona una interfaz duplicada al módulo `_thread`. Estaba destinado a ser importado cuando el módulo `_thread` no fuera proporcionado en una plataforma.

Tenga cuidado de no usar este módulo donde pueda ocurrir un bloqueo mutuo debido a la creación de un hilo que bloquee la espera de crear otro hilo. Esto sucede a menudo con el bloqueo de E/S.

17.12 `dummy_threading` — Reemplazo directo para el modulo `threading`

Código fuente: `Lib/dummy_threading.py`

Obsoleto desde la versión 3.7: Ahora Python siempre tiene `threading` habilitado. Por favor utilice en su lugar `threading`.

Este módulo proporciona una interfaz duplicada al modulo `threading`. Estaba destinado a ser importado cuando el modulo `_thread` no fuera proporcionado por una plataforma.

Tenga cuidado de no usar este módulo donde pueda ocurrir un bloqueo mutuo debido a la creación de un hilo que bloquee la espera de crear otro hilo. Esto sucede a menudo con el bloqueo de E/S.

Comunicación en redes y entre procesos

Los módulos descritos en este capítulo proveen los mecanismos para la comunicación en red y entre procesos.

Algunos módulos solo funcionan para dos procesos que están en una misma máquina, e.g. *signal* y *mmap*. Otros módulos soportan protocolos de red que dos o mas procesos pueden utilizar para comunicarse entre máquinas.

La lista de módulos descritos en este capítulo es:

18.1 `asyncio` — E/S Asíncrona

¡Hola Mundo!

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

# Python 3.7+
asyncio.run(main())
```

`asyncio` es una biblioteca para escribir código **concurrente** utilizando la sintaxis **`async/await`**.

`asyncio` es utilizado como base en múltiples *frameworks* asíncronos de Python y provee un alto rendimiento en redes y servidores web, bibliotecas de conexión de base de datos, colas de tareas distribuidas, etc.

`asyncio` suele encajar perfectamente para operaciones con límite de E/S y código de red **estructurado** de alto nivel.

`asyncio` provee un conjunto de *APIs* de **alto nivel** para:

- *ejecutar corutinas de Python* de manera concurrente y tener control total sobre su ejecución;
- realizar *redes E/S y comunicación entre procesos (IPC)*;
- controlar *subprocesos*;
- distribuir tareas a través de *colas*;

- *sincronizar* código concurrente;

Adicionalmente, existen *APIs* de **bajo nivel** para *desarrolladores de bibliotecas y frameworks* para:

- crear y administrar *bucles de eventos*, los cuales proveen *APIs* asíncronas para *redes*, ejecutando *subprocesos*, gestionando *señales del sistema operativo*, etc;
- implementar protocolos eficientes utilizando *transportes*;
- Bibliotecas *puente* basadas en retrollamadas y código con sintaxis *async/wait*.

asyncio REPL

You can experiment with an `asyncio` concurrent context in the REPL:

```
$ python -m asyncio
asyncio REPL ...
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio
>>> await asyncio.sleep(10, result='hello')
'hello'
```

Raises an *auditing event* `cpython.run_stdin` with no arguments.

Distinto en la versión 3.8.20: Emits audit events.

Referencias

18.1.1 Corrutinas y Tareas

Esta sección describe las API de `asyncio` de alto nivel para trabajar con corrutinas y tareas.

- *Corrutinas*
- *Esperables*
- *Ejecutando un programa asyncio*
- *Creando Tareas*
- *Durmiendo*
- *Ejecutando Tareas Concurrentemente*
- *Protección contra Cancelación*
- *Tiempo agotado*
- *Esperando Primitivas*
- *Planificación Desde Otros Hilos*
- *Introspección*
- *Objeto Task*
- *Corrutinas basadas en generadores*

Corrutinas

Coroutines declarado con la sintaxis `async/await` es la forma preferida de escribir aplicaciones `asyncio`. Por ejemplo, el siguiente fragmento de código (requiere Python 3.7+) imprime «hola», espera 1 segundo y, a continuación, imprime «mundo»:

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')

>>> asyncio.run(main())
hello
world
```

Tenga en cuenta que simplemente llamando a una corrutina no programará para que se ejecute:

```
>>> main()
<coroutine object main at 0x1053bb7c8>
```

Para ejecutar realmente una corrutina, `asyncio` proporciona tres mecanismos principales:

- La función `asyncio.run()` para ejecutar la función de punto de entrada de nivel superior «`main()`» (consulte el ejemplo anterior.)
- Esperando en una corrutina. El siguiente fragmento de código imprimirá «hola» después de esperar 1 segundo y luego imprimirá «mundo» después de esperar *otros* 2 segundos:

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    await say_after(1, 'hello')
    await say_after(2, 'world')

    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

Salida esperada:

```
started at 17:13:52
hello
world
finished at 17:13:55
```

- La función `asyncio.create_task()` para ejecutar corrutinas concurrentemente como `asyncio Tasks`. Modifiquemos el ejemplo anterior y ejecutemos dos corrutinas `say_after` *concurrentemente*:

```
async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
```

(continué en la próxima página)

(proviene de la página anterior)

```
say_after(2, 'world'))

print(f"started at {time.strftime('%X')}")

# Wait until both tasks are completed (should take
# around 2 seconds.)
await task1
await task2

print(f"finished at {time.strftime('%X')}")
```

Tenga en cuenta que la salida esperada ahora muestra que el fragmento de código se ejecuta 1 segundo más rápido que antes:

```
started at 17:14:32
hello
world
finished at 17:14:34
```

Esperables

Decimos que un objeto es un objeto **esperable** si se puede utilizar en una expresión `await`. Muchas API de asyncio están diseñadas para aceptar los valores esperables.

Hay tres tipos principales de objetos *esperables*: **corrutinas**, **Tareas** y **Futuros**.

Corrutinas

Las corrutinas de Python son *esperables* y por lo tanto se pueden esperar de otras corrutinas:

```
import asyncio

async def nested():
    return 42

async def main():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited,
    # so it *won't run at all*.
    nested()

    # Let's do it differently now and await it:
    print(await nested())  # will print "42".

asyncio.run(main())
```

Importante: En esta documentación se puede utilizar el término «corrutina» para dos conceptos estrechamente relacionados:

- una *función corrutina*: una función `async def`;
 - un *objeto corrutina*: un objeto retornado llamando a una *función corrutina*.
-

asyncio también es compatible con corrutinas heredadas *generator-based*.

Tareas

Las *tareas* se utilizan para programar corrutinas *concurrentemente*.

Cuando una corrutina se envuelve en una *Tarea* con funciones como `asyncio.create_task()` la corrutina se programa automáticamente para ejecutarse pronto:

```
import asyncio

async def nested():
    return 42

async def main():
    # Schedule nested() to run soon concurrently
    # with "main()".
    task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete:
    await task

asyncio.run(main())
```

Futuros

Un *Future* es un objeto esperable especial de **bajo-nivel** que representa un **resultado eventual** de una operación asíncrona.

Cuando un objeto Future es *esperado* significa que la corrutina esperará hasta que el Future se resuelva en algún otro lugar.

Los objetos Future de asyncio son necesarios para permitir que el código basado en retro llamada se use con `async/await`.

Normalmente, **no es necesario** crear objetos Future en el código de nivel de aplicación.

Los objetos Future, a veces expuestos por bibliotecas y algunas API de asyncio, pueden ser esperados:

```
async def main():
    await function_that_returns_a_future_object()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

Un buen ejemplo de una función de bajo nivel que retorna un objeto Future es `loop.run_in_executor()`.

Ejecutando un programa asyncio

`asyncio.run(coro, *, debug=False)`

Ejecuta la *coroutine* `coro` y retornando el resultado.

Esta función ejecuta la corrutina pasada, encargándose de administrar el bucle de eventos asyncio y *finalizar los generadores asíncronos*.

Esta función no se puede llamar cuando otro bucle de eventos asyncio se está ejecutando en el mismo hilo.

Si `debug` es `True`, el bucle de eventos se ejecutará en modo debug.

Esta función siempre crea un nuevo ciclo de eventos y lo cierra al final. Debe usarse como un punto de entrada principal para los programas asyncio, e idealmente solo debe llamarse una vez.

Ejemplo:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

asyncio.run(main())
```

Nuevo en la versión 3.7.

Nota: El código fuente para `asyncio.run()` se puede encontrar en [Lib/asyncio/runners.py](#).

Creando Tareas

`asyncio.create_task(coro, *, name=None)`

Envuelve una *coroutine* `coro` en una *Task* y programa su ejecución. Retorna el objeto Tarea.

Si `name` no es `None`, se establece como el nombre de la tarea mediante `Task.set_name()`.

La tarea se ejecuta en el bucle retornado por `get_running_loop()`, `RuntimeError` se genera si no hay ningún bucle en ejecución en el subproceso actual.

Esta función se ha **añadido en Python 3.7**. Antes de Python 3.7, la función de bajo nivel `asyncio.ensure_future()` se puede utilizar en su lugar:

```
async def coro():
    ...

# In Python 3.7+
task = asyncio.create_task(coro())
...

# This works in all Python versions but is less readable
task = asyncio.ensure_future(coro())
...
```

Nuevo en la versión 3.7.

Distinto en la versión 3.8: Se ha añadido el parámetro `name`.

Durmiendo

coroutine `asyncio.sleep(delay, result=None, *, loop=None)`

Bloquea por `delay` segundos.

Si se proporciona `result`, se retorna al autor de la llamada cuando se completa la corrutina.

`sleep()` siempre suspende la tarea actual, permitiendo que se ejecuten otras tareas.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro `loop`. Ejemplo de una rutina que muestra la fecha actual cada segundo durante 5 segundos:

```
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
```

(continúe en la próxima página)

(proviene de la página anterior)

```

if (loop.time() + 1.0) >= end_time:
    break
await asyncio.sleep(1)

asyncio.run(display_date())

```

Ejecutando Tareas Concurrentemente

awaitable `asyncio.gather(*aws, loop=None, return_exceptions=False)`

Ejecute *objetos esperables* en la secuencia *aws* de forma *concurrently*.

Si cualquier esperable en *aws* es una corrutina, se programa automáticamente como una Tarea.

Si todos los esperables se completan correctamente, el resultado es una lista agregada de valores retornados. El orden de los valores de resultado corresponde al orden de esperables en *aws*.

Si *return_exceptions* es `False` (valor predeterminado), la primera excepción provocada se propaga inmediatamente a la tarea que espera en `gather()`. Otros esperables en la secuencia *aws* **no se cancelarán** y continuarán ejecutándose.

Si *return_exceptions* es `True`, las excepciones se tratan igual que los resultados correctos y se agregan en la lista de resultados.

Si `gather()` es *cancelado*, todos los esperables enviados (que aún no se han completado) también se *cancelan*.

Si alguna Tarea o Futuro de la secuencia *aws* se *cancela*, se trata como si se lanzara `CancelledError` – la llamada `gather()` **no** se cancela en este caso. Esto es para evitar la cancelación de una Tarea/Futuro enviada para hacer que otras Tareas/Futuros sean canceladas.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro *loop*. Ejemplo:

```

import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({i})...")
        await asyncio.sleep(1)
    f *= i
    print(f"Task {name}: factorial({number}) = {f}")

async def main():
    # Schedule three calls *concurrently*:
    await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )

asyncio.run(main())

# Expected output:
#
# Task A: Compute factorial(2)...
# Task B: Compute factorial(2)...
# Task C: Compute factorial(2)...
# Task A: factorial(2) = 2
# Task B: Compute factorial(3)...
# Task C: Compute factorial(3)...
# Task B: factorial(3) = 6

```

(continúe en la próxima página)

(proviene de la página anterior)

```
# Task C: Compute factorial(4)...
# Task C: factorial(4) = 24
```

Nota: If `return_exceptions` is `False`, cancelling `gather()` after it has been marked done won't cancel any submitted awaitables. For instance, `gather` can be marked done after propagating an exception to the caller, therefore, calling `gather.cancel()` after catching an exception (raised by one of the awaitables) from `gather` won't cancel any other awaitables.

Distinto en la versión 3.7: Si se cancela el propio `gather`, la cancelación se propaga independientemente de `return_exceptions`.

Protección contra Cancelación

awaitable `asyncio.shield(aw, *, loop=None)`

Protege un *objeto esperable* de ser *cancelado*.

Si *aw* es una corrutina, se programa automáticamente como una Tarea.

La declaración:

```
res = await shield(something())
```

es equivalente a:

```
res = await something()
```

excepto que si la corrutina que lo contiene se cancela, la tarea que se ejecuta en `something()` no se cancela. Desde el punto de vista de `something()`, la cancelación no ocurrió. Aunque su invocador siga cancelado, por lo que la expresión «await» sigue generando un `CancelledError`.

Si `something()` se cancela por otros medios (es decir, desde dentro de sí mismo) eso también cancelaría `shield()`.

Si se desea ignorar por completo la cancelación (no se recomienda) la función `shield()` debe combinarse con una cláusula `try/except`, como se indica a continuación:

```
try:
    res = await shield(something())
except CancelledError:
    res = None
```

Deprecated since version 3.8, will be removed in version 3.10: El parámetro *loop*.

Tiempo agotado

coroutine `asyncio.wait_for(aw, timeout, *, loop=None)`

Espere a que el *aw* *esperable* se complete con un tiempo agotado.

Si *aw* es una corrutina, se programa automáticamente como una Tarea.

timeout puede ser `None` o punto flotante o un número entero de segundos a esperar. Si *timeout* es `None`, se bloquea hasta que futuro se complete.

Si se produce un agotamiento de tiempo, cancela la tarea y genera `asyncio.TimeoutError`.

Para evitar la *cancelación* de la tarea, envuélvala en `shield()`.

La función esperará hasta que el futuro este cancelado, por lo que el tiempo de espera total puede superar el *timeout*.

Si se cancela la espera, el futuro `aw` también se cancela.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro `loop`. Ejemplo:

```
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except asyncio.TimeoutError:
        print('timeout!')

asyncio.run(main())

# Expected output:
#
#     timeout!
```

Distinto en la versión 3.7: Cuando `aw` se cancela debido a un agotamiento de tiempo, `wait_for` espera a que se cancele `aw`. Anteriormente, se lanzó inmediatamente `asyncio.TimeoutError`.

Esperando Primitivas

coroutine `asyncio.wait(aws, *, loop=None, timeout=None, return_when=ALL_COMPLETED)`

Run *awaitable objects* in the `aws` iterable concurrently and block until the condition specified by `return_when`.

Retorna dos conjuntos de Tareas/Futuros: (`done`, `pending`).

Uso:

```
done, pending = await asyncio.wait(aws)
```

`timeout` (un punto flotante o int), si se especifica, se puede utilizar para controlar el número máximo de segundos que hay que esperar antes de retornar.

Tenga en cuenta que esta función no lanza `asyncio.TimeoutError`. Los Futuros o Tareas que no terminan cuando se agota el tiempo simplemente se retornan en el segundo conjunto.

`return_when` indica cuándo debe retornar esta función. Debe ser una de las siguientes constantes:

Constante	Descripción
<code>FIRST_COMPLETED</code>	La función retornará cuando cualquier Futuro termine o se cancele.
<code>FIRST_EXCEPTION</code>	La función retornará cuando cualquier Futuro finalice lanzando una excepción. Si ningún Futuro lanza una excepción, entonces es equivalente a <code>ALL_COMPLETED</code> .
<code>ALL_COMPLETED</code>	La función retornará cuando todos los Futuros terminen o se cancelen.

A diferencia de `wait_for()`, `wait()` no cancela los Futuros cuando se produce un agotamiento de tiempo.

Obsoleto desde la versión 3.8: Si cualquier aguardable en `aws` es una corrutina, se programa automáticamente como una Tarea. El paso de objetos corrutinas a `wait()` directamente está en desuso, ya que conduce a *comportamiento confuso*.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro `loop`.

Nota: `wait()` programa las corrutinas como Tareas automáticamente y posteriormente retorna los objetos Tarea creados implícitamente en conjuntos (`done`, `pending`). Por lo tanto, el código siguiente no funcionará como se esperaba:

```
async def foo():
    return 42

coro = foo()
done, pending = await asyncio.wait({coro})

if coro in done:
    # This branch will never be run!
```

Aquí es cómo se puede arreglar el fragmento de código anterior:

```
async def foo():
    return 42

task = asyncio.create_task(foo())
done, pending = await asyncio.wait({task})

if task in done:
    # Everything will work as expected now.
```

Obsoleto desde la versión 3.8: El paso de objetos corrutina a `wait()` directamente está en desuso.

`asyncio.as_completed(aws, *, loop=None, timeout=None)`

Run *awaitable objects* in the *aws* iterable concurrently. Return an iterator of coroutines. Each coroutine returned can be awaited to get the earliest next result from the iterable of the remaining awaitables.

Lanza `asyncio.TimeoutError` si el agotamiento de tiempo ocurre antes que todos los Futuros terminen.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro *loop*.

Ejemplo:

```
for coro in as_completed(aws):
    earliest_result = await coro
    # ...
```

Planificación Desde Otros Hilos

`asyncio.run_coroutine_threadsafe(coro, loop)`

Envía una corrutina al bucle de eventos especificado. Seguro para Hilos.

Retorna `concurrent.futures.Future` para esperar el resultado de otro hilo del SO (Sistema Operativo).

Esta función está pensada para llamarse desde un hilo del SO diferente al que se ejecuta el bucle de eventos.

Ejemplo:

```
# Create a coroutine
coro = asyncio.sleep(1, result=3)

# Submit the coroutine to a given loop
future = asyncio.run_coroutine_threadsafe(coro, loop)

# Wait for the result with an optional timeout argument
assert future.result(timeout) == 3
```

Si se lanza una excepción en la corrutina, el Futuro retornado será notificado. También se puede utilizar para cancelar la tarea en el bucle de eventos:


```

try:
    result = future.result(timeout)
except asyncio.TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
except Exception as exc:
    print(f'The coroutine raised an exception: {exc!r}')
else:
    print(f'The coroutine returned: {result!r}')

```

Consulte la sección de la documentación *Concurrencia y multi hilos*.

A diferencia de otras funciones `asyncio`, esta función requiere que el argumento `loop` se pase explícitamente.

Nuevo en la versión 3.5.1.

Introspección

`asyncio.current_task(loop=None)`

Retorna la instancia `Task` actualmente en ejecución o `None` si no se está ejecutando ninguna tarea.

Si `loop` es `None` `get_running_loop()` se utiliza para obtener el bucle actual.

Nuevo en la versión 3.7.

`asyncio.all_tasks(loop=None)`

Retorna un conjunto de objetos `Task` que se ejecutan por el bucle.

Si `loop` es `None`, `get_running_loop()` se utiliza para obtener el bucle actual.

Nuevo en la versión 3.7.

Objeto Task

class `asyncio.Task` (*coro*, *, *loop=None*, *name=None*)

Un objeto *similar a Futuro* que ejecuta Python *coroutine*. No es seguro hilos.

Las tareas se utilizan para ejecutar corrutinas en bucles de eventos. Si una corrutina aguarda en un Futuro, la Tarea suspende la ejecución de la corrutina y espera la finalización del Futuro. Cuando el Futuro *termina*, se reanuda la ejecución de la corrutina envuelta.

Los bucles de eventos usan la programación cooperativa: un bucle de eventos ejecuta una tarea a la vez. Mientras una Tarea espera para la finalización de un Futuro, el bucle de eventos ejecuta otras tareas, retorno de llamada o realiza operaciones de E/S.

Utilice la función de alto nivel `asyncio.create_task()` para crear Tareas, o las funciones de bajo nivel `loop.create_task()` o `ensure_future()`. Se desaconseja la creación de instancias manuales de Tareas.

Para cancelar una Tarea en ejecución, utilice el método `cancel()`. Llamarlo hará que la tarea lance una excepción `CancelledError` en la corrutina contenida. Si una corrutina está esperando en un objeto Futuro durante la cancelación, se cancelará el objeto Futuro.

`cancelled()` se puede utilizar para comprobar si la Tarea fue cancelada. El método devuelve `True` si la corrutina contenida no suprimió la excepción `CancelledError` y se canceló realmente.

`asyncio.Task` hereda de `Future` todas sus API excepto `Future.set_result()` y `Future.set_exception()`.

Las tareas admiten el módulo `contextvars`. Cuando se crea una Tarea, copia el contexto actual y, posteriormente, ejecuta su corrutina en el contexto copiado.

Distinto en la versión 3.7: Agregado soporte para el módulo `contextvars`.

Distinto en la versión 3.8: Se ha añadido el parámetro `name`.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro *loop*.

cancel()

Solicita que se cancele la Tarea.

Esto hace que una excepción `CancelledError` sea lanzada a la corrutina contenida en el próximo ciclo del bucle de eventos.

La corrutina entonces tiene la oportunidad de limpiar o incluso denegar la solicitud suprimiendo la excepción con un bloque `try ... except CancelledError ... finally`. Por lo tanto, a diferencia de `Future.cancel()`, `Task.cancel()` no garantiza que la tarea será cancelada, aunque suprimir la cancelación por completo no es común y se desalienta activamente.

En el ejemplo siguiente se muestra cómo las corrutinas pueden interceptar la solicitud de cancelación:

```
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
        # Wait for 1 hour
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
        raise
    finally:
        print('cancel_me(): after sleep')

async def main():
    # Create a "cancel_me" Task
    task = asyncio.create_task(cancel_me())

    # Wait for 1 second
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

asyncio.run(main())

# Expected output:
#
#   cancel_me(): before sleep
#   cancel_me(): cancel sleep
#   cancel_me(): after sleep
#   main(): cancel_me is cancelled now
```

cancelled()

Retorna True si la Tarea se *cancela*.

La tarea se *cancela* cuando se solicitó la cancelación con `cancel()` y la corrutina contenida propagó la excepción `CancelledError` que se le ha lanzado.

done()

Retorna True si la Tarea está *finalizada*.

Una tarea está *finalizada* cuando la corrutina contenida retornó un valor, lanzó una excepción, o se canceló la Tarea.

result()

Retorna el resultado de la Tarea.

Si la tarea está *terminada*, se devuelve el resultado de la corrutina contenida (o si la corrutina lanzó una excepción, esa excepción se vuelve a relanzar.)

Si la Tarea ha sido *cancelada*, este método lanza una excepción `CancelledError`.

Si el resultado de la Tarea aún no está disponible, este método lanza una excepción `InvalidStateError`.

exception()

Retorna la excepción de la Tarea.

Si la corrutina contenida lanzó una excepción, esa excepción es retornada. Si la corrutina contenida retorna normalmente, este método retorna `None`.

Si la Tarea ha sido *cancelada*, este método lanza una excepción `CancelledError`.

Si la Tarea aún no está *terminada*, este método lanza una excepción `InvalidStateError`.

add_done_callback(callback, *, context=None)

Agrega una retro llamada que se ejecutará cuando la Tarea esté *terminada*.

Este método solo se debe usar en código basado en retrollamada de bajo nivel.

Consulte la documentación de `Future.add_done_callback()` para obtener más detalles.

remove_done_callback(callback)

Remueve la *retrollamada* de la lista de retrollamadas.

Este método solo se debe usar en código basado en retrollamada de bajo nivel.

Consulte la documentación de `Future.remove_done_callback()` para obtener más detalles.

get_stack(*, limit=None)

Retorna la lista de marcos de pila para esta tarea.

Si la corrutina contenida no se termina, esto retorna la pila donde se suspende. Si la corrutina se ha completado correctamente o se ha cancelado, retorna una lista vacía. Si la corrutina terminó por una excepción, esto retorna la lista de marcos de seguimiento.

Los marcos siempre se ordenan de más antiguo a más nuevo.

Solo se retorna un marco de pila para una corrutina suspendida.

El argumento opcional *limit* establece el número máximo de marcos que se retornarán; de forma predefinida se retornan todos los marcos disponibles. El orden de la lista devuelta varía en función de si se retorna una pila o un *traceback*: se devuelven los marcos más recientes de una pila, pero se devuelven los marcos más antiguos de un *traceback*. (Esto coincide con el comportamiento del módulo `traceback`.)

print_stack(*, limit=None, file=None)

Imprime la pila o el seguimiento de esta tarea.

Esto produce una salida similar a la del módulo `traceback` para los marcos recuperados por `get_stack()`.

El argumento *limit* se pasa directamente a `get_stack()`.

El argumento *file* es un flujo de E/S en el que se escribe la salida; por defecto, la salida se escribe en `sys.stderr`.

get_coro()

Retorna el objeto corrutina contenido por `Task`.

Nuevo en la versión 3.8.

get_name()

Retorna el nombre de la Tarea.

Si no se ha asignado explícitamente ningún nombre a la Tarea, la implementación de Tarea `asyncio` predeterminada genera un nombre predeterminado durante la creación de instancias.

Nuevo en la versión 3.8.

set_name (*value*)

Establece el nombre de la Tarea.

El argumento *value* puede ser cualquier objeto, que luego se convierte en una cadena.

En la implementación de Task predeterminada, el nombre será visible en la salida `repr()` de un objeto de tarea.

Nuevo en la versión 3.8.

classmethod all_tasks (*loop=None*)

Retorna un conjunto de todas las tareas para un bucle de eventos.

De forma predeterminada, se retornan todas las tareas del bucle de eventos actual. Si *loop* es `None`, la función `get_event_loop()` se utiliza para obtener el bucle actual.

Deprecated since version 3.7, will be removed in version 3.9: No llame a esto como un método de tarea. Utilice la función `asyncio.all_tasks()` en su lugar.

classmethod current_task (*loop=None*)

Retorna la tarea que se está ejecutando actualmente o `None`.

Si *bucle* es `None`, la función `get_event_loop()` se utiliza para obtener el bucle actual.

Deprecated since version 3.7, will be removed in version 3.9: No llame a esto como un método de tarea. Utilice la función `asyncio.current_task()` en su lugar.

Corrutinas basadas en generadores

Nota: La compatibilidad con corrutinas basadas en generadores está **en desuso** y está programada para su eliminación en Python 3.10.

Las corrutinas basadas en generadores son anteriores a la sintaxis `async/await`. Son generadores de Python que utilizan expresiones `yield from` para esperar en Futuros y otras corrutinas.

Las corrutinas basadas en generadores deben estar decoradas con `@asyncio.coroutine`, aunque esto no se aplica.

@asyncio.coroutine

Decorador para marcar corrutinas basadas en generadores.

Este decorador permite que las corrutinas basadas en generadores de versiones anteriores (*legacy*) sean compatibles con el código `async/await`:

```
@asyncio.coroutine
def old_style_coroutine():
    yield from asyncio.sleep(1)

async def main():
    await old_style_coroutine()
```

Este decorador no debe utilizarse para corrutinas `async def`.

Deprecated since version 3.8, will be removed in version 3.10: Usar `async def` en su lugar.

asyncio.iscoroutine (*obj*)

Retorna `True` si *obj* es un *coroutine object*.

Este método es diferente de `inspect.iscoroutine()` porque retorna `True` para corrutinas basadas en generadores.

asyncio.iscoroutinefunction (*func*)

Retorna `True` si *func* es una *coroutine function*.

Este método es diferente de `inspect.iscoroutinefunction()` porque retorna `True` para funciones de corrutinas basadas en generadores decoradas con `@coroutine`.

18.1.2 Streams

Código fuente: `Lib/asyncio/streams.py`

Los *streams* son `async/await` primitivos de alto nivel para trabajar con conexiones de red. Los *streams* permiten enviar y recibir datos sin utilizar *callbacks* o protocolos y transportes de bajo nivel.

Aquí hay un ejemplo de un cliente eco TCP escrito utilizando *streams* `asyncio`:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

Consulte también la sección de [Examples](#) a continuación.

Funciones *stream*

Las siguientes funciones `asyncio` de nivel superior se pueden utilizar para crear y trabajar con *streams*:

coroutine `asyncio.open_connection` (*host=None, port=None, *, loop=None, limit=None, ssl=None, family=0, proto=0, flags=0, sock=None, local_addr=None, server_hostname=None, ssl_handshake_timeout=None*)

Establece una conexión de red y retorna un par de objetos (*reader, writer*).

Los objetos retornados *reader* y *writer* son instancias de las clases `StreamReader` y `StreamWriter`.

El argumento *loop* es opcional y siempre se puede determinar automáticamente cuando se espera esta función de una corrutina.

limit determina el límite de tamaño del búfer utilizado por la instancia de `StreamReader` retornada. De forma predeterminada, *limit* está establecido en 64 KiB.

El resto de los argumentos se pasan directamente a `loop.create_connection()`.

Nuevo en la versión 3.7: El parámetro *ssl_handshake_timeout*.

coroutine `asyncio.start_server` (*client_connected_cb, host=None, port=None, *, loop=None, limit=None, family=socket.AF_UNSPEC, flags=socket.AI_PASSIVE, sock=None, backlog=100, ssl=None, reuse_address=None, reuse_port=None, ssl_handshake_timeout=None, start_serving=True*)

Inicia un servidor socket.

La retrollamada `client_connected_cb` se llama siempre que se establece una nueva conexión de cliente. Recibe un par (`reader`, `writer`) como dos argumentos, instancias de las clases `StreamReader` y `StreamWriter`.

`client_connected_cb` puede ser una función simple invocable o de *corrutina*; si es una función de corrutina, se programará automáticamente como un *Task*.

El argumento `loop` es opcional y siempre se puede determinar automáticamente cuando se espera este método de una corrutina.

`limit` determina el límite de tamaño del búfer utilizado por la instancia de `StreamReader` retornada. De forma predeterminada, `limit` está establecido en 64 KiB.

El resto de los argumentos se pasan directamente a `loop.create_server()`.

Nuevo en la versión 3.7: Los parámetros `ssl_handshake_timeout` y `start_serving`.

Sockets Unix

coroutine `asyncio.open_unix_connection` (`path=None`, *, `loop=None`, `limit=None`,
`ssl=None`, `sock=None`, `server_hostname=None`,
`ssl_handshake_timeout=None`)

Establece una conexión de socket Unix y retorna un par de (`reader`, `writer`).

Similar a `open_connection()` pero opera en sockets Unix.

Consulte también la documentación de `loop.create_unix_connection()`.

Disponibilidad: Unix.

Nuevo en la versión 3.7: El parámetro `ssl_handshake_timeout`.

Distinto en la versión 3.7: El parámetro `path` ahora puede ser un objeto similar a una ruta (*path-like object*)

coroutine `asyncio.start_unix_server` (`client_connected_cb`, `path=None`, *, `loop=None`,
`limit=None`, `sock=None`, `backlog=100`, `ssl=None`,
`ssl_handshake_timeout=None`, `start_serving=True`)

Inicia un servidor socket Unix.

Similar a `start_server()` pero funciona con sockets Unix.

Consulte también la documentación de `loop.create_unix_server()`.

Disponibilidad: Unix.

Nuevo en la versión 3.7: Los parámetros `ssl_handshake_timeout` y `start_serving`.

Distinto en la versión 3.7: El parámetro `path` ahora puede ser un objeto similar a una ruta (*path-like object*).

StreamReader

class `asyncio.StreamReader`

Representa un objeto lector que proporciona APIs para leer datos del flujo de E/S.

No se recomienda crear instancias de objetos `StreamReader` directamente; utilice `open_connection()` y `start_server()` en su lugar.

coroutine `read` (`n=-1`)

Lee hasta `n` bytes. Si no se proporciona `n`, o se establece en `-1`, lee hasta EOF (final del archivo) y retorna todos los bytes leídos.

Si se recibió EOF (final del archivo) y el búfer interno está vacío, retorna un objeto de `bytes` vacío.

coroutine `readline` ()

Lea una línea, donde «línea» es una secuencia de bytes que termina en `\n`.

Si se recibe EOF (final del archivo) y no se encontró `\n`, el método retorna datos leídos parcialmente.

Si se recibe EOF (final de archivo) y el búfer interno está vacío, retorna un objeto de `bytes` vacío.

coroutine readexactly(*n*)

Lee exactamente *n* bytes.

Genera un `IncompleteReadError` si se alcanza EOF (final del archivo) antes de que se pueda leer *n*. Utilice el atributo `IncompleteReadError.partial` para obtener los datos leídos parcialmente.

coroutine readuntil(*separator=b'\n'*)

Lee datos de la secuencia hasta que se encuentre el separador (*separator*).

En caso de éxito, los datos y el separador se eliminarán del búfer interno (consumido). Los datos retornados incluirán el separador al final.

Si la cantidad de datos leídos excede el límite de flujo configurado, se genera una excepción `LimitOverrunError` y los datos se dejan en el búfer interno y se pueden leer nuevamente.

Si se alcanza EOF (final del archivo) antes de que se encuentre el separador completo, se genera una excepción `IncompleteReadError` y se restablece el búfer interno. El atributo `IncompleteReadError.partial` puede contener una parte del separador.

Nuevo en la versión 3.5.2.

at_eof()

Retorna `True` si el buffer está vacío y `feed_eof()` fue llamado.

StreamWriter

class asyncio.StreamWriter

Representa un objeto de escritura que proporciona APIs para escribir datos en el flujo de E/S.

No se recomienda crear instancias de objetos `StreamWriter` directamente; use `open_connection()` y `start_server()` en su lugar.

write(*data*)

El método intenta escribir los datos (*data*) en el socket subyacente inmediatamente. Si eso falla, los datos se ponen en cola en un búfer de escritura interno hasta que se puedan enviar.

El método debe usarse junto con el método `drain()`:

```
stream.write(data)
await stream.drain()
```

writelines(*data*)

El método escribe una lista (o cualquier iterable) de bytes en el socket subyacente inmediatamente. Si eso falla, los datos se ponen en cola en un búfer de escritura interno hasta que se puedan enviar.

El método debe usarse junto con el método `drain()`:

```
stream.writelines(lines)
await stream.drain()
```

close()

El método cierra la secuencia y el socket subyacente.

El método debe usarse junto con el método `wait_closed()`:

```
stream.close()
await stream.wait_closed()
```

can_write_eof()

Retorna `True` si el transporte subyacente admite el método `write_eof()`, `False` en caso contrario.

write_eof()

Cierra la escritura de la secuencia después de que se vacíen los datos de escritura almacenados en búfer.

transport

Retorna el transporte asyncio subyacente.

get_extra_info (*name*, *default=None*)

Accede a información de transporte opcional; consulte `BaseTransport.get_extra_info()` para obtener más detalles.

coroutine drain ()

Espera hasta que sea apropiado reanudar la escritura en la transmisión. Ejemplo:

```
writer.write(data)
await writer.drain()
```

Este es un método de control de flujo que interactúa con el búfer de escritura de E/S subyacente. Cuando el tamaño del búfer alcanza la marca de agua alta, `drain()` bloquea hasta que el tamaño del búfer se agota hasta la marca de agua baja y se pueda reanudar la escritura. Cuando no hay nada que esperar, `drain()` regresa inmediatamente.

is_closing ()

Retorna `True` si la secuencia está cerrada o en proceso de cerrarse.

Nuevo en la versión 3.7.

coroutine wait_closed ()

Espera hasta que se cierre la secuencia.

Debería llamarse después de `close()` para esperar hasta que se cierre la conexión subyacente.

Nuevo en la versión 3.7.

Ejemplos

Cliente eco TCP usando *streams*

Cliente eco TCP usando la función `asyncio.open_connection()`:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()

asyncio.run(tcp_echo_client('Hello World!'))
```

Ver también:

El ejemplo del *protocolo de cliente eco TCP* utiliza el método `loop.create_connection()` de bajo nivel.

Servidor eco TCP usando *streams*

Servidor eco TCP usando la función `asyncio.start_server()`:

```
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")

    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addr = server.sockets[0].getsockname()
    print(f'Serving on {addr}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

Ver también:

El ejemplo del *protocolo de servidor eco TCP* utiliza el método `loop.create_server()`.

Obtener encabezados HTTP

Ejemplo simple de consulta de encabezados HTTP de la URL pasada en la línea de comando:

```
import asyncio
import urllib.parse
import sys

async def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url.hostname, 80)

    query = (
        f"HEAD {url.path or '/'} HTTP/1.0\r\n"
        f"Host: {url.hostname}\r\n"
        f"\r\n"
    )

    writer.write(query.encode('latin-1'))
    while True:
```

(continué en la próxima página)

(proviene de la página anterior)

```
line = await reader.readline()
if not line:
    break

line = line.decode('latin1').rstrip()
if line:
    print(f'HTTP header> {line}')

# Ignore the body, close the socket
writer.close()

url = sys.argv[1]
asyncio.run(print_http_headers(url))
```

Uso:

```
python example.py http://example.com/path/page.html
```

o con HTTPS:

```
python example.py https://example.com/path/page.html
```

Registrar un socket abierto para esperar datos usando *streams*

Corutina esperando hasta que un socket reciba datos usando la función `open_connection()` function:

```
import asyncio
import socket

async def wait_for_data():
    # Get a reference to the current event loop because
    # we want to access low-level APIs.
    loop = asyncio.get_running_loop()

    # Create a pair of connected sockets.
    rsock, wsock = socket.socketpair()

    # Register the open socket to wait for data.
    reader, writer = await asyncio.open_connection(sock=rsock)

    # Simulate the reception of data from the network
    loop.call_soon(wsock.send, 'abc'.encode())

    # Wait for data
    data = await reader.read(100)

    # Got data, we are done: close the socket
    print("Received:", data.decode())
    writer.close()

    # Close the second socket
    wsock.close()

asyncio.run(wait_for_data())
```

Ver también:

El ejemplo de *registro de un socket abierto para esperar datos usando un protocolo* utiliza un protocolo de bajo nivel y el método `loop.create_connection()`.

El ejemplo de *observar un descriptor de archivo para leer eventos* utiliza el método `loop.add_reader()` de bajo nivel para ver un descriptor de archivo.

18.1.3 Primitivas de sincronización

Código fuente: [Lib/asyncio/locks.py](#)

Las primitivas de sincronización de `asyncio` están diseñadas para ser similares a las del módulo `threading`, con dos importantes advertencias:

- las primitivas de `asyncio` no son seguras en hilos, por lo tanto, no deben ser usadas para la sincronización de hilos del sistema operativo (usa `threading` para esto);
- los métodos de estas primitivas de sincronización no aceptan el argumento `timeout`. Usa la función `asyncio.wait_for()` para realizar operaciones que involucren tiempos de espera.

`asyncio` tiene las siguientes primitivas de sincronización básicas:

- `Lock`
- `Event`
- `Condition`
- `Semaphore`
- `BoundedSemaphore`

Lock

class `asyncio.Lock` (*, `loop=None`)

Implementa un cierre de exclusión mutua para tareas `asyncio`. No es seguro en hilos.

Un cierre `asyncio` puede usarse para garantizar el acceso en exclusiva a un recurso compartido.

La forma preferida de usar un `Lock` es mediante una declaración `async with`:

```
lock = asyncio.Lock()

# ... later
async with lock:
    # access shared state
```

lo que es equivalente a:

```
lock = asyncio.Lock()

# ... later
await lock.acquire()
try:
    # access shared state
finally:
    lock.release()
```

Deprecated since version 3.8, will be removed in version 3.10: El parámetro `loop`.

coroutine `acquire()`

Adquiere el cierre.

Este método espera hasta que el cierre está *abierto*, lo establece como *cerrado* y retorna `True`.

Cuando más de una corrutina está bloqueada en `acquire()`, esperando a que el cierre se abra, solo una de las corrutinas proseguirá finalmente.

Adquirir un cierre es *justo*: la corrutina que prosigue será la primera corrutina que comenzó a esperarlo.

release()

Libera el cierre.

Cuando el cierre está *cerrado*, lo restablece al estado *abierto* y retorna.

Si el cierre está *abierto*, se lanza una excepción `RuntimeError`.

locked()

Retorna `True` si el cierre está *cerrado*.

Event

class `asyncio.Event` (*, *loop=None*)

Un objeto de eventos. No es seguro en hilos.

Un evento `asyncio` puede usarse para notificar a múltiples tareas `asyncio` que ha ocurrido algún evento.

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 set to *true*. The flag is set to *false* initially.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro *loop*. Ejemplo:

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Create an Event object.
    event = asyncio.Event()

    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()

    # Wait until the waiter task is finished.
    await waiter_task

asyncio.run(main())
```

coroutine `wait()`

Espera hasta que se establezca el evento.

If the event is set, return `True` immediately. Otherwise block until another task calls `set()`.

set()

Establece el evento.

Todas las tareas esperando a que el evento se establezca serán activadas inmediatamente.

clear()

Borra (restablece) el evento.

Tasks awaiting on `wait()` will now block until the `set()` method is called again.

is_set()

Retorna `True` si el evento está establecido.

Condition

class `asyncio.Condition` (*lock=None*, *, *loop=None*)

Un objeto Condition. No seguro en hilos.

Una tarea puede usar una condición primitiva de asyncio para esperar a que suceda algún evento y luego obtener acceso exclusivo a un recurso compartido.

En esencia, un objeto Condition combina la funcionalidad de un objeto *Event* y un objeto *Lock*. Es posible tener múltiples objetos Condition que compartan un mismo Lock, lo que permite coordinar el acceso exclusivo a un recurso compartido entre diferentes tareas interesadas en estados particulares de ese recurso compartido.

El argumento opcional *lock* debe ser un objeto *Lock* o *None*. En este último caso, se crea automáticamente un nuevo objeto Lock.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro *loop*.

La forma preferida de usar una condición es mediante una declaración `async with`:

```
cond = asyncio.Condition()

# ... later
async with cond:
    await cond.wait()
```

lo que es equivalente a:

```
cond = asyncio.Condition()

# ... later
await cond.acquire()
try:
    await cond.wait()
finally:
    cond.release()
```

coroutine `acquire()`

Adquiere el cierre (lock) subyacente.

Este método espera hasta que el cierre subyacente esté *abierto*, lo establece en *cerrado* y retorna `True`.

notify (*n=1*)

Despierta como máximo *n* tareas (1 por defecto) que estén esperando a esta condición. El método no es operativo si no hay tareas esperando.

El cierre debe adquirirse antes de llamar a este método y liberarse poco después. Si se llama con un cierre *abierto*, se lanza una excepción *RuntimeError*.

locked ()

Retorna `True` si el cierre subyacente está adquirido.

notify_all ()

Despierta todas las tareas que esperan a esta condición.

Este método actúa como `notify()`, pero despierta todas las tareas en espera.

El cierre debe adquirirse antes de llamar a este método y liberarse poco después. Si se llama con un cierre *abierto*, se lanza una excepción *RuntimeError*.

release ()

Libera el cierre subyacente.

Cuando se invoca en un cierre abierto, se lanza una excepción *RuntimeError*.

coroutine `wait` ()

Espera hasta que se le notifique.

Si la tarea que llama no ha adquirido el cierre cuando se llama a este método, se lanza una excepción `RuntimeError`.

Este método libera el cierre subyacente y luego se bloquea hasta que lo despierte una llamada `notify()` o `notify_all()`. Una vez despertado, la condición vuelve a adquirir su cierre y este método retorna `True`.

coroutine `wait_for(predicate)`

Espera hasta que un predicado se vuelva *verdadero*.

El predicado debe ser un objeto invocable cuyo resultado se interpretará como un valor booleano. El valor final es el valor de retorno.

Semaphore

class `asyncio.Semaphore(value=1, *, loop=None)`

Un objeto Semaphore. No es seguro en hilos.

Un semáforo gestiona un contador interno que se reduce en cada llamada al método `acquire()` y se incrementa en cada llamada al método `release()`. El contador nunca puede bajar de cero, cuando `acquire()` encuentra que es cero, se bloquea, esperando hasta que alguna tarea llame a `release()`.

El argumento opcional *value* proporciona el valor inicial para el contador interno (1 por defecto). Si el valor dado es menor que 0 se lanza una excepción `ValueError`.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro *loop*.

La forma preferida de utilizar un semáforo es mediante una declaración `async with`:

```
sem = asyncio.Semaphore(10)

# ... later
async with sem:
    # work with shared resource
```

lo que es equivalente a:

```
sem = asyncio.Semaphore(10)

# ... later
await sem.acquire()
try:
    # work with shared resource
finally:
    sem.release()
```

coroutine `acquire()`

Adquiere un semáforo.

Si el contador interno es mayor que cero, lo reduce en uno y retorna `True` inmediatamente. Si es cero, espera hasta que se llame al método `release()` y retorna `True`.

locked()

Retorna `True` si el semáforo no se puede adquirir de inmediato.

release()

Libera un semáforo, incrementando el contador interno en uno. Puede despertar una tarea que esté a la espera para adquirir el semáforo.

A diferencia de `BoundedSemaphore`, `Semaphore` permite hacer más llamadas `release()` que llamadas `acquire()`.

BoundedSemaphore

class `asyncio.BoundedSemaphore` (*value=1, *, loop=None*)

Un objeto semáforo delimitado. No es seguro en hilos.

`BoundedSemaphore` es una versión de la clase `Semaphore` que lanza una excepción `ValueError` en `release()` si aumenta el contador interno por encima del *valor* inicial.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro *loop*.

Obsoleto desde la versión 3.7: Adquirir un bloqueo usando `await lock` o `yield from lock` y/o la declaración `with` (`with await lock, with (yield from lock)`) está obsoleto. En su lugar, se debe usar `async with lock`.

18.1.4 Sub-procesos

Código fuente: `Lib/asyncio/subprocess.py`, `Lib/asyncio/base_subprocess.py`

Esta sección describe APIs de alto nivel de *async/await* de `asyncio` para crear y gestionar sub-procesos.

Aquí podemos encontrar un ejemplo de cómo `asyncio` puede ejecutar un comando de *shell* y obtener su resultado:

```
import asyncio

async def run(cmd):
    proc = await asyncio.create_subprocess_shell(
        cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE)

    stdout, stderr = await proc.communicate()

    print(f'[{cmd!r} exited with {proc.returncode}]')
    if stdout:
        print(f'[stdout]\n{stdout.decode()}')
    if stderr:
        print(f'[stderr]\n{stderr.decode()}')

asyncio.run(run('ls /zzz'))
```

mostrará en pantalla:

```
[ 'ls /zzz' exited with 1]
[stderr]
ls: /zzz: No such file or directory
```

Ya que todas las funciones de sub-procesos de `asyncio` son asíncronas y `asyncio` proporciona herramientas para trabajar con tales funciones, es fácil ejecutar y monitorear múltiples subprocesos en paralelo. De hecho es trivial modificar los ejemplos de arriba para ejecutar varios comandos simultáneamente:

```
async def main():
    await asyncio.gather(
        run('ls /zzz'),
        run('sleep 1; echo "hello"'))

asyncio.run(main())
```

Véase también la subsección *Examples*.

Creando sub-procesos

coroutine `asyncio.create_subprocess_exec` (*program*, **args*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

Crea un sub-proceso.

El argumento *limit* establece el límite del buffer para los envoltorios `StreamReader` para `Process.stdout` y `Process.stderr` (si se pasa `subprocess.PIPE` a los argumentos *stdout* y *stderr*).

Retorna una instancia de `Process`.

Véase la documentación de `loop.subprocess_exec()` para los otros parámetros.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro *loop*.

coroutine `asyncio.create_subprocess_shell` (*cmd*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

Ejecuta el comando de shell *cmd*.

El argumento *limit* establece el límite del buffer para los envoltorios `StreamReader` para `Process.stdout` y `Process.stderr` (si se pasa `subprocess.PIPE` a los argumentos *stdout* y *stderr*).

Retorna una instancia de `Process`.

Véase la documentación de `loop.subprocess_shell()` para los otros parámetros.

Importante: Es la responsabilidad de la aplicación asegurarse que todos los espacios en blanco y caracteres especiales estén citados apropiadamente para evitar vulnerabilidades de *inyección de instrucciones*. La función `shlex.quote()` puede ser usada para escapar caracteres en blanco y caracteres especiales de *shell* en cadenas de caracteres a ser usadas para construir comandos de *shell*.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro *loop*.

Nota: Subprocesses are available for Windows if a `ProactorEventLoop` is used. See *Subprocess Support on Windows* for details.

Ver también:

`asyncio` también tiene las siguientes APIs de *bajo nivel* para trabajar con sub-procesos: `loop.subprocess_exec()`, `loop.subprocess_shell()`, `loop.connect_read_pipe()`, `loop.connect_write_pipe()`, así como también los *Sub-procesos de Transportes* y los *Sub-procesos de Protocolos*.

Constantes

`asyncio.subprocess.PIPE`

Se le puede pasar a los parámetros *stdin*, *stdout* o *stderr*.

Si se le pasa *PIPE* al argumento *stdin*, el atributo `Process.stdin` apuntará a la instancia `StreamWriter`.

Si se pasa *PIPE* a los argumentos *stdout* o *stderr*, los atributos `Process.stdout` y `Process.stderr` apuntarán a las instancias `StreamReader`.

`asyncio.subprocess.STDOUT`

Un valor especial que puede ser usado como el argumento de *stderr* e indica que los errores estándar deben ser redireccionados en la salida estándar.

`asyncio.subprocess.DEVNULL`

Un valor especial que puede ser usado como el argumento de *stdin*, *stdout*, *stderr* para procesar funciones de creación. Indica que el archivo especial `os.devnull` será usado para la correspondiente transmisión del sub-proceso.

Interactuando con Subprocesos

Las dos funciones `create_subprocess_exec()` y `create_subprocess_shell()` retornan instancias de la clase `Process`. `Process` es un envoltorio de alto nivel que permite comunicarse con los subprocessos y estar atento a sus términos.

class `asyncio.subprocess.Process`

Un objeto que envuelve procesos del SO creados por las funciones `create_subprocess_exec()` y `create_subprocess_shell()`.

Esta clase está diseñada para tener un API similar a la clase `subprocess.Popen`, pero hay algunas diferencias notables:

- a diferencia de `Popen`, las instancias de `Process` no tiene un equivalente del método `poll()`;
- los métodos `communicate()` y `wait()` no tienen un parámetro `timeout`: use la función `wait_for()`;
- el método `Process.wait()` es asíncrono, mientras que el método `subprocess.Popen.wait()` es implementado como un bucle de espera activa;
- el parámetro `universal_newlines` no es compatible.

Esta clase no es segura para subprocessos (*not thread safe*).

Véase también la sección *Subprocesos e Hilos*.

coroutine `wait()`

Espera a que el proceso hijo termine.

Define y retorna el atributo `returncode`.

Nota: Este método puede bloquearse cuando usa `stdout=PIPE` o `stderr=PIPE` y el proceso hijo genera tanta salida que se bloquea esperando a que la tubería de búfer del SO acepte más datos. Use el método `communicate()` cuando use tuberías para evitar esta condición.

coroutine `communicate(input=None)`

Interactuar con el proceso:

1. envía datos a `stdin` (si `input` no es `None`);
2. lee datos desde `stdout` y `stderr`, hasta que el llegue al EOF;
3. espera a que el proceso termine.

El argumento opcional `input` son los datos (objetos `bytes`) que serán enviados a los procesos hijos.

Retorna una tupla (`stdout_data`, `stderr_data`).

Si se levanta la excepción `BrokenPipeError` o `ConnectionResetError` cuando se escribe `input` en `stdin`, la excepción es ignorada. Esta condición ocurre cuando el proceso finaliza antes de que todos los datos sean escritos en `stdin`.

Si se desea enviar datos al `stdin` del proceso, el proceso necesita ser creado con `stdin=PIPE`. De manera similar, para obtener cualquier cosa excepto `None` en la tupla del resultado, el proceso tiene que ser creado con los argumentos `stdout=PIPE` y/o `stderr=PIPE`.

Tenga en cuenta que los datos leídos son almacenados en memoria, por lo que no utilice este método si el tamaño de los datos es más grande o ilimitado.

send_signal(signal)

Envíe la señal `signal` al proceso hijo.

Nota: En Windows, `SIGTERM` es un alias para `terminate()`. Se puede enviar `CTRL_C_EVENT` y `CTRL_BREAK_EVENT` a procesos iniciados con un parámetro `creationflags` que incluye

`CREATE_NEW_PROCESS_GROUP`.

terminate()

Para al proceso hijo.

En sistemas POSIX este método envía `signal.SIGTERM` al proceso hijo.

En Windows, la función de la API de Win32 `TerminateProcess()` es llamado para parar a los procesos hijos.

kill()

Kill the child process.

En sistemas POSIX, este método envía `SIGKILL` al proceso hijo.

En Windows este método es un alias para `terminate()`.

stdin

Flujo de entrada estándar (`StreamWriter`) o `None` si el proceso fue creado con `stdin=None`.

stdout

Flujo de salida estándar (`StreamReader`) o `None` si el proceso fue creado con `stdout=None`.

stderr

Flujo de salida estándar (`StreamReader`) o `None` si el proceso fue creado con `stderr=None`.

Advertencia: Utilice el método `communicate()` en vez de `process.stdin.write()`, `await process.stdout.read()` o `await process.stderr.read`. Esto evita bloqueos debido a que los flujos pausan la lectura o escritura y bloqueando al proceso hijo.

pid

Número de identificación del Proceso (PID por sus siglas en inglés).

Tenga en cuenta que para procesos creados por la función `create_subprocess_shell()`, este atributo es el PID del shell generado.

returncode

Código de retorno del proceso cuando finaliza.

Un valor `None` indica que el proceso todavía no ha finalizado.

Un valor negativo `-N` indica que el hijo ha finalizado por la señal `N` (sólo para POSIX).

Subprocesos y Hilos

Por defecto el bucle de eventos de `asyncio` estándar permite correr subprocesos desde hilos diferentes.

En Windows, sólo `ProactorEventLoop` proporciona subprocesos (por defecto), `SelectorEventLoop` no es compatible con subprocesos.

En UNIX, los *observadores de hijos* son usados para la espera de finalización de subprocesos, véase *Observadores de procesos* para más información.

Distinto en la versión 3.8: UNIX cambió para usar `ThreadedChildWatcher` para generar subprocesos de hilos diferentes sin ninguna limitación.

Crear un subproceso con el observador del hijo *inactivo* lanza un `RuntimeError`.

Tenga en cuenta que implementaciones alternativas del bucle de eventos pueden tener limitaciones propias; por favor consulte su documentación.

Ver también:

La sección *Concurrencia y multihilos en asyncio*.

Ejemplos

Un ejemplo usando la clase `Process` para controlar un subprocesso y la clase `StreamReader` para leer de su salida estándar.

El subprocesso es creado por la función `create_subprocess_exec()`:

```
import asyncio
import sys

async def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # Create the subprocess; redirect the standard output
    # into a pipe.
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # Read one line of output.
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit.
    await proc.wait()
    return line

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

Véase también los *ejemplos de prueba* escritos usando APIs de bajo nivel.

18.1.5 Colas

Código fuente: `Lib/asyncio/queue.py`

Las colas `asyncio` son diseñadas para ser similares a clases del módulo `queue`. Sin embargo las colas `asyncio` no son seguras para hilos, son diseñadas para usar específicamente en código `async/await`.

Nota que los métodos de colas de `asyncio` no tienen un parámetro *timeout*; usa la función `asyncio.wait_for()` para hacer operaciones de cola con un tiempo de espera.

Ver también la sección *Examples* a continuación.

Cola

class `asyncio.Queue` (*maxsize=0*, *, *loop=None*)

Una cola primero en entrar, primero en salir (PEPS, o *FIFO* en inglés).

Si *maxsize* es menor que o igual a cero, el tamaño de la cola es infinito. Si es un entero mayor a 0, entonces `await put()` se bloquea cuando una cola alcanza su *maxsize* hasta que un elemento es removido por `get()`.

Diferente de los subprocessos de la biblioteca estándar `queue`, el tamaño de la cola siempre es conocido y puede ser retornado llamando el método `qsize()`.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro *loop*.

Esta clase es *no segura para hilos*.

maxsize

Número de ítems permitidos en la cola.

empty()

Retorna `True` si la cola es vacía, o `False` en caso contrario.

full()

Retorna `True` si hay `maxsize` ítems en la cola.

Si la cola fue inicializada con `maxsize=0` (el predeterminado), entonces `full()` nunca retorna `True`.

coroutine get()

Remueve y retorna un ítem de la cola. Si la cola es vacía, espera hasta que un ítem esté disponible.

get_nowait()

Retorna un ítem si uno está inmediatamente disponible, de otra manera levanta `QueueEmpty`.

coroutine join()

Se bloquea hasta que todos los ítems en la cola han sido recibidos y procesados.

El conteo de tareas no terminadas sube siempre que un ítem es agregado a la cola. El conteo baja siempre que la ejecución de una corrutina `task_done()` para indicar que el ítem fue recuperado y todo el trabajo en él está completo. Cuando el conteo de tareas inacabadas llega a cero, `join()` se desbloquea.

coroutine put(item)

Pone un ítem en la cola. Si la cola está completa, espera hasta que una entrada vacía esté disponible antes de agregar el ítem.

put_nowait(item)

Pone un ítem en la cola sin bloquearse.

Si no hay inmediatamente disponibles entradas vacías, lanza `QueueFull`.

qsize()

Retorna el número de ítems en la cola.

task_done()

Indica que una tarea formalmente en cola está completa.

Usada por consumidores de la cola. Para cada `get()` usado para buscar una tarea, una ejecución subsecuente a `task_done()` dice a la cola que el procesamiento de la tarea está completo.

Si un `join()` está actualmente bloqueando, éste se resumirá cuando todos los ítems han sido procesados (implicado que un método `task_done()` fue recibido por cada ítem que ha sido `put()` en la cola.

Lanza `ValueError` si fue llamado más veces que los ítems en la cola.

Cola de prioridad

class asyncio.PriorityQueue

Una variante de `Queue`; recupera entradas en su orden de prioridad (el más bajo primero).

Las entradas son típicamente tuplas de la forma `(priority_number, data)`.

Cola UEPA (o *LIFO* en inglés)

`class asyncio.LifoQueue`

Una variante de una *Queue* que recupera primero el elemento agregado más reciente (último en entrar, primero en salir).

Excepciones

`exception asyncio.QueueEmpty`

Esta excepción es lanzada cuando el método `get_nowait()` es ejecutado en una cola vacía.

`exception asyncio.QueueFull`

Las excepciones son lanzadas cuando el método `put_nowait()` es lanzado en una cola que ha alcanzado su *maxsize*.

Ejemplos

Las colas pueden ser usadas para distribuir cargas de trabajo entre múltiples tareas concurrentes:

```

import asyncio
import random
import time

async def worker(name, queue):
    while True:
        # Get a "work item" out of the queue.
        sleep_for = await queue.get()

        # Sleep for the "sleep_for" seconds.
        await asyncio.sleep(sleep_for)

        # Notify the queue that the "work item" has been processed.
        queue.task_done()

        print(f'{name} has slept for {sleep_for:.2f} seconds')

async def main():
    # Create a queue that we will use to store our "workload".
    queue = asyncio.Queue()

    # Generate random timings and put them into the queue.
    total_sleep_time = 0
    for _ in range(20):
        sleep_for = random.uniform(0.05, 1.0)
        total_sleep_time += sleep_for
        queue.put_nowait(sleep_for)

    # Create three worker tasks to process the queue concurrently.
    tasks = []
    for i in range(3):
        task = asyncio.create_task(worker(f'worker-{i}', queue))
        tasks.append(task)

    # Wait until the queue is fully processed.
    started_at = time.monotonic()
    await queue.join()
    total_slept_for = time.monotonic() - started_at

```

(continué en la próxima página)

(proviene de la página anterior)

```
# Cancel our worker tasks.
for task in tasks:
    task.cancel()
# Wait until all worker tasks are cancelled.
await asyncio.gather(*tasks, return_exceptions=True)

print('====')
print(f'3 workers slept in parallel for {total_slept_for:.2f} seconds')
print(f'total expected sleep time: {total_sleep_time:.2f} seconds')

asyncio.run(main())
```

18.1.6 Excepciones

Código Fuente [Lib/asyncio/exceptions.py](#)

exception `asyncio.TimeoutError`

La operación ha excedido el tiempo límite.

Importante: Esta excepción es diferente a la excepción incorporada `TimeoutError`.

exception `asyncio.CancelledError`

La operación ha sido cancelada.

Esta excepción se puede capturar para realizar operaciones personalizadas cuando se cancelan las tareas de `asyncio`. En casi todas las situaciones, la excepción debe volver a lanzarse.

Distinto en la versión 3.8: `CancelledError` es ahora una subclase de `BaseException`.

exception `asyncio.InvalidStateError`

Estado Interno no válido de `Task` o `Future`.

Se puede lanzar en situaciones como establecer un valor de resultado para un objeto `Future` que ya tiene un valor de resultado establecido.

exception `asyncio.SendfileNotAvailableError`

La llamada al sistema «sendfile» no esta disponible desde el `socket` o tipo de archivo dado.

Una subclase de `RuntimeError`.

exception `asyncio.IncompleteReadError`

La operación de lectura solicitada no se completó completamente.

Lanzado por la `asyncio stream APIs`.

La excepción es una subclase de `EOFError`.

expected

El número total (`int`) de bytes esperados.

partial

Un cadena de `bytes` leída antes de que alcance al final del flujo.

exception `asyncio.LimitOverrunError`

Alcanzó el límite de tamaño del búfer mientras buscaba un separador.

Lanzado por `asyncio stream APIs`.

consumed

El número total de bytes que se consumirán.

18.1.7 Bucle de eventos

Código fuente: `Lib/asyncio/events.py`, `Lib/asyncio/base_events.py`

Prólogo

El bucle de eventos es el núcleo de cada aplicación `asyncio`. Los bucles de eventos ejecutan tareas asíncronas y llamadas de retorno, realizan operaciones de E/S de red y ejecutan subprocesos.

Los desarrolladores de aplicaciones normalmente deberían usar las funciones `asyncio` de alto nivel, como: `asyncio.run()`, y rara vez deberían necesitar hacer referencia al objeto de bucle o llamar a sus métodos. Esta sección está dirigida principalmente a autores de código de nivel inferior, bibliotecas y frameworks, quienes necesitan un control mas preciso sobre el comportamiento del bucle de eventos.

Obtención del bucle de eventos

Las siguientes funciones de bajo nivel se pueden utilizar para obtener, establecer o crear un bucle de eventos:

`asyncio.get_running_loop()`

Retorna el bucle de eventos en ejecución en el hilo del sistema operativo actual.

Si no hay un bucle de eventos en ejecución, se levanta un `RuntimeError`. Esta función únicamente puede ser llamada desde una corrutina o una llamada de retorno.

Nuevo en la versión 3.7.

`asyncio.get_event_loop()`

Obtiene bucle de eventos actual.

Si no hay un bucle de eventos actual establecido en el hilo actual del sistema operativo, el hilo del sistema operativo es el principal, y `set_event_loop()` aún no ha sido llamado, `asyncio` creará un nuevo bucle de eventos y lo establecerá como el actual.

Dado que esta función tiene un comportamiento bastante complejo (especialmente cuando están en uso las políticas de bucle de eventos personalizadas), usar la función `get_running_loop()` es preferible antes que `get_event_loop()` en corrutinas y llamadas de retorno.

Considere también usar la función `asyncio.run()` en lugar de usar funciones de bajo nivel para crear y cerrar manualmente un bucle de eventos.

`asyncio.set_event_loop(loop)`

Establece `loop` como el bucle de eventos actual para el hilo actual del sistema operativo.

`asyncio.new_event_loop()`

Crea un nuevo objeto de bucle de eventos.

Tenga en cuenta que el comportamiento de las funciones `get_event_loop()`, `set_event_loop()`, y `new_event_loop()` puede ser modificado mediante *estableciendo una política de bucle de eventos personalizada*.

Contenidos

Esta página de documentación contiene las siguientes secciones:

- La sección *Métodos del bucle de eventos* es la documentación de referencia de las APIs del bucle de eventos;
- La sección *Callback Handles* documenta las instancias *Handle* y *TimerHandle* las cuales son retornadas por métodos planificados como `loop.call_soon()` y `loop.call_later()`;
- La sección *Objetos del servidor* documenta tipos retornados por los métodos del bucle de eventos como `loop.create_server()`;
- La sección *Implementaciones de bucle de eventos* documenta las clases *SelectorEventLoop* y *ProactorEventLoop*;
- La sección *Ejemplos* muestra como trabajar con algunas APIs de bucle de eventos.

Métodos del bucle de eventos

Los bucles de eventos tienen APIs de **bajo nivel** para lo siguiente:

- *Iniciar y para el bucle*
- *Programación de llamadas de retorno*
- *Planificando llamadas retardadas*
- *Creando Futuros y Tareas*
- *Abriendo conexiones de red*
- *Creando servidores de red*
- *Transfiriendo archivos*
- *Actualización de TLS*
- *Viendo descriptores de archivos*
- *Trabajar con objetos sockets directamente*
- *DNS*
- *Trabajando con tuberías*
- *Señales Unix*
- *Ejecutando código en un hilos o grupos de procesos*
- *API para manejo de errores*
- *Habilitando el modo depuración*
- *Ejecutando Subprocesos*

Iniciar y para el bucle

`loop.run_until_complete(future)`

Se ejecuta hasta que *future* (una instancia de *Future*) se haya completado.

Si el argumento es un *objeto corrutina* está implícitamente planificado para ejecutarse como una *asyncio.Task*.

Retorna el resultado del Futuro o genera una excepción.

`loop.run_forever()`

Ejecuta el bucle de eventos hasta que *stop()* es llamado.

Si *stop()* es llamado antes que *run_forever()*, el bucle va a sondear el selector de E/S una sola vez con un plazo de ejecución de cero, ejecuta todas las llamadas planificadas como respuesta a eventos E/S (y aquellas que ya hayan sido planificados), y entonces termina.

Si *stop()* es llamado mientras *run_forever()* se está ejecutando, el loop ejecutará el lote actual de llamadas y después finalizará. Tenga en cuenta que llamadas planificadas por otras llamadas no se ejecutarán en este caso; en su lugar, ellas correrán la próxima vez que *run_forever()* o *run_until_complete()* sean llamados.

`loop.stop()`

Detener el bucle de eventos.

`loop.is_running()`

Retorna True si el bucle de eventos esta en ejecución actualmente.

`loop.is_closed()`

Retorna True si el bucle de eventos se cerró.

`loop.close()`

Cierra el bucle de eventos.

El bucle no debe estar en ejecución cuando se llama a esta función. Cualquier llamada de retorno pendiente será descartada.

Este método limpia todas las colas y apaga el ejecutor, pero no espera a que el ejecutor termine.

Este método es idempotente e irreversible. No se debe llamar ningún otro método después que el bucle de eventos es cerrado.

coroutine `loop.shutdown_asyncgens()`

Programa todos los objetos *asynchronous generator* abiertos actualmente para cerrarlos con una llamada *aclose()*. Después de llamar este método, el bucle de eventos emitirá una advertencia si un nuevo generador asíncrono es iterado. Esto debe ser usado para finalizar de manera confiable todos los generadores asíncronos planificados.

Tenga en cuenta que no hay necesidad de llamar esta función cuando *asyncio.run()* es utilizado.

Ejemplo:

```
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

Nuevo en la versión 3.6.

Programación de llamadas de retorno

`loop.call_soon(callback, *args, context=None)`

Schedule the *callback* `callback` to be called with *args* arguments at the next iteration of the event loop.

Llamadas que son ejecutadas en el orden en el que fueron registradas. Cada llamada será ejecutada exactamente una sola vez.

Un argumento *context* opcional y solo de palabra clave que permite especificar una clase `contextvars.Context` personalizada en la cual *callback* será ejecutada. Cuando no se provee *context* el contexto actual es utilizado.

Una instancia de `asyncio.Handle` es retornada, que puede ser utilizada después para cancelar la llamada.

Este método no es seguro para subprocesos.

`loop.call_soon_threadsafe(callback, *args, context=None)`

Una variante de `call_soon()` que es segura para subprocesos. Debe ser usada en llamadas planificadas desde otro hilo.

Vea sección *concurrency y multiproceso* de la documentación.

Distinto en la versión 3.7: Fue agregado el parámetro solo de palabra clave *context*. Vea **PEP 567** para mas detalles.

Nota: La mayoría de las funciones planificadas de `asyncio` no permiten pasar argumentos de palabra clave. Para hacer eso utilice `functools.partial()`:

```
# will schedule "print("Hello", flush=True)"
loop.call_soon(
    functools.partial(print, "Hello", flush=True))
```

El uso de objetos parciales es usualmente mas conveniente que utilizar lambdas, ya que `asyncio` puede renderizar mejor objetos parciales en mensajes de depuración y error.

Planificando llamadas retardadas

El bucle de eventos provee mecanismos para planificar funciones de llamadas que serán ejecutadas en algún punto en el futuro. El bucle de eventos usa relojes monotónicos para seguir el tiempo.

`loop.call_later(delay, callback, *args, context=None)`

Planifica *callback* para ser ejecutada luego de *delay* número de segundos (puede ser tanto un entero como un flotante).

Una instancia de `asyncio.TimerHandle` es retornada, la que puede ser utilizada para cancelar la ejecución.

callback será ejecutada exactamente una sola vez. Si dos llamadas son planificadas para el mismo momento exacto, el orden en el que son ejecutadas es indefinido.

El argumento posicional opcional *args* será pasado a la llamada cuando esta sea ejecutada. Si quieres que la llamada sea ejecutada con argumentos de palabra clave usa `functools.partial()`.

Un argumento *context* opcional y solo de palabra clave que permite especificar una clase `contextvars.Context` personalizada en la cual *callback* será ejecutada. Cuando no se provee *context* el contexto actual es utilizado.

Distinto en la versión 3.7: Fue agregado el parámetro solo de palabra clave *context*. Vea **PEP 567** para mas detalles.

Distinto en la versión 3.8: En Python 3.7 y versiones anteriores con la implementación del bucle de eventos predeterminada, el *delay* no puede exceder un día. Esto fue arreglado en Python 3.8.

`loop.call_at` (*when*, *callback*, **args*, *context=None*)

Planifica *callback* para ser ejecutada en una marca de tiempo absoluta *when* (un entero o un flotante), usando la misma referencia de tiempo que `loop.time()`.

El comportamiento de este método es el mismo que `call_later()`.

Una instancia de `asyncio.TimerHandle` es retornada, la que puede ser utilizada para cancelar la ejecución.

Distinto en la versión 3.7: Fue agregado el parámetro solo de palabra clave *context*. Vea [PEP 567](#) para mas detalles.

Distinto en la versión 3.8: En Python 3.7 y versiones anteriores con la implementación del bucle de eventos predeterminada, la diferencia entre *when* y el tiempo actual no puede exceder un día. Esto fue arreglado en Python 3.8.

`loop.time()`

Retorna el tiempo actual, como un *float*, de acuerdo al reloj monotónico interno del bucle de evento.

Nota: Distinto en la versión 3.8: En Python 3.7 y versiones anteriores los tiempos de espera (*delay* relativo o *when* absoluto) no deben exceder un día. Esto fue arreglado en Python 3.8.

Ver también:

La función `asyncio.sleep()`.

Creando Futuros y Tareas

`loop.create_future()`

Crea un objeto `asyncio.Future` adjunto al bucle de eventos.

Esta es la manera preferida de crear Futures en asyncio. Esto permite que bucles de eventos de terceros provean implementaciones alternativas del objeto Future (con mejor rendimiento o instrumentación).

Nuevo en la versión 3.5.2.

`loop.create_task` (*coro*, ***, *name=None*)

Planifica la ejecución de una *Corrutinas*. Retorna un objeto *Task*.

Bucles de eventos de terceros pueden usar sus propias subclases de *Task* por interoperabilidad. En este caso, el tipo de resultado es una subclase de *Task*.

Si el argumento *name* es provisto y no None, se establece como el nombre de la tarea usando `Task.set_name()`.

Distinto en la versión 3.8: Agregado el parámetro *name*.

`loop.set_task_factory` (*factory*)

Establece una fábrica de tareas que será utilizada por `loop.create_task()`.

Si *factory* es None se establecerá la fábrica de tareas por defecto. En cualquier otro caso, *factory* debe ser un *callable* con la misma firma (`loop, coro`), donde *loop* es una referencia al bucle de eventos activo y *coro* es un objeto de corrutina. El ejecutable debe retornar una objeto `asyncio.Future` compatible.

`loop.get_task_factory()`

Retorna una fábrica de tareas o None si la predefinida está en uso.

Abriendo conexiones de red

coroutine `loop.create_connection` (*protocol_factory*, *host=None*, *port=None*, *, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*)

Abre una conexión de transmisión de transporte a una dirección especificada por *host* y *port*.

La familia de sockets puede ser tanto `AF_INET` como `AF_INET6` dependiendo de *host* (o del argumento *family* si es que fue provisto).

El tipo de socket será `SOCK_STREAM`.

protocol_factory debe ser un ejecutable que retorna una implementación del *asyncio protocol*.

Este método tratará de establecer la conexión en un segundo plano. Cuando es exitosa, retorna un par (*transport*, *protocol*).

La sinopsis cronológica de las operaciones subyacentes es como sigue:

1. La conexión es establecida y un *transport* es creado para ello.
2. *protocol_factory* es llamado sin argumentos y se espera que retorne una instancia de *protocol*.
3. La instancia del protocolo se acopla con el transporte mediante el llamado de su método `connection_made()`.
4. Una tupla (*transport*, *protocol*) es retornada cuando se tiene éxito.

El transporte creado es una transmisión (*stream*) bidireccional que depende de la implementación.

Otros argumentos:

- *ssl*: si se provee y no es falso, un transporte SSL/TLS es creado (de manera predeterminada se crea un transporte TCP plano). Si *ssl* es un objeto `ssl.SSLContext`, este contexto es utilizado para crear el transporte; si *ssl* es `True`, se utiliza un contexto predeterminado retornado por `ssl.create_default_context()`.

Ver también:

Consideraciones de seguridad SSL/TLS

- *server_hostname* establece o reemplaza el nombre de servidor (*hostname*) contra el cual el certificado del servidor de destino será comparado. Sólo debería ser pasado si *ssl* no es `None`. De manera predeterminada es usado el valor del argumento *host*. Si *host* está vacío, no hay valor predeterminado y debes pasar un valor para *server_hostname*. Si *server_hostname* es una cadena vacía, la comparación de nombres de servidores es deshabilitada (lo que es un riesgo de seguridad serio, permitiendo potenciales ataques de hombre-en-el-medio, *man-in-the-middle attacks*).
- *family*, *proto*, *flags* son dirección de familia, protocolo y banderas opcionales que serán pasadas a través de `getaddrinfo()` para la resolución de *host*. Si están dados, todos ellos deberían ser enteros de las constantes del módulo `socket` correspondiente.
- *happy_eyeballs_delay*, if given, enables Happy Eyeballs for this connection. It should be a floating-point number representing the amount of time in seconds to wait for a connection attempt to complete, before starting the next attempt in parallel. This is the «Connection Attempt Delay» as defined in [RFC 8305](#). A sensible default value recommended by the RFC is 0.25 (250 milliseconds).
- *interleave* controla reordenamientos de dirección cuando un nombre de servidor resuelve a múltiples direcciones IP. Si es 0 o no es especificado, no se hace ningún reordenamiento, y las direcciones son intentadas en el orden retornado por `getaddrinfo()`. Si un entero positivo es especificado, las direcciones son intercaladas por dirección de familia, y el entero dado es interpretado como «Número de familias de la primera dirección» (*First Address Family Count*) como es definida en [RFC 8305](#). El valor predefinido es 0 si *happy_eyeballs_delay* no es especificado, y 1 si lo es.

- *sock*, si está dado, debe ser un objeto `socket.socket` existente y ya conectado, que será utilizado por el transporte. Si *sock* es dado, ningún *host*, *port*, *family*, *proto*, *flags*, *happy_eyeballs_delay*, *interleave* o *local_addr* deben ser especificados.
- *local_addr*, si está dado, es una tupla (`local_host`, `local_port`) usada para enlazar el socket localmente. Los `*local_host*` y `*local_port*` son buscados usando `getaddrinfo()`, de manera similar que con *host* y *puerto*.
- *ssl_handshake_timeout* es (para una conexión TLS) el tiempo en segundos a esperar que se complete el apretón de manos (*handshake*) TLS antes de abortar la conexión. 60.0 segundos si es `None` (predeterminado).

Nuevo en la versión 3.8: Agregados los parámetros *happy_eyeballs_delay* y *interleave*.

Algoritmo de Globos Oculares Felices (*Happy Eyeballs*): Éxito con Servidores de Doble-Pila (Dual-Stack Hosts). Cuando la ruta IPv4 y el protocolo de un servidor están funcionando, pero la ruta IPv6 y el protocolo no están funcionando, una aplicación del cliente de doble-pila experimenta una demora de conexión significativa en comparación con un cliente sólo de IPv4. Esto no es deseable porque causa que el cliente de doble-pila tenga la peor experiencia de usuario. Este documento especifica requerimientos para algoritmos que reducen esta demora visible por el usuario, y provee un algoritmo.

Para mas información: <https://tools.ietf.org/html/rfc6555>

Nuevo en la versión 3.7: El parámetro *ssl_handshake_timeout*.

Distinto en la versión 3.6: La opción del socket `TCP_NODELAY` es establecida de manera predeterminada para todas las conexiones TCP.

Distinto en la versión 3.5: Agregado el soporte para SSL/TLS en *ProactorEventLoop*.

Ver también:

La función `open_connection()` es una API alternativa de alto nivel. Retorna un par de (*StreamReader*, *StreamWriter*) que puede ser usado directamente en código `async/await`.

```
coroutine loop.create_datagram_endpoint(protocol_factory, local_addr=None,
                                         remote_addr=None, *, family=0, proto=0,
                                         flags=0, reuse_address=None, reuse_port=None,
                                         allow_broadcast=None, sock=None)
```

Nota: El parámetro *reuse_address* ya no es soportado, como utiliza `SO_REUSEADDR` plantea un problema de seguridad importante para UDP. Pasando explícitamente *reuse_address=True* lanzará una excepción.

Cuando múltiples procesos con `UIDs` diferentes asignan sockets a una misma dirección socket UDP con `SO_REUSEADDR`, los paquetes entrantes pueden distribuirse aleatoriamente entre los sockets.

Para plataformas soportadas, *reuse_port* puede ser utilizado como un reemplazo para funcionalidades similares. Con *reuse_port*, `SO_REUSEPORT` es usado en su lugar, que específicamente previene que procesos con distintos `UIDs` asignen sockets a la misma dirección de socket.

Crea un datagrama de conexión.

La familia de socket puede ser tanto `AF_INET`, `AF_INET6`, como `AF_UNIX`, dependiendo de *host* (o del argumento *family*, si fue provisto).

El tipo de socket será `SOCK_DGRAM`.

protocol_factory debe ser un ejecutable que retorne una implementación de *protocol*.

Una tupla de (`transport`, `protocol`) es retornada cuando se tiene éxito.

Otros argumentos:

- *local_addr*, si está dado, es una tupla (`local_host`, `local_port`) usada para enlazar el socket localmente. Los *local_host* y *local_port* son buscados utilizando `getaddrinfo()`.

- *remote_addr*, si está dado, es una tupla (*remote_host*, *remote_port*) utilizada para conectar el socket a una dirección remota. Los *remote_host* y *remote_port* son buscados utilizando *getaddrinfo()*.
- *family*, *proto*, *flags* son direcciones de familia, protocolo y banderas opcionales que serán pasadas a través de *getaddrinfo()* para la resolución de *host*. Si está dado, estos deben ser todos enteros de las constantes del módulo *socket* correspondiente.
- *reuse_port* dice al kernel que habilite este punto de conexión para ser unido al mismo puerto de la misma forma que otros puntos de conexión existentes también están unidos, siempre y cuando todos ellos establezcan esta bandera al ser creados. Esta opción no es soportada en Windows y algunos sistemas Unix. Si la constante *SO_REUSEPORT* no está definida entonces esta funcionalidad no es soportada.
- *allow_broadcast* dice al kernel que habilite este punto de conexión para enviar mensajes a la dirección de transmisión (*broadcast*).
- *sock* puede opcionalmente ser especificado para usar un objeto *socket.socket* preexistente y ya conectado que será utilizado por el transporte. Si están especificados, *local_addr* y *remote_addr* deben ser omitidos (tienen que ser *None*).

Refiérase a los ejemplos *UDP echo client protocol* y *UDP echo server protocol*.

Distinto en la versión 3.4.4: Los parámetros *family*, *proto*, *flags*, *reuse_address*, *reuse_port*, *allow_broadcast* y *sock* fueron agregados.

Distinto en la versión 3.8.1: El parámetro *reuse_address* ya no es soportado debido a problemas de seguridad.

Distinto en la versión 3.8: Se agregó soporte para Windows.

```
coroutine loop.create_unix_connection(protocol_factory, path=None, *, ssl=None,
                                     sock=None, server_hostname=None,
                                     ssl_handshake_timeout=None)
```

Crear una conexión Unix.

La familia de sockets será *AF_UNIX*; el tipo de socket será *SOCK_STREAM*.

Una tupla de (*transport*, *protocol*) es retornada cuando se tiene éxito.

path es el nombre de un dominio de un socket Unix y es requerido, a menos que un parámetro *sock* sea especificado. Los socket Unix abstractos, *str*, *bytes*, y *Path* son soportados.

Vea la documentación del método *loop.create_connection()* para información acerca de los argumentos de este método.

Availability: Unix.

Nuevo en la versión 3.7: El parámetro *ssl_handshake_timeout*.

Distinto en la versión 3.7: El parámetro *path* ahora puede ser un *path-like object*.

Creando servidores de red

```
coroutine loop.create_server(protocol_factory, host=None, port=None, *, fa-
                             mily=socket.AF_UNSPEC, flags=socket.AI_PASSIVE, sock=None,
                             backlog=100, ssl=None, reuse_address=None, reuse_port=None,
                             ssl_handshake_timeout=None, start_serving=True)
```

Crea un servidor TCP (tipo de socket *SOCK_STREAM*) escuchando en *port* de la dirección *host*.

Retorna un objeto *Server*.

Argumentos:

- *protocol_factory* debe ser un ejecutable que retorne una implementación de *protocol*.
- El parámetro *host* puede ser establecido a distintos tipos que determinan donde el servidor estaría escuchando:
 - Si *host* es una cadena, el servidor TCP está enlazado a una sola interfaz de red especificada por *host*.

- Si *host* es una secuencia de cadenas, el servidor TCP está enlazado a todas las interfaces de red especificadas por la secuencia.
- Si *host* es una cadena vacía o `None`, se asumen todas las interfaces y una lista con múltiples sockets será retornada (mas probablemente uno para IPv4 y otro para IPv6).
- *family* puede ser establecido como `socket.AF_INET` o `AF_INET6` para forzar al socket a usar IPv4 o IPv6. Si no es establecido, la *family* será determinada por medio del nombre del host (por defecto será `AF_UNSPEC`).
- *flags* es una máscara de bits para `getaddrinfo()`.
- *sock* puede ser especificado opcionalmente para usar objetos socket preexistentes. Si se utiliza, entonces *host* y *port* no deben ser especificados.
- *backlog* es el número máximo de conexiones encoladas pasadas a `listen()` (el valor predeterminado es 100).
- *ssl* puede ser establecido como una instancia de `SSLContext` para habilitar TLS sobre las conexiones aceptadas.
- *reuse_address* indica al kernel que reutilice un socket local en estado `TIME_WAIT`, sin esperar que su plazo de ejecución expire. Si no es especificado será establecido automáticamente como `True` en Unix.
- *reuse_port* dice al kernel que habilite este punto de conexión para ser unido al mismo puerto de la misma forma que otros puntos de conexión existentes también están unidos, siempre y cuando todos ellos establezcan esta bandera al ser creados.
- *ssl_handshake_timeout* es (para un servidor TLS) el tiempo en segundos a esperar por el apretón de manos (*handshake*) TLS a ser completado antes de abortar la conexión. `60.0` si es `None` (su valor predeterminado).
- *start_serving* establecido como `True` (de manera predeterminada) produce que los servidores creados comiencen a aceptar conexiones inmediatamente. Si es establecido como `False`, el usuario debe esperar por `Server.start_serving()` o `Server.serve_forever()` para que el servidor comience a aceptar conexiones.

Nuevo en la versión 3.7: Agregados los parámetros *ssl_handshake_timeout* y *start_serving*.

Distinto en la versión 3.6: La opción del socket `TCP_NODELAY` es establecida de manera predeterminada para todas las conexiones TCP.

Distinto en la versión 3.5: Agregado el soporte para SSL/TLS en `ProactorEventLoop`.

Distinto en la versión 3.5.1: El parámetro *host* puede ser una secuencia de cadenas.

Ver también:

La función `start_server()` es una API alternativa de alto nivel que retorna un par de `StreamReader` y `StreamWriter` que pueden ser usados en código `async/await`.

```
coroutine loop.create_unix_server(protocol_factory, path=None, *, sock=None, backlog=100, ssl=None, ssl_handshake_timeout=None, start_serving=True)
```

Similar a `loop.create_server()` pero funciona con la familia de sockets `AF_UNIX`.

path es el nombre de un dominio de socket Unix, y es requerido a menos que el argumento *sock* sea provisto. Son soportados sockets unix abstractos, *str*, *bytes*, y rutas *Path*.

Vea la documentación de el método `loop.create_server()` para mas información acerca de los argumentos de este método.

Availability: Unix.

Nuevo en la versión 3.7: Los parámetros *ssl_handshake_timeout**y **start_serving*.

Distinto en la versión 3.7: El parámetro *path* ahora puede ser un objeto *Path*.

coroutine `loop.connect_accepted_socket(protocol_factory, sock, *, ssl=None, ssl_handshake_timeout=None)`

Envuelve una conexión ya aceptada en un par de transporte/protocolo.

Este método puede ser usado por servidores que acepten conexiones por fuera de `asyncio`, pero que usen `asyncio` para manejarlas.

Parámetros:

- `protocol_factory` debe ser un ejecutable que retorne una implementación de `protocol`.
- `sock` es un objeto `socket` preexistente retornado por `socket.accept`.
- `ssl` puede ser establecido como un `SSLContext` para habilitar SSL sobre las conexiones aceptadas.
- `ssl_handshake_timeout` es (para una conexión SSL) el tiempo en segundos que se esperará para que se complete el apretón de manos (*handshake*) SSL antes de abortar la conexión. `60.0` si es `None` (su valor predeterminado).

Retorna un par (`transport`, `protocol`).

Nuevo en la versión 3.7: El parámetro `ssl_handshake_timeout`.

Nuevo en la versión 3.5.3.

Transfiriendo archivos

coroutine `loop.sendfile(transport, file, offset=0, count=None, *, fallback=True)`

Envía un `file` a través de un `transport`. Retorna el numero total de bytes enviados.

El método usa `os.sendfile()` de alto rendimiento si está disponible.

`file` debe ser un objeto de archivo regular abierto en modo binario.

`offset` indica desde donde se empezará a leer el archivo. Si es especificado, `count` es el número total de bytes a transmitir en contraposición con enviar el archivo hasta que se alcance EOF. La posición del archivo es actualizada siempre, incluso cuando este método genere un error, y `file.tell()` puede ser usado para obtener el número de bytes enviados hasta el momento.

`fallback` establecido como `True` hace que `asyncio` lea y envíe el archivo manualmente cuando la plataforma no soporta la llamada de envío de archivos del sistema (por ejemplo, Windows o `sockets` SSL en Unix).

Lanza `SendfileNotAvailableError` si el sistema no soporta la llamada de envío de archivos del sistema y `fallback` es `True`.

Nuevo en la versión 3.7.

Actualización de TLS

coroutine `loop.start_tls(transport, protocol, sslcontext, *, server_side=False, server_hostname=None, ssl_handshake_timeout=None)`

Actualiza una conexión basada en transporte ya existente a TLS.

Retorna una nueva instancia de transporte, que el `protocol` debe empezar a usar inmediatamente después del `await`. La instancia `transport` pasada al método `start_tls` nunca debe ser usada de nuevo.

Parámetros:

- Las instancias `transport` y `protocol` que retornan los métodos como `create_server()` y `create_connection()`.
- `sslcontext`: una instancia configurada de `SSLContext`.
- `server_side` pasa `True` cuando se actualiza una conexión del lado del servidor (como en el caso de una creada por `create_server()`).

- `server_hostname`: establece o reemplaza el nombre del host contra el cual se compara el certificado del servidor de destino.
- `ssl_handshake_timeout` es (para una conexión TLS) el tiempo en segundos a esperar que se complete el apretón de manos (*handshake*) TLS antes de abortar la conexión. 60.0 segundos si es `None` (predeterminado).

Nuevo en la versión 3.7.

Viendo descriptores de archivos

`loop.add_reader(fd, callback, *args)`

Empieza a monitorear el descriptor de archivos `fd` para disponibilidad de lectura e invoca `callback` con los argumentos especificados una vez que `fd` está habilitado para ser leído.

`loop.remove_reader(fd)`

Deja de monitorear el descriptor de archivos `fd` para disponibilidad de lectura.

`loop.add_writer(fd, callback, *args)`

Empieza a monitorear el descriptor de archivos `fd` para disponibilidad de escritura e invoca `callback` con los argumentos especificados una vez que `fd` está habilitado para ser escrito.

Use `functools.partial()` para pasar argumentos de palabra clave a `callback`.

`loop.remove_writer(fd)`

Deja de monitorear el descriptor de archivos `fd` para disponibilidad de escritura.

Vea también la sección [Soporte de plataforma](#) para algunas limitaciones de estos métodos.

Trabajar con objetos sockets directamente

En general, implementaciones de protocolo que usen APIs basadas en transporte como `loop.create_connection()` y `loop.create_server()` son mas rápidas que aquellas implementaciones que trabajan directamente con sockets. De cualquier forma, hay algunos casos de uso en los cuales el rendimiento no es crítico, y trabajar directamente con objetos `socket` es mas conveniente.

coroutine `loop.sock_recv(sock, nbytes)`

Recibe hasta `nbytes` de `sock`. Versión asíncrona de `socket.recv()`.

Retorna los datos recibidos como un objeto bytes.

`sock` debe ser un socket no bloqueante.

Distinto en la versión 3.7: A pesar de que este método siempre fue documentado como un método de corrutina, los lanzamientos previos a Python 3.7 retornaban un `Future`. Desde Python 3.7 este es un método `async def`.

coroutine `loop.sock_recv_into(sock, buf)`

Recibe datos desde `sock` en el búfer `buf`. Modelado después del método bloqueante `socket.recv_into()`.

Retorna el número de bytes escritos en el búfer.

`sock` debe ser un socket no bloqueante.

Nuevo en la versión 3.7.

coroutine `loop.sock_sendall(sock, data)`

Envía `data` al socket `sock`. Versión asíncrona de `socket.sendall()`.

Este método continua enviando al socket hasta que se hayan enviado todos los datos en `data` u ocurra un error. `None` es retornado cuando se tiene éxito. Cuando ocurre un error, se lanza una excepción. Adicionalmente, no hay manera de determinar cuantos datos, si es que se hubo alguno, se procesaron correctamente por el extremo receptor de la conexión.

`sock` debe ser un socket no bloqueante.

Distinto en la versión 3.7: A pesar de que este método siempre fue documentado como un método de corrutina, antes de Python 3.7 retorna un *Future*. Desde Python 3.7, este es un método `async def`.

coroutine `loop.sock_connect(sock, address)`

Conecta *sock* a un socket remoto en *address*.

Versión asíncrona de `socket.connect()`.

sock debe ser un socket no bloqueante.

Distinto en la versión 3.5.2: *address* ya no necesita ser resuelto. `sock_connect` va a intentar verificar si *address* ya fue resuelto a partir del llamado de `socket.inet_pton()`. Si no lo fue, se utilizará `loop.getaddrinfo()` para resolver *address*.

Ver también:

`loop.create_connection()` y `asyncio.open_connection()`.

coroutine `loop.sock_accept(sock)`

Acepta una conexión. Modelado después del método bloqueante `socket.accept()`.

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.

sock debe ser un socket no bloqueante.

Distinto en la versión 3.7: A pesar de que este método siempre fue documentado como un método de corrutina, antes de Python 3.7 retorna un *Future*. Desde Python 3.7, este es un método `async def`.

Ver también:

`loop.create_server()` y `start_server()`.

coroutine `loop.sock_sendfile(sock, file, offset=0, count=None, *, fallback=True)`

Envía un archivo usando `os.sendfile` de alto rendimiento si es posible. Retorna el número total de bytes enviados.

Versión asíncrona de `socket.sendfile()`.

sock debe ser un `socket.SOCK_STREAM` socket no bloqueante.

file debe ser un objeto de archivo regular abierto en modo binario.

offset indica desde donde se empezará a leer el archivo. Si es especificado, *count* es el número total de bytes a transmitir en contraposición con enviar el archivo hasta que se alcance EOF. La posición del archivo es actualizada siempre, incluso cuando este método genere un error, y `file.tell()` puede ser usado para obtener el número de bytes enviados hasta el momento.

fallback, cuando es establecida como `True`, hace que `asyncio` lea y escriba el archivo manualmente cuando el sistema no soporta la llamada de envío de archivos del sistema (por ejemplo, Windows o sockets SSL en Unix).

Lanza `SendfileNotAvailableError` si el sistema no soporta la llamada de envío de archivos del sistema *sendfile* y *fallback* es `False`.

sock debe ser un socket no bloqueante.

Nuevo en la versión 3.7.

DNS

coroutine `loop.getaddrinfo(host, port, *, family=0, type=0, proto=0, flags=0)`
Versión asíncrona de `socket.getaddrinfo()`.

coroutine `loop.getnameinfo(sockaddr, flags=0)`
Asynchronous version of `socket.getnameinfo()`.

Distinto en la versión 3.7: Ambos métodos `getaddrinfo` y `getnameinfo` siempre fueron documentados para retornar una corrutina, pero antes de Python 3.7 retornaban, de hecho, objetos *Future*. A partir de Python 3.7, ambos métodos son corrutinas.

Trabajando con tuberías

coroutine `loop.connect_read_pipe(protocol_factory, pipe)`
Registra el fin de lectura de *pipe* en el bucle de eventos.

protocol_factory debe ser un ejecutable que retorna una implementación del *asyncio protocol*.

pipe es un *objeto de tipo archivo*.

Retorna un par (*transport*, *protocol*), donde *transport* soporta la interface *ReadTransport* y *protocol* es un objeto instanciado por *protocol_factory*.

Con el bucle de eventos *SelectorEventLoop*, el *pipe* es establecido en modo no bloqueante.

coroutine `loop.connect_write_pipe(protocol_factory, pipe)`
Registra el fin de escritura de *pipe* en el bucle de eventos.

protocol_factory debe ser un ejecutable que retorna una implementación del *asyncio protocol*.

pipe es un *objeto de tipo archivo*.

Retorna un par (*transport*, *protocol*), donde *transport* soporta la interface *WriteTransport* y *protocol* es un objeto inicializado por *protocol_factory*.

Con el bucle de eventos *SelectorEventLoop*, el *pipe* es establecido en modo no bloqueante.

Nota: *SelectorEventLoop* no soporta los métodos anteriores en windows. En su lugar, use *ProactorEventLoop* para Windows.

Ver también:

Los métodos `loop.subprocess_exec()` y `loop.subprocess_shell()`.

Señales Unix

`loop.add_signal_handler(signum, callback, *args)`
Establece *callback* como el gestor para la señal *signum*.

La llamada será invocada por *loop*, junto con otras llamadas encoladas y corrutinas ejecutables de ese bucle de eventos. A menos que los gestores de señal la registren usando `signal.signal()`, una llamada registrada con esta función tiene permitido interactuar con el bucle de eventos.

Lanza *ValueError* si el número de señal es invalido o inalcanzable. Lanza *RuntimeError* si hay algún problema preparando el gestor.

Use `functools.partial()` para pasar argumentos de palabra clave a *callback*.

Como `signal.signal()`, esta función debe ser invocada en el hilo principal.

`loop.remove_signal_handler(sig)`

Elimina el gestor para la señal *sig*.

Retorna `True` si el gestor de señal fue eliminado, o `False` si no se estableció gestor para la señal dada.

Availability: Unix.

Ver también:

El módulo `signal`.

Ejecutando código en un hilos o grupos de procesos

awaitable `loop.run_in_executor(executor, func, *args)`

Arrange for *func* to be called in the specified executor.

El argumento *executor* debe ser una instancia de `concurrent.futures.Executor`. El ejecutor predefinido es usado si *executor* es `None`.

Ejemplo:

```
import asyncio
import concurrent.futures

def blocking_io():
    # File operations (such as logging) can block the
    # event loop: run them in a thread pool.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    # CPU-bound operations will block the event loop:
    # in general it is preferable to run them in a
    # process pool.
    return sum(i * i for i in range(10 ** 7))

async def main():
    loop = asyncio.get_running_loop()

    ## Options:

    # 1. Run in the default loop's executor:
    result = await loop.run_in_executor(
        None, blocking_io)
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
    with concurrent.futures.ThreadPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, blocking_io)
        print('custom thread pool', result)

    # 3. Run in a custom process pool:
    with concurrent.futures.ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom process pool', result)

asyncio.run(main())
```

Este método retorna un objeto `asyncio.Future`.

Use `functools.partial()` para pasar argumentos de palabra clave a *func*.

Distinto en la versión 3.5.3: `loop.run_in_executor()` no longer configures the `max_workers` of the thread pool executor it creates, instead leaving it up to the thread pool executor (`ThreadPoolExecutor`) to set the default.

`loop.set_default_executor(executor)`

Establece `executor` como el ejecutor predeterminado utilizado por `run_in_executor()`. `executor` debe ser una instancia de `ThreadPoolExecutor`.

Obsoleto desde la versión 3.8: Usar un ejecutor que no es una instancia de `ThreadPoolExecutor` es obsoleto y disparará un error en Python 3.9.

`executor` debe ser una instancia de `concurrent.futures.ThreadPoolExecutor`.

API para manejo de errores

Permite personalizar como son manejadas las excepciones en el bucle de eventos.

`loop.set_exception_handler(handler)`

Establece `handler` como el nuevo gestor de excepciones del bucle de eventos.

Si `handler` es `None`, se establecerá el gestor de excepciones predeterminado. De otro modo, `handler` debe ser un invocable con la misma firma (`loop, context`), donde `loop` es una referencia al bucle de eventos activo, y `context` es un objeto `dict` que contiene los detalles de la excepción (vea la documentación de `call_exception_handler()` para detalles acerca del contexto).

`loop.get_exception_handler()`

Retorna el gesto de excepciones actual, o `None` si no fue establecido ningún gestor de excepciones personalizado.

Nuevo en la versión 3.5.2.

`loop.default_exception_handler(context)`

Gestor de excepciones por defecto.

Esto es llamado cuando ocurre una excepción y no se estableció ningún gestor de excepciones. Esto puede ser llamado por un gestor de excepciones personalizado que quiera cambiar el comportamiento del gestor predeterminado.

El parámetro `context` tiene el mismo significado que en `call_exception_handler()`.

`loop.call_exception_handler(context)`

Llama al gestor de excepciones del bucle de eventos actual.

`context` es un objeto `dict` conteniendo las siguientes claves (en futuras versiones de Python podrían introducirse nuevas claves):

- “message”: Mensaje de error;
- “exception” (opcional): Objeto de excepción;
- “future” (opcional): instancia de `asyncio.Future`;
- “handle” (opcional): instancia de `asyncio.Handle`;
- “protocol” (opcional): instancia de `Protocol`;
- “transport” (opcional): instancia de `Transport`;
- “socket” (opcional): instancia de `socket.socket`.

Nota: Este método no debe ser sobrecargado en bucles de eventos en subclase. Para gestión de excepciones personalizadas, use el método `set_exception_handler()`.

Habilitando el modo depuración

`loop.get_debug()`

Obtiene el modo depuración (*bool*) del bucle de eventos.

El valor predeterminado es `True` si la variable de entorno `PYTHONASYNCIODEBUG` es establecida a una cadena no vacía, de otro modo será `False`.

`loop.set_debug(enabled: bool)`

Establece el modo de depuración del bucle de eventos.

Distinto en la versión 3.7: La nueva opción de línea de comandos `-X dev` ahora también puede ser utilizada para habilitar el modo depuración.

Ver también:

El *modo depuración de asyncio*.

Ejecutando Subprocesos

Los métodos descritos en esta subsección son de bajo nivel. En código `async/await` regular considere usar las convenientes funciones de alto nivel `asyncio.create_subprocess_shell()` y `asyncio.create_subprocess_exec()`.

Nota: El bucle de eventos predeterminado de `asyncio` en **Windows** no soporta subprocesos. Vea *Soporte de subprocesos en Windows* para mas detalles.

coroutine `loop.subprocess_exec(protocol_factory, *args, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE, **kwargs)`

Crea un subproceso de uno o mas argumentos de cadena especificados por *args*.

args debe ser una lista de cadenas representadas por:

- *str*;
- o *bytes*, codificados a la *codificación del sistema de archivos*.

La primer cadena especifica el programa ejecutable, y las cadenas restantes especifican los argumentos. En conjunto, los argumentos de cadena forman el `argv` del programa.

Esto es similar a la clase de la librería estándar `subprocess.Popen` llamada con `shell=False` y la lista de cadenas pasadas como el primer argumento; de cualquier forma, cuando `Popen` toma un sólo argumento que es una lista de cadenas, `subprocess_exec` toma múltiples cadenas como argumentos.

El *protocol_factory* debe ser un ejecutable que retorne una subclase de la clase `asyncio.SubprocessProtocol`.

Otros parámetros:

- *stdin* puede ser cualquier de estos:
 - un objeto de tipo archivo representando una tubería que será conectada al flujo de entrada estándar del subproceso utilizando `connect_write_pipe()`
 - la constante `subprocess.PIPE` (predeterminado) que creará una tubería nueva y la conectará,
 - el valor `None` que hará que el subproceso herede el descriptor de archivo de este proceso
 - la constante `subprocess.DEVNULL` que indica que el archivo especial `os.devnull` será utilizado
- *stdout* puede ser cualquier de estos:
 - un objeto de tipo archivo representando una tubería que será conectada al flujo de salida estándar del subproceso utilizando `connect_write_pipe()`

- la constante `subprocess.PIPE` (predeterminado) que creará una tubería nueva y la conectará,
 - el valor `None` que hará que el subprocesso herede el descriptor de archivo de este proceso
 - la constante `subprocess.DEVNULL` que indica que el archivo especial `os.devnull` será utilizado
- `stderr` puede ser cualquier de estos:
 - un objeto de tipo archivo representando una tubería que será conectada al flujo de error estándar del subprocesso utilizando `connect_write_pipe()`
 - la constante `subprocess.PIPE` (predeterminado) que creará una tubería nueva y la conectará,
 - el valor `None` que hará que el subprocesso herede el descriptor de archivo de este proceso
 - la constante `subprocess.DEVNULL` que indica que el archivo especial `os.devnull` será utilizado
 - la constante `subprocess.STDOUT` que conectará el flujo de errores predeterminado al flujo de salida predeterminado del proceso
 - El resto de argumentos de palabra clave son pasados a `subprocess.Popen` sin interpretación, excepto por `bufsize`, `universal_newlines`, `shell`, `text`, `encoding` y `errors`, que no deben ser especificados en lo absoluto.
- La API subprocesso `asyncio` no soporta decodificar los flujos como texto. `bytes.decode()` puede ser usado para convertir a texto los bytes retornados por el flujo.

Vea el constructor de la clase `subprocess.Popen` para documentación acerca de otros argumentos.

Retorna un par de `(transport, protocol)`, donde `transport` se ajusta a la clase base `asyncio.SubprocessTransport` y `protocol` es un objeto instanciado por `protocol_factory`.

coroutine `loop.subprocess_shell(protocol_factory, cmd, *, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE, **kwargs)`
 Crea un subprocesso desde `cmd`, que puede ser una cadena `str` o `bytes` codificado a la *codificación del sistema de archivos*, usando la sintaxis «shell» de la plataforma.

Esto es similar a la clase de la librería estándar `subprocess.Popen` llamada con `shell=True`.

El `protocol_factory` debe ser un ejecutable que retorne una subclase de la clase `asyncio.SubprocessProtocol`.

Vea `subprocess_exec()` para mas detalles acerca de los argumentos restantes.

Retorna un par de `(transport, protocol)`, donde `transport` se ajusta a la clase base `SubprocessTransport` y `protocol` es un objeto instanciado por `protocol_factory`.

Nota: Es responsabilidad de la aplicación asegurar que todos los espacios en blanco y caracteres especiales estén escapados correctamente para evitar vulnerabilidades de *inyección de código*. La función `shlex.quote()` puede ser usada para escapar apropiadamente espacios en blanco y caracteres especiales en cadenas que van a ser usadas para construir comandos de consola.

Gestores de llamadas

class `asyncio.Handle`

Un objeto de contenedor de llamada retornado por `loop.call_soon()`, `loop.call_soon_threadsafe()`.

cancel()

Cancela la llamada. Si la llamada ya fue cancelada o ejecutada, este método no tiene efecto.

cancelled()

Retorna `True` si la llamada fue cancelada.

Nuevo en la versión 3.7.

class `asyncio.TimerHandle`

Un objeto de contenedor de llamada retornado por `loop.call_later()`, and `loop.call_at()`.

Esta clase es una subclase de `Handle`.

when()

Retorna el tiempo de una llamada planificada como `float` segundos.

El tiempo es una marca de tiempo absoluta, usando la misma referencia de tiempo que `loop.time()`.

Nuevo en la versión 3.7.

Objetos Servidor

Los objetos de servidor son creados por las funciones `loop.create_server()`, `loop.create_unix_server()`, `start_server()`, y `start_unix_server()`.

No instanciar la clase directamente.

class `asyncio.Server`

Los objetos `Server` son gestores de asíncronos de contexto. Cuando son usados en una declaración `async with`, está garantizado que el objeto Servidor está cerrado y no está aceptando nuevas conexiones cuando la declaración `async with` es completada:

```
srv = await loop.create_server(...)

async with srv:
    # some code

# At this point, srv is closed and no longer accepts new connections.
```

Distinto en la versión 3.7: El objeto Servidor es un gestor asíncrono de contexto desde Python 3.7.

close()

Deja de servir: deja de escuchar sockets y establece el atributo `sockets` a `None`.

Los sockets que representan conexiones entrantes existentes de clientes se dejan abiertas.

El servidor es cerrado de manera asíncrona, usa la corrutina `wait_closed()` para esperar hasta que el servidor esté cerrado.

get_loop()

Retorna el bucle de eventos asociado con el objeto Servidor.

Nuevo en la versión 3.7.

coroutine start_serving()

Comienza a aceptar conexiones.

Este método es idempotente, así que puede ser llamado cuando el servidor ya está sirviendo.

El parámetro sólo de palabra clave `start_serving` de `loop.create_server()` y `asyncio.start_server()` permite crear un objeto Servidor que no está aceptando conexiones inicialmente. En este caso `Server.start_serving()`, o `Server.serve_forever()` pueden ser usados para hacer que el servidor empiece a aceptar conexiones.

Nuevo en la versión 3.7.

coroutine serve_forever()

Comienza a aceptar conexiones hasta que la corrutina sea cancelada. La cancelación de la tarea `serve_forever` hace que el servidor sea cerrado.

Este método puede ser llamado si el servidor ya está aceptando conexiones. Solamente una tarea `serve_forever` puede existir para un objeto `Server`.

Ejemplo:


```

async def client_connected(reader, writer):
    # Communicate with the client with
    # reader/writer streams. For example:
    await reader.readline()

async def main(host, port):
    srv = await asyncio.start_server(
        client_connected, host, port)
    await srv.serve_forever()

asyncio.run(main('127.0.0.1', 0))

```

Nuevo en la versión 3.7.

is_serving()

Retorna True si el servidor está aceptando nuevas conexiones.

Nuevo en la versión 3.7.

coroutine wait_closed()

Espera hasta que el método *close()* se complete.

sockets

Lista todos los objetos *socket.socket* en los que el servidor está escuchando.

Distinto en la versión 3.7: Antes de Python 3.7 *Server.sockets* solía retornar directamente una lista interna de servidores socket. En 3.7 se retorna una copia de esa lista.

Implementaciones del bucle de eventos

asyncio viene con dos implementaciones diferentes del bucle de eventos: *SelectorEventLoop* y *ProactorEventLoop*.

De manera predefinida *asyncio* está configurado para usar *SelectorEventLoop* en Unix y *ProactorEventLoop* en Windows.

class *asyncio.SelectorEventLoop*

Un bucle de eventos basado en el módulo *selectors*.

Usa el *selector* disponible mas eficiente para la plataforma dada. También es posible configurar manualmente la implementación exacta del selector a utilizar:

```

import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)

```

Disponibilidad: Unix, Windows.

class *asyncio.ProactorEventLoop*

Un bucle de eventos para Windows que usa «E/S Puertos de Finalización» (IOCP).

Disponibilidad: Windows.

Ver también:

Documentación de MSDN sobre E/S Puertos de Finalización.

class *asyncio.AbstractEventLoop*

Clase base abstracta para bucles de evento compatibles con *asyncio*.

La sección *Métodos del bucle de eventos* lista todos los métodos que como implementación alternativa de *AbstractEventLoop* debería haber estado definido.

Examples

Nótese que todos los ejemplos en esta sección muestran **a propósito** como usar las APIs de bucle de eventos de bajo nivel, como ser `loop.run_forever()` y `loop.call_soon()`. Aplicaciones `asyncio` modernas raramente necesitan ser escritas de esta manera; considere utilizar funciones de alto nivel como `asyncio.run()`.

Hola Mundo con `call_soon()`

Un ejemplo usando el método `loop.call_soon()` para planificar una llamada. La llamada muestra "Hello World" y luego para el bucle de eventos:

```
import asyncio

def hello_world(loop):
    """A callback to print 'Hello World' and stop the event loop"""
    print('Hello World')
    loop.stop()

loop = asyncio.get_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

Ver también:

Un ejemplo similar de *Hola Mundo* creado con una corrutina y la función `run()`.

Muestra la fecha actual con `call_later()`

Un ejemplo de llamada mostrando la fecha actual cada un segundo. La llamada usa el método `loop.call_later()` para volver a planificarse después de 5 segundos, y después para el bucle de eventos:

```
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.get_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

Ver también:

Un ejemplo similar a *fecha actual* creado con una corrutina y la función `run()`.

Mirar un descriptor de archivo para leer eventos

Espera hasta que el descriptor de archivo reciba algún dato usando el método `loop.add_reader()` y entonces cierra el bucle de eventos:

```
import asyncio
from socket import socketpair

# Create a pair of connected file descriptors
rsock, wsock = socketpair()

loop = asyncio.get_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())

    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)

    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # Run the event loop
    loop.run_forever()
finally:
    # We are done. Close sockets and the event loop.
    rsock.close()
    wsock.close()
    loop.close()
```

Ver también:

- Un *ejemplo* similar usando transportes, protocolos y el método `loop.create_connection()`.
- Otro *ejemplo* similar usando la función de alto nivel `asyncio.open_connection()` y transmisiones.

Establece los gestores de señal para SIGINT y SIGTERM

(Este ejemplo de `signals` solamente funcionan en Unix.)

Registra gestores para las señales SIGINT y SIGTERM usando el método `loop.add_signal_handler()`:

```
import asyncio
import functools
import os
import signal

def ask_exit(signame, loop):
    print("got signal %s: exit" % signame)
```

(continué en la próxima página)

(proviene de la página anterior)

```
loop.stop()

async def main():
    loop = asyncio.get_running_loop()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(
            getattr(signal, signame),
            functools.partial(ask_exit, signame, loop))

    await asyncio.sleep(3600)

print("Event loop running for 1 hour, press Ctrl+C to interrupt.")
print(f"pid {os.getpid()}: send SIGINT or SIGTERM to exit.")

asyncio.run(main())
```

18.1.8 Futures

Código fuente: `Lib/asyncio/futures.py`, `Lib/asyncio/base_futures.py`

Los objetos *Future* se utilizan para conectar **código basado en retrollamadas de bajo nivel** (*low-level callback-based code*) con código *async/await* de alto nivel.

Funciones Future

`asyncio.isfuture(obj)`

Retorna True si *obj* es uno de los siguientes:

- una instancia de `asyncio.Future`,
- una instancia de `asyncio.Task`,
- un objeto tipo Future con un atributo `_asyncio_future_blocking`.

Nuevo en la versión 3.5.

`asyncio.ensure_future(obj, *, loop=None)`

Retorna:

- el argumento *obj* inalterado, si *obj* es una *Future*, *Task*, o un objeto tipo Future (esto se puede verificar con `isfuture()`).
- un objeto *Task* envolviendo *obj*, si *obj* es una corrutina (esto se puede verificar con `iscoroutine()`); en este caso, la corrutina será programada por `ensure_future()`.
- un objeto *Task* que aguardará a *obj*, si *obj* es aguardable (esto se puede verificar con `inspect.isawaitable()`).

Si *obj* no es ninguno de los superiores, se lanzará `TypeError`.

Importante: Ver también la función `create_task()`, que es la forma preferida de crear nuevas *Tasks*.

Distinto en la versión 3.5.1: La función acepta cualquier objeto *awaitable*.

`asyncio.wrap_future(future, *, loop=None)`

Envuelve un objeto `concurrent.futures.Future` en un objeto `asyncio.Future`.

Objeto Future

class `asyncio.Future` (*, *loop=None*)

Un Future representa un resultado eventual de una operación asíncrona. No es seguro en hilos (*thread-safe*).

Future es un objeto *awaitable*. Las corrutinas pueden esperar (*await*) a objetos Future hasta que obtengan un resultado o excepción, o hasta que se cancelen.

Normalmente, los Futures se utilizan para permitir que código basado en retrollamadas de bajo nivel (*low-level callback-based code*) (por ejemplo, en protocolos implementados utilizando *asyncio transports*) interactúe con código *async/await* de alto nivel.

Es recomendable no exponer nunca objetos Future en APIs expuestas al usuario, y la forma recomendada de crear un objeto Future es llamando a `loop.create_future()`. De esta forma, implementaciones alternativas de bucles de eventos (*event loop*) pueden inyectar sus propias implementaciones optimizadas de un objeto Future.

Distinto en la versión 3.7: Añadido soporte para el módulo *contextvars*.

result ()

Retorna el resultado del Future.

Si el Future es *done* y tiene un resultado establecido por el método `set_result()`, el valor resultante es retornado.

Si el Future es *done* y tiene una excepción establecida por el método `set_exception()`, este método lanzará esta excepción.

Si un evento es *cancelled*, este método lanzará una excepción *CancelledError*.

Si el resultado del Future todavía no está disponible, este método lanzará una excepción *InvalidStateError*.

set_result (*result*)

Marca el Future como *done* y establece su resultado.

Lanza un error *InvalidStateError* si el Future ya está *done*.

set_exception (*exception*)

Marca el Future como *done* y establece una excepción.

Lanza un error *InvalidStateError* si el Future ya está *done*.

done ()

Retorna True si el Future está *done*.

Un Future está *done* si estaba *cancelled* o si tiene un resultado o excepción establecidos mediante llamadas a `set_result()` o `set_exception()`.

cancelled ()

Retorna True si el Future fue *cancelled*.

El método suele utilizarse para comprobar que un Future no es *cancelled* antes de establecer un resultado o excepción al mismo:

```
if not fut.cancelled():
    fut.set_result(42)
```

add_done_callback (*callback*, *, *context=None*)

Añade una retrollamada (*callback*) a ser ejecutada cuando el Future es *done*.

La retrollamada (*callback*) es llamada con el objeto Future como su único argumento.

Si el Future ya es *done* cuando se llama a este método, la retrollamada (*callback*) es programada con `loop.call_soon()`.

Un argumento opcional de contexto, por palabra clave, permite especificar un `contextvars.Context` personalizado para ser ejecutado en la retrollamada (*callback*). El contexto actual se utiliza cuando no se provee un contexto (*context*).

`functools.partial()` se puede utilizar para dar parámetros a la retrollamada (*callback*), por ejemplo:

```
# Call 'print("Future:", fut)' when "fut" is done.
fut.add_done_callback(
    functools.partial(print, "Future:"))
```

Distinto en la versión 3.7: El parámetro de contexto (*context*) por palabra clave fue añadido. Ver [PEP 567](#) para más detalles.

remove_done_callback (*callback*)

Elimina la retrollamada (*callback*) de la lista de retrollamadas.

Retorna el número de retrollamadas (*callbacks*) eliminadas, que normalmente es 1, excepto si una retrollamada fue añadida más de una vez.

cancel ()

Cancela el Future y programa retrollamadas (*callbacks*).

Si el Future ya está *done* o *cancelled*, retorna `False`. De lo contrario, cambia el estado del Future a *cancelled*, programa las retrollamadas, y retorna `True`.

exception ()

Retorna la excepción definida en este Future.

La excepción (o `None` si no se había establecido ninguna excepción) es retornada sólo si Future es *done*.

Si un evento es *cancelled*, este método lanzará una excepción `CancelledError`.

Si el Future todavía no es *done*, este método lanza una excepción `InvalidStateError`.

get_loop ()

Retorna el bucle de eventos (*event loop*) al cual el objeto Future está asociado.

Nuevo en la versión 3.7.

Este ejemplo crea un objeto Future, crea y programa una Task asíncrona para establecer el resultado para el Future, y espera hasta que el Future tenga un resultado:

```
async def set_after(fut, delay, value):
    # Sleep for *delay* seconds.
    await asyncio.sleep(delay)

    # Set *value* as a result of *fut* Future.
    fut.set_result(value)

async def main():
    # Get the current event loop.
    loop = asyncio.get_running_loop()

    # Create a new Future object.
    fut = loop.create_future()

    # Run "set_after()" coroutine in a parallel Task.
    # We are using the low-level "loop.create_task()" API here because
    # we already have a reference to the event loop at hand.
    # Otherwise we could have just used "asyncio.create_task()".
    loop.create_task(
        set_after(fut, 1, '... world'))

    print('hello ...')
```

(continué en la próxima página)

(proviene de la página anterior)

```
# Wait until *fut* has a result (1 second) and print it.
print(await fut)

asyncio.run(main())
```

Importante: El objeto Future fue diseñado para imitar a `concurrent.futures.Future`. Entre las principales diferencias están:

- al contrario que Futures de `asyncio`, las instancias de `concurrent.futures.Future` no son aguardables (`await`).
- `asyncio.Future.result()` y `asyncio.Future.exception()` no aceptan el argumento `timeout`.
- `asyncio.Future.result()` y `asyncio.Future.exception()` lanzan una excepción `InvalidStateError` cuando el Future no es `done`.
- Las retrollamadas (`callbacks`) registradas con `asyncio.Future.add_done_callback()` no son llamadas inmediatamente, sino que son programadas con `loop.call_soon()`.
- `asyncio.Future` no es compatible con las funciones `concurrent.futures.wait()` ni `concurrent.futures.as_completed()`.

18.1.9 Transportes y protocolos

Prefacio

Los transportes y protocolos son utilizados por las APIs de **bajo nivel** de los bucles de eventos, como `loop.create_connection()`. Utilizan un estilo de programación basado en retrollamadas y permiten implementaciones de alto rendimiento de protocolos de red o IPC (p. ej. HTTP).

Esencialmente, los transportes y protocolos solo deben usarse en bibliotecas y frameworks, nunca en aplicaciones `asyncio` de alto nivel.

Esta página de la documentación cubre tanto *Transports* como *Protocols*.

Introducción

En el nivel más alto, el transporte se ocupa de *cómo* se transmiten los bytes, mientras que el protocolo determina *qué* bytes transmitir (y hasta cierto punto cuándo).

Una forma diferente de decir lo mismo: Un transporte es una abstracción para un socket (o un punto final de E/S similar) mientras que un protocolo es una abstracción para una aplicación, desde el punto de vista del transporte.

Otro punto de vista más es que las interfaces de transporte y protocolo definen juntas una interfaz abstracta para usar E/S de red y E/S entre procesos.

Siempre existe una relación 1:1 entre el transporte y los objetos protocolo: el protocolo llama a los métodos del transporte para enviar datos, mientras que el transporte llama a los métodos del protocolo para enviarle los datos que se han recibido.

La mayoría de los métodos del bucle de eventos orientados a la conexión (como `loop.create_connection()`) aceptan generalmente un argumento `protocol_factory` que es usado para crear un objeto `Protocol` para una conexión aceptada, representada por un objeto `Transport`. Estos métodos suelen retornar una tupla de la forma (transporte, protocolo).

Contenidos

Esta página de la documentación contiene las siguientes secciones:

- La sección *Transports* documenta las clases `asyncio.BaseTransport`, `ReadTransport`, `WriteTransport`, `Transport`, `DatagramTransport` y `SubprocessTransport`.
- La sección *Protocols* documenta las clases `asyncio.BaseProtocol`, `Protocol`, `BufferedProtocol`, `DatagramProtocol` y `SubprocessProtocol`.
- La sección *Examples* muestra cómo trabajar con transportes, protocolos y las APIs de bajo nivel del bucle de eventos.

Transportes

Código fuente: `Lib/asyncio/transport.py`

Los transportes son clases proporcionadas por `asyncio` para abstraer varios tipos de canales de comunicación.

Los objetos transporte siempre son instanciados por un *bucle de eventos* `asyncio`.

`asyncio` implementa transportes para TCP, UDP, SSL y pipes de subprocesos. Los métodos disponibles en un transporte dependen del tipo de transporte.

Las clases transporte *no son seguras en hilos*.

Jerarquía de transportes

class `asyncio.BaseTransport`

Clase base para todos los transportes. Contiene métodos que todos los transportes `asyncio` comparten.

class `asyncio.WriteTransport` (*BaseTransport*)

Un transporte base para conexiones de solo escritura.

Las instancias de la clase `WriteTransport` se retornan desde el método del bucle de eventos `loop.connect_write_pipe()` y también se utilizan en métodos relacionados con subprocesos como `loop.subprocess_exec()`.

class `asyncio.ReadTransport` (*BaseTransport*)

Un transporte base para conexiones de solo lectura.

Las instancias de la clase `ReadTransport` se retornan desde el método del bucle de eventos `loop.connect_read_pipe()` y también se utilizan en métodos relacionados con subprocesos como `loop.subprocess_exec()`.

class `asyncio.Transport` (*WriteTransport*, *ReadTransport*)

Interfaz que representa un transporte bidireccional, como una conexión TCP.

El usuario no crea una instancia de transporte directamente; en su lugar se llama a una función de utilidad, pasándole una fábrica de protocolos junto a otra información necesaria para crear el transporte y el protocolo.

Las instancias de la clase `Transport` son retornadas o utilizadas por métodos del bucle de eventos como `loop.create_connection()`, `loop.create_unix_connection()`, `loop.create_server()`, `loop.sendfile()`, etc.

class `asyncio.DatagramTransport` (*BaseTransport*)

Un transporte para conexiones de datagramas (UDP).

Las instancias de la clase `DatagramTransport` se retornan desde el método `loop.create_datagram_endpoint()` del bucle de eventos.

class `asyncio.SubprocessTransport` (*BaseTransport*)

Una abstracción para representar una conexión entre un proceso padre y su proceso OS hijo.

Las instancias de la clase *SubprocessTransport* se retornan desde los métodos del bucle de eventos *loop.subprocess_shell()* y *loop.subprocess_exec()*.

Transporte base

`BaseTransport.close()`

Cierra el transporte.

Si el transporte tiene un búfer para datos de salida, los datos almacenados en el búfer se eliminarán de forma asíncrona. No se recibirán más datos. Después de que se vacíen todos los datos almacenados en el búfer, el método *protocol.connection_lost()* del protocolo se llamará con *None* como argumento.

`BaseTransport.is_closing()`

Retorna True si el transporte se está cerrando o está ya cerrado.

`BaseTransport.get_extra_info(name, default=None)`

Retorna información sobre el transporte o los recursos subyacentes que utiliza.

name es una cadena de caracteres que representa la información específica del transporte que se va a obtener.

default es el valor que se retornará si la información no está disponible o si el transporte no admite la consulta con la implementación del bucle de eventos de terceros dada o en la plataforma actual.

Por ejemplo, el siguiente código intenta obtener el objeto socket subyacente del transporte:

```
sock = transport.get_extra_info('socket')
if sock is not None:
    print(sock.getsockopt(...))
```

Categorías de información que se pueden consultar sobre algunos transportes:

- socket:
 - 'peername': la dirección remota a la que está conectado el socket, resultado de *socket.socket.getpeername()* (None en caso de error)
 - 'socket': instancia de *socket.socket*
 - 'sockname': la dirección propia del socket, resultado de *socket.socket.getsockname()*
- socket SSL:
 - 'compression': el algoritmo de compresión que se está usando como una cadena de caracteres o None si la conexión no está comprimida; resultado de *ssl.SSLSocket.compression()*
 - 'cipher': una tupla de tres valores que contiene el nombre del cifrado que se está utilizando, la versión del protocolo SSL que define su uso y la cantidad de bits de la clave secreta que se utilizan; resultado de *ssl.SSLSocket.cipher()*
 - 'peercert': certificado de pares; resultado de *ssl.SSLSocket.getpeercert()*
 - 'sslcontext': instancia de *ssl.SSLContext*
 - 'ssl_object': instancia de *ssl.SSLObject* o *ssl.SSLSocket*
- pipe:
 - 'pipe': objeto pipe
- subprocess:
 - 'subprocess': instancia de *subprocess.Popen*

`BaseTransport.set_protocol(protocol)`

Establece un nuevo protocolo.

El cambio de protocolo solo debe realizarse cuando esté documentado que ambos protocolos admiten el cambio.

`BaseTransport.get_protocol()`

Retorna el protocolo actual.

Transportes de solo lectura

`ReadTransport.is_reading()`

Retorna `True` si el transporte está recibiendo nuevos datos.

Nuevo en la versión 3.7.

`ReadTransport.pause_reading()`

Pausa el extremo receptor del transporte. No se pasarán datos al método `protocol.data_received()` del protocolo hasta que se llame a `resume_reading()`.

Distinto en la versión 3.7: El método es idempotente, es decir, se puede llamar cuando el transporte ya está en pausa o cerrado.

`ReadTransport.resume_reading()`

Reanuda el extremo receptor. El método `protocol.data_received()` del protocolo se llamará una vez más si hay algunos datos disponibles para su lectura.

Distinto en la versión 3.7: El método es idempotente, es decir, se puede llamar cuando el transporte está leyendo.

Transportes de solo escritura

`WriteTransport.abort()`

Cierra el transporte inmediatamente, sin esperar a que finalicen las operaciones pendientes. Se perderán los datos almacenados en el búfer. No se recibirán más datos. El método `protocol.connection_lost()` del protocolo será llamado eventualmente con `None` como argumento.

`WriteTransport.can_write_eof()`

Retorna `True` si el transporte admite `write_eof()`, en caso contrario `False`.

`WriteTransport.get_write_buffer_size()`

Retorna el tamaño actual del búfer de salida utilizado por el transporte.

`WriteTransport.get_write_buffer_limits()`

Obtiene los límites superior e inferior para el control del flujo de escritura. Retorna una tupla (`low`, `high`) donde `low` (“inferior”) y `high` (“superior”) son un número de bytes positivo.

Usa `set_write_buffer_limits()` para establecer los límites.

Nuevo en la versión 3.4.2.

`WriteTransport.set_write_buffer_limits(high=None, low=None)`

Establece los límites `high` (“superior”) y `low` (“inferior”) para el control del flujo de escritura.

Estos dos valores (medidos en número de bytes) controlan cuándo se llaman los métodos `protocol.pause_writing()` y `protocol.resume_writing()` del protocolo. Si se especifica, el límite inferior debe ser menor o igual que el límite superior. Ni `high` ni `low` pueden ser negativos.

`pause_writing()` se llama cuando el tamaño del búfer es mayor o igual que el valor `high` (“superior”). Si se ha pausado la escritura, se llama a `resume_writing()` cuando el tamaño del búfer es menor o igual que el valor `low` (“inferior”).

Los valores por defecto son específicos de la implementación. Si solo se proporciona el límite superior, el inferior toma de forma predeterminada un valor específico, dependiente de la implementación, menor o igual

que el límite superior. Establecer *high* (“superior”) en cero fuerza *low* (“inferior”) a cero también y hace que `pause_writing()` sea llamado siempre que el búfer no esté vacío. Establecer *low* (“inferior”) en cero hace que `resume_writing()` sea llamado únicamente cuando el búfer esté vacío. El uso de cero para cualquiera de los límites es generalmente subóptimo, ya que reduce las oportunidades para realizar E/S y cálculos simultáneamente.

Usa `get_write_buffer_limits()` para obtener los límites.

`WriteTransport.write(data)`

Escribe los bytes de *data* en el transporte.

Este método no bloquea; almacena los datos en el búfer y organiza que se envíen de forma asincrónica.

`WriteTransport.writelines(list_of_data)`

Escribe una lista (o cualquier iterable) de bytes de datos en el transporte. Esto es funcionalmente equivalente a llamar a `write()` en cada elemento generado por el iterable, pero puede ser implementado de manera más eficiente.

`WriteTransport.write_eof()`

Cierra el extremo de escritura del transporte después de vaciar todos los datos almacenados en el búfer. Aún es posible recibir datos.

Este método puede lanzar una excepción `NotImplementedError` si el transporte (p. ej. SSL) no soporta conexiones half-closed (“semi-cerradas”).

Transportes de datagramas

`DatagramTransport.sendto(data, addr=None)`

Envía los bytes *data* al par remoto proporcionado por *addr* (una dirección de destino dependiente del transporte). Si *addr* es *None*, los datos se envían a la dirección de destino proporcionada en la creación del transporte.

Este método no bloquea; almacena los datos en el búfer y organiza que se envíen de forma asincrónica.

`DatagramTransport.abort()`

Cierra el transporte inmediatamente, sin esperar a que finalicen las operaciones pendientes. Se perderán los datos almacenados en el búfer. No se recibirán más datos. El método `protocol.connection_lost()` del protocolo será llamado eventualmente con *None* como argumento.

Transportes de subprocessos

`SubprocessTransport.get_pid()`

Retorna la id del subprocesso como un número entero.

`SubprocessTransport.get_pipe_transport(fd)`

Retorna el transporte para la pipe de comunicación correspondiente al descriptor de archivo entero *fd*:

- 0: transporte de *streaming* para lectura de la entrada estándar (*stdin*) o *None* si el subprocesso no se creó con `stdin = PIPE`
- 1: transporte de *streaming* para escritura de la salida estándar (*stdout*) o *None* si el subprocesso no se creó con `stdout = PIPE`
- 2: transporte de *streaming* para escritura del error estándar (*stderr*) o *None* si el subprocesso no se creó con `stderr = PIPE`
- otro *fd*: *None*

`SubprocessTransport.get_returncode()`

Retorna el código de retorno del subprocesso como un entero o *None* si no ha retornado aún, lo que es similar al atributo `subprocess.Popen.returncode`.

`SubprocessTransport.kill()`

Mata al subprocesso.

En los sistemas POSIX, la función envía SIGKILL al subprocesso. En Windows, este método es un alias para `terminate()`.

Ver también `subprocess.Popen.kill()`.

`SubprocessTransport.send_signal(signal)`

Envía el número de *señal* al subprocesso, como en `subprocess.Popen.send_signal()`.

`SubprocessTransport.terminate()`

Detiene el subprocesso.

En los sistemas POSIX, este método envía SIGTERM al subprocesso. En Windows, se llama a la función de la API de Windows `TerminateProcess()` para detener el subprocesso.

Ver también `subprocess.Popen.terminate()`.

`SubprocessTransport.close()`

Mata al subprocesso llamando al método `kill()`.

Si el subprocesso aún no ha retornado, cierra los transportes de las pipes `stdin`, `stdout` y `stderr`.

Protocolos

Código fuente: `Lib/asyncio/protocols.py`

asyncio proporciona un conjunto de clases base abstractas que pueden usarse para implementar protocolos de red. Estas clases están destinadas a ser utilizadas junto con los *transportes*.

Las subclases de las clases abstractas de protocolos base pueden implementar algunos o todos los métodos. Todos estos métodos son retrollamadas: son llamados por los transportes en ciertos eventos, por ejemplo, cuando se reciben algunos datos. Un método del protocolo base debe ser llamado por el transporte correspondiente.

Protocolos base

class `asyncio.BaseProtocol`

Protocolo base con métodos que comparten todos los demás protocolos.

class `asyncio.Protocol` (*BaseProtocol*)

La clase base para implementar protocolos de *streaming* (TCP, sockets Unix, etc).

class `asyncio.BufferedProtocol` (*BaseProtocol*)

Una clase base para implementar protocolos de *streaming* con control manual del búfer de recepción.

class `asyncio.DatagramProtocol` (*BaseProtocol*)

La clase base para implementar protocolos de datagramas (UDP).

class `asyncio.SubprocessProtocol` (*BaseProtocol*)

La clase base para implementar protocolos que se comunican con procesos secundarios (pipes unidireccionales).

Protocolo base

Todos los protocolos asyncio pueden implementar las retrollamadas del protocolo base.

Retrollamadas de conexión

Las retrollamadas de conexión son llamadas exactamente una vez por conexión establecida en todos los protocolos. Todas las demás retrollamadas del protocolo solo pueden ser llamadas entre estos dos métodos.

`BaseProtocol.connection_made(transport)`

Se llama cuando se establece una conexión.

El argumento *transport* es el transporte que representa la conexión. El protocolo se encarga de almacenar la referencia a su propio transporte.

`BaseProtocol.connection_lost(exc)`

Se llama cuando la conexión se pierde o se cierra.

El argumento es un objeto excepción o *None*. Esto último significa que se recibió un EOF regular o que la conexión fue cancelada o cerrada por este lado de la conexión.

Retrollamadas de control de flujo

Los transportes pueden llamar a las retrollamadas de control de flujo para pausar o reanudar la escritura llevada a cabo por el protocolo.

Consulta la documentación del método `set_write_buffer_limits()` para obtener más detalles.

`BaseProtocol.pause_writing()`

Se llama cuando el búfer del transporte supera el límite superior.

`BaseProtocol.resume_writing()`

Se llama cuando el búfer del transporte se vacía por debajo del límite inferior.

Si el tamaño del búfer es igual al límite superior, `pause_writing()` no será llamado: el tamaño del búfer debe superarse estrictamente.

Por el contrario, se llama a `resume_writing()` cuando el tamaño del búfer es igual o menor que el límite inferior. Estas condiciones finales son importantes para garantizar que todo salga como se espera cuando cualquiera de los dos límites sea cero.

Protocolos de *streaming*

Los métodos de eventos, como `loop.create_server()`, `loop.create_unix_server()`, `loop.create_connection()`, `loop.create_unix_connection()`, `loop.connect_accepted_socket()`, `loop.connect_read_pipe()`, y `loop.connect_write_pipe()` aceptan fábricas que retornan protocolos de *streaming*.

`Protocol.data_received(data)`

Se llama cuando se reciben algunos datos. *data* es un objeto bytes no vacío que contiene los datos entrantes.

Que los datos se almacenen en un búfer, que se fragmenten o se vuelvan a ensamblar depende del transporte. En general, no debe confiar en semánticas específicas y, en cambio, hacer que su análisis sea genérico y flexible. Sin embargo, los datos siempre se reciben en el orden correcto.

El método se puede llamar un número arbitrario de veces mientras una conexión esté abierta.

Sin embargo, `protocol.eof_received()` se llama como máximo una vez. En el momento que se llama a `eof_received()`, ya no se llama a `data_received()`.

`Protocol.eof_received()`

Se llama cuando el otro extremo indica que no enviará más datos (por ejemplo, llamando a `transport.write_eof()` si el otro extremo también usa *asyncio*).

Este método puede retornar un valor falso (incluido *None*), en cuyo caso el transporte se cerrará solo. Por el contrario, si este método retorna un valor verdadero, el protocolo utilizado determina si se debe cerrar el transporte. Dado que la implementación por defecto retorna *None*, en éste caso, se cierra implícitamente la conexión.

Algunos transportes, incluido SSL, no admiten conexiones half-closed (“semi-cerradas”), en cuyo caso retornar verdadero desde este método resultará en el cierre de la conexión.

Máquina de estado:

```
start -> connection_made
      [-> data_received]*
      [-> eof_received]?
-> connection_lost -> end
```

Protocolos de *streaming* mediante búfer

Nuevo en la versión 3.7: **Importante:** ¡esto se ha agregado a `asyncio` en Python 3.7 *de forma provisional!* Debe tratarse como una API experimental que puede cambiarse o eliminarse por completo en Python 3.8.

Los protocolos que hacen uso de un búfer se pueden utilizar con cualquier método del bucle de eventos que admita *Streaming Protocols*.

Las implementaciones de `BufferedProtocol` permiten la asignación manual explícita y el control del búfer de recepción. Los bucles de eventos pueden utilizar el búfer proporcionado por el protocolo para evitar copias de datos innecesarias. Esto puede resultar en una mejora notable del rendimiento de los protocolos que reciben grandes cantidades de datos. Las implementaciones de protocolos sofisticados pueden reducir significativamente la cantidad de asignaciones de búfer.

Las siguientes retrollamadas son llamadas en instancias `BufferedProtocol`:

`BufferedProtocol.get_buffer(sizehint)`

Se llama para asignar un nuevo búfer de recepción.

sizehint es el tamaño mínimo recomendado para el búfer retornado. Es aceptable retornar búferes más pequeños o más grandes de lo que sugiere *sizehint*. Cuando se establece en -1, el tamaño del búfer puede ser arbitrario. Es un error retornar un búfer con tamaño cero.

`get_buffer()` debe retornar un objeto que implemente el protocolo de búfer.

`BufferedProtocol.buffer_updated(nbytes)`

Se llama cuando el búfer se ha actualizado con los datos recibidos.

nbytes es el número total de bytes que se escribieron en el búfer.

`BufferedProtocol.eof_received()`

Consulte la documentación del método `protocol.eof_received()`.

`get_buffer()` se puede llamar un número arbitrario de veces durante una conexión. Sin embargo, `protocol.eof_received()` se llama como máximo una vez y, si se llama, `get_buffer()` y `buffer_updated()` no serán llamados después de eso.

Máquina de estado:

```
start -> connection_made
      [-> get_buffer
        [-> buffer_updated]?
      ]*
      [-> eof_received]?
-> connection_lost -> end
```

Protocolos de datagramas

Las instancias del protocolo de datagramas deben ser construidas por fábricas de protocolos pasadas al método `loop.create_datagram_endpoint()`.

`DatagramProtocol.datagram_received(data, addr)`

Se llama cuando se recibe un datagrama. *data* es un objeto bytes que contiene los datos entrantes. *addr* es la dirección del par que envía los datos; el formato exacto depende del transporte.

`DatagramProtocol.error_received(exc)`

Se llama cuando una operación de envío o recepción anterior genera una `OSError`. *exc* es la instancia `OSError`.

Este método se llama en condiciones excepcionales, cuando el transporte (por ejemplo, UDP) detecta que un datagrama no se pudo entregar a su destinatario. Sin embargo, en la mayoría de casos, los datagramas que no se puedan entregar se eliminarán silenciosamente.

Nota: En los sistemas BSD (macOS, FreeBSD, etc.) el control de flujo no es compatible con los protocolos de datagramas, esto se debe a que no hay una forma confiable de detectar fallos de envío causados por escribir demasiados paquetes.

El socket siempre aparece como disponible (“ready”) y se eliminan los paquetes sobrantes. Un error `OSError` con `errno` establecido en `errno.ENOBUFS` puede o no ser generado; si se genera, se informará a `DatagramProtocol.error_received()` pero en caso contrario se ignorará.

Protocolos de subprocessos

Las instancias del protocolo de datagramas deben ser construidas por fábricas de protocolos pasadas a los métodos `loop.subprocess_exec()` y `loop.subprocess_shell()`.

`SubprocessProtocol.pipe_data_received(fd, data)`

Se llama cuando el proceso hijo escribe datos en su pipe stdout o stderr.

fd es el descriptor de archivo entero de la pipe.

data es un objeto bytes no vacío que contiene los datos recibidos.

`SubprocessProtocol.pipe_connection_lost(fd, exc)`

Se llama cuando se cierra una de las pipes que se comunican con el proceso hijo.

fd es el descriptor de archivo entero que se cerró.

`SubprocessProtocol.process_exited()`

Se llama cuando el proceso hijo ha finalizado.

Ejemplos

Servidor de eco TCP

Crear un servidor de eco TCP usando el método `loop.create_server()`, enviar de vuelta los datos recibidos y cerrar la conexión:

```
import asyncio

class EchoServerProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
```

(continué en la próxima página)

(proviene de la página anterior)

```
self.transport = transport

def data_received(self, data):
    message = data.decode()
    print('Data received: {!r}'.format(message))

    print('Send: {!r}'.format(message))
    self.transport.write(data)

    print('Close the client socket')
    self.transport.close()

async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    server = await loop.create_server(
        lambda: EchoServerProtocol(),
        '127.0.0.1', 8888)

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

Ver también:

El ejemplo *Servidor de eco TCP usando streams* usa la función de alto nivel `asyncio.start_server()`.

Cliente de eco TCP

Un cliente de eco TCP usando el método `loop.create_connection()`, envía datos y espera hasta que la conexión se cierre:

```
import asyncio

class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        self.on_con_lost.set_result(True)

async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
```

(continué en la próxima página)

(proviene de la página anterior)

```

loop = asyncio.get_running_loop()

on_con_lost = loop.create_future()
message = 'Hello World!'

transport, protocol = await loop.create_connection(
    lambda: EchoClientProtocol(message, on_con_lost),
    '127.0.0.1', 8888)

# Wait until the protocol signals that the connection
# is lost and close the transport.
try:
    await on_con_lost
finally:
    transport.close()

asyncio.run(main())

```

Ver también:

El ejemplo *Cliente de eco TCP usando streams* usa la función de alto nivel `asyncio.open_connection()`.

Servidor de eco UDP

Un servidor de eco UDP, usando el método `loop.create_datagram_endpoint()`, envía de vuelta los datos recibidos:

```

import asyncio

class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)

async def main():
    print("Starting UDP server")

    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    # One protocol instance will be created to serve all
    # client requests.
    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoServerProtocol(),
        local_addr=('127.0.0.1', 9999))

    try:
        await asyncio.sleep(3600) # Serve for 1 hour.
    finally:
        transport.close()

```

(continué en la próxima página)

(proviene de la página anterior)

```
asyncio.run(main())
```

Cliente de eco UDP

Un cliente de eco UDP, usando el método `loop.create_datagram_endpoint()`, envía datos y cierra el transporte cuando recibe la respuesta:

```
import asyncio

class EchoClientProtocol:
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Connection closed")
        self.on_con_lost.set_result(True)

async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = "Hello World!"

    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoClientProtocol(message, on_con_lost),
        remote_addr=('127.0.0.1', 9999))

    try:
        await on_con_lost
    finally:
        transport.close()

asyncio.run(main())
```

Conectando sockets existentes

Espera hasta que un socket reciba datos usando el método `loop.create_connection()` mediante un protocolo:

```
import asyncio
import socket

class MyProtocol(asyncio.Protocol):

    def __init__(self, on_con_lost):
        self.transport = None
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())

        # We are done: close the transport;
        # connection_lost() will be called automatically.
        self.transport.close()

    def connection_lost(self, exc):
        # The socket has been closed
        self.on_con_lost.set_result(True)

async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()
    on_con_lost = loop.create_future()

    # Create a pair of connected sockets
    rsock, wsock = socket.socketpair()

    # Register the socket to wait for data.
    transport, protocol = await loop.create_connection(
        lambda: MyProtocol(on_con_lost), sock=rsock)

    # Simulate the reception of data from the network.
    loop.call_soon(wsock.send, 'abc'.encode())

    try:
        await protocol.on_con_lost
    finally:
        transport.close()
        wsock.close()

asyncio.run(main())
```

Ver también:

El ejemplo *monitorizar eventos de lectura en un descriptor de archivo* utiliza el método de bajo nivel `loop.add_reader()` para registrar un descriptor de archivo.

El ejemplo *registrar un socket abierto a la espera de datos usando streams* usa *streams* de alto nivel creados por la función `open_connection()` en una corrutina.

loop.subprocess_exec() y *SubprocessProtocol*

Un ejemplo de un protocolo de subprocesso que se utiliza para obtener la salida de un subprocesso y esperar su terminación.

The subprocess is created by the `loop.subprocess_exec()` method:

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exit_future.set_result(True)

async def get_date():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by DateProtocol;
    # redirect the standard output into a pipe.
    transport, protocol = await loop.subprocess_exec(
        lambda: DateProtocol(exit_future),
        sys.executable, '-c', code,
        stdin=None, stderr=None)

    # Wait for the subprocess exit using the process_exited()
    # method of the protocol.
    await exit_future

    # Close the stdout pipe.
    transport.close()

    # Read the output which was collected by the
    # pipe_data_received() method of the protocol.
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

Consulte también el *mismo ejemplo* escrito utilizando la API de alto nivel.

18.1.10 Políticas

Una política del bucle de eventos es un objeto por proceso que controla la administración del bucle de eventos. Cada bucle de eventos tiene una política por defecto, que puede ser cambiada y editada usando la política de API.

Una política define la noción de *contexto* y administra un bucle de eventos separado por contexto. La política por defecto define un *contexto* como el estado actual.

Usando una política de bucle de eventos personalizada, la conducta de las funciones `get_event_loop()`, `set_event_loop()`, y `new_event_loop()` puede ser personalizada.

Los objetos de política deberían implementar las APIs definidas en la clase abstracta base `AbstractEventLoopPolicy`.

Obteniendo y Configurando la Política

Las siguientes funciones pueden ser usadas para obtener y configurar la política de los procesos actuales:

```
asyncio.get_event_loop_policy()
    Retorna la política actual en todo el proceso.

asyncio.set_event_loop_policy(policy)
    Establece la política actual en todo el proceso a policy.

    Si policy está configurado a None, la política por defecto se reestablece.
```

Objetos de Política

La clase base de política de bucle de eventos abstractos se define de la siguiente manera:

```
class asyncio.AbstractEventLoopPolicy
    Una clase base abstracta para políticas asyncio.

    get_event_loop()
        Retorna el bucle de eventos para el contexto actual.

        Retorna un objeto bucle de eventos implementando la interfaz AbstractEventLoop.

        Este método nunca debería retornar None.

        Distinto en la versión 3.6.

    set_event_loop(loop)
        Establece el bucle de eventos para el contexto a loop.

    new_event_loop()
        Crea y retorna un nuevo objeto de bucle de eventos.

        Este método nunca debería retornar None.

    get_child_watcher()
        Retorna un objeto observador de procesos secundarios.

        Retorna un objeto observador implementando la interfaz AbstractChildWatcher.

        Esta función es específica de Unix.

    set_child_watcher(watcher)
        Establece el observador de procesos secundarios actuales a watcher.

        Esta función es específica de Unix.
```

`asyncio` se envía con las siguientes políticas integradas:

class `asyncio.DefaultEventLoopPolicy`

La política por defecto `asyncio`. Usa `SelectorEventLoop` en Unix y `ProactorEventLoop` en Windows.

No hay necesidad de instalar la política por defecto manualmente. `asyncio` está configurado para usar la política por defecto automáticamente.

Distinto en la versión 3.8: En Windows, `ProactorEventLoop` ahora se usa por defecto.

class `asyncio.WindowsSelectorEventLoopPolicy`

Una política de bucle de eventos alternativa que usa la implementación de bucle de eventos `SelectorEventLoop`.

Disponibilidad: Windows.

class `asyncio.WindowsProactorEventLoopPolicy`

Una política de bucle de eventos alternativa que usa la implementación de bucle de eventos `ProactorEventLoop`.

Disponibilidad: Windows.

Observadores de procesos

Un observador de procesos permite personalizar cómo un bucle de eventos monitorea procesos secundarios en Unix. Específicamente, un bucle de eventos necesita saber cuándo un proceso secundario ha terminado.

En `asyncio`, los procesos secundarios son creados con las funciones `create_subprocess_exec()` y `loop.subprocess_exec()`.

`asyncio` define la clase base abstracta `AbstractChildWatcher`, qué observadores de subprocessos deberían implementarse, y tiene cuatro implementaciones diferentes: `ThreadedChildWatcher` (configurado para ser usado por defecto), `MultiLoopChildWatcher`, `SaferChildWatcher`, y `FastChildWatcher`.

Mirar también la sección *Subprocesos e hilos*.

Las siguientes dos funciones pueden ser usadas para personalizar la implementación de observadores de procesos secundarios usados por el bucle de eventos de `asyncio`:

`asyncio.get_child_watcher()`

Retorna el observador de procesos secundarios para la política actual.

`asyncio.set_child_watcher(watcher)`

Establece el observador de procesos secundarios actuales a `watcher` para la política actual. `watcher` debe implementar métodos definidos en la clase base `AbstractChildWatcher`.

Nota: Implementaciones de bucles de eventos de terceras partes no deben dar soporte a observadores de procesos secundarios personalizados. Para dichos bucles de eventos, usando `set_child_watcher()` podría estar prohibido o no tener efecto.

class `asyncio.AbstractChildWatcher`

add_child_handler (*pid*, *callback*, **args*)

Registra un nuevo gestor de proceso secundario.

Arreglo para `callback(pid, returncode, *args)` a ser invocado cuando un proceso con PID igual a `pid` termina. Especificando otro retrollamada para el mismo proceso reemplaza el gestor previo.

El `callback` invocable debe ser seguro para hilos.

remove_child_handler (*pid*)

Remueve el gestor para el proceso con PID igual a `pid`.

La función retorna `True` si el gestor fue removido de forma exitosa, `False` si no hubo nada que remover.

attach_loop (*loop*)

Adjunta el observador a un bucle de eventos.

Si el observador estaba previamente adjuntado a un bucle de eventos, entonces primero es separado antes de adjuntar el nuevo bucle.

Nota: el bucle puede ser `None`.

is_active ()

Retorna `True` si el observador está listo para usarse.

Generar un nuevo subproceso con observador de procesos secundarios actual *inactive* lanza `RuntimeError`.

Nuevo en la versión 3.8.

close ()

Cierra el observador.

Este método tiene que ser invocado para asegurar que los objetos subyacentes están limpiados.

class `asyncio.ThreadedChildWatcher`

Esta implementación inicia un nuevo hilo esperando para cada subproceso generado.

Trabaja de manera confiable incluso cuando el bucle de eventos `asyncio` se ejecuta en un hilo de SO no principal.

There is no noticeable overhead when handling a big number of children ($O(1)$ each time a child terminates), but starting a thread per process requires extra memory.

Este observador es usado por defecto.

Nuevo en la versión 3.8.

class `asyncio.MultiLoopChildWatcher`

This implementation registers a `SIGCHLD` signal handler on instantiation. That can break third-party code that installs a custom handler for `SIGCHLD` signal.

El observador evita interrumpir otro código generando procesos sondeando cada proceso explícitamente en una señal `SIGCHLD`.

No hay limitación para ejecutar subprocesos de diferentes hilos una vez el observador es instalado.

La solución es segura pero tiene una sobrecarga significativa cuando se gestiona un número grande de procesos ($O(n)$ cada vez que un `SIGCHLD` es recibido).

Nuevo en la versión 3.8.

class `asyncio.SafeChildWatcher`

Esta implementación usa bucles de eventos activos del hilo principal para gestionar la señal `SIGCHLD`. Si el hilo principal no tiene bucles de eventos en ejecución otro hilo no puede generar un subproceso (`RuntimeError` es disparada).

El observador evita interrumpir otro código generando procesos sondeando cada proceso explícitamente en una señal `SIGCHLD`.

Esta solución es tan segura como `MultiLoopChildWatcher` y tiene la misma complejidad $O(n)$ pero requiere de un bucle de eventos ejecutándose en el hilo principal para trabajar.

class `asyncio.FastChildWatcher`

Esta implementación cosecha cada proceso terminado llamando `os.waitpid(-1)` directamente, posiblemente rompiendo otro código generando procesos y esperando por su terminación.

No hay sobrecarga notable cuando se gestiona un número grande de procesos secundarios ($O(1)$ cada vez que un proceso secundario termina).

Esta solución requiere un bucle de eventos ejecutándose en el hilo principal para trabajar, como `SafeChildWatcher`.

Personalizar Políticas

Para implementar una nueva política de bucle de eventos, se recomienda heredar `DefaultEventLoopPolicy` y sobrescribir los métodos para los cuales se desea una conducta personalizada, por ejemplo:

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

18.1.11 Soporte de plataforma

El módulo `asyncio` está diseñado para ser portátil, pero algunas plataformas tienen diferencias y limitaciones sutiles debido a la arquitectura y las capacidades subyacentes de las plataformas.

Todas las Plataformas

- `loop.add_reader()` y `loop.add_writer()` no se pueden utilizar para supervisar la E/S del archivo.

Windows

Código fuente: `Lib/asyncio/proactor_events.py`, `Lib/asyncio/windows_events.py`, `Lib/asyncio/windows_utils.py`

Distinto en la versión 3.8: En Windows, `ProactorEventLoop` es ahora el bucle de eventos predeterminado.

Todos los bucles de eventos en Windows no admiten los métodos siguientes:

- `loop.create_unix_connection()` y `loop.create_unix_server()` no son compatibles. La familia de sockets `socket.AF_UNIX` es específica de Unix.
- `loop.add_signal_handler()` y `loop.remove_signal_handler()` no son compatibles.

`SelectorEventLoop` tiene las siguientes limitaciones:

- `SelectSelector` se utiliza para esperar los eventos de los sockets: soporta los sockets y está limitado a 512 sockets.
- `loop.add_reader()` y `loop.add_writer()` sólo aceptan manejadores de sockets (por ejemplo, los descriptores de archivos de tuberías no están soportados).
- Las tuberías no están soportadas, por lo que los métodos `loop.connect_read_pipe()` y `loop.connect_write_pipe()` no están implementados.
- `Subprocesos` no están soportados, es decir, los métodos `loop.subprocess_exec()` y `loop.subprocess_shell()` no están implementados.

`ProactorEventLoop` tiene las siguientes limitaciones:

- Los métodos `loop.add_reader()` y `loop.add_writer()` no están soportados.

La resolución del reloj monótono de Windows suele ser de unos 15,6 mseg. La mejor resolución es de 0,5 mseg. La resolución depende del hardware (disponibilidad de HPET) y de la configuración de Windows.

Soporte de sub-procesos en Windows

En Windows, el bucle de eventos por defecto *ProactorEventLoop* soporta subprocesos, mientras que *SelectorEventLoop* no lo hace.

La función *policy.set_child_watcher()* tampoco está soportada, ya que *ProactorEventLoop* tiene un mecanismo diferente para vigilar los procesos hijos.

macOS

Las versiones modernas de MacOS son totalmente compatibles.

macOS <= 10.8

En macOS 10.6, 10.7 y 10.8, el bucle de eventos por defecto utiliza *selectors.KqueueSelector*, que no soporta dispositivos de caracteres en estas versiones. El *SelectorEventLoop* puede ser configurado manualmente para usar *SelectSelector* o *PollSelector* para soportar dispositivos de caracteres en estas versiones antiguas de macOS. Ejemplo:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

18.1.12 Índice de API de alto nivel

Esta página lista todas las APIs *async/await* habilitadas de alto nivel.

Tareas

Utilidades para ejecutar programas *asyncio*, crear Tareas y esperar a varias cosas con tiempos de expiración.

<i>run()</i>	Crea un loop de eventos, ejecuta una corrutina, cierra el loop.
<i>create_task()</i>	Lanza una Tarea <i>asyncio</i> .
<i>await sleep()</i>	Duerme por un número de segundos.
<i>await gather()</i>	Programa y espera por cosas concurrentemente.
<i>await wait_for()</i>	Ejecuta con un tiempo de expiración.
<i>await shield()</i>	Protege de la cancelación.
<i>await wait()</i>	Monitorea la completitud.
<i>current_task()</i>	Retorna la Tarea actual.
<i>all_tasks()</i>	Retorna todas las tareas para un loop de eventos.
<i>Task</i>	Objeto Tarea.
<i>run_coroutine_threadsafe()</i>	Programa una corrutina de desde otro hilo del sistema operativo.
<i>for in as_completed()</i>	Monitorea por completitud con un loop <i>for</i> .

Ejemplos

- Usando `asyncio.gather()` para ejecutar cosas en paralelo.
- Usando `asyncio.wait_for()` para forzar un tiempo de expiración.
- Cancelación.
- Usando `asyncio.sleep()`.
- Ver también la [página principal de documentación de Tareas](#).

Colas

Las colas deberían ser usadas para distribuir trabajo entre múltiples Tareas `asyncio`, implementar pools de conexiones y patrones pub/sub.

<code>Queue</code>	Una cola FIFO.
<code>PriorityQueue</code>	Una cola de prioridad.
<code>LifoQueue</code>	Una cola LIFO.

Ejemplos

- Usando `asyncio.Queue` para distribuir carga de trabajo entre varias Tareas.
- Ver también la [página de documentación de Colas](#).

Subprocesos

Utilidades para generar subprocessos y ejecutar comandos de consola.

<code>await create_subprocess_exec()</code>	Crea un subprocesso.
<code>await create_subprocess_shell()</code>	Ejecuta un comando de consola.

Ejemplos

- Ejecutando un comando de consola.
- Ver también la documentación de las [APIs de subprocessos](#).

Flujos

APIs de alto nivel para trabajar con IO de red.

<code>await open_connection()</code>	Establece una conexión TCP.
<code>await open_unix_connection()</code>	Establece una conexión de un socket Unix.
<code>await start_server()</code>	Lanza un servidor TCP.
<code>await start_unix_server()</code>	Lanza un servidor de sockets Unix.
<code>StreamReader</code>	Objeto de alto nivel <code>async/await</code> para recibir datos de red.
<code>StreamWriter</code>	Objeto de alto nivel <code>async/await</code> para enviar datos de red.

Ejemplos

- *Cliente TCP de ejemplo.*
- Ver también la documentación de *APIs de flujos*.

Sincronización

Primitivas de sincronización al estilo hilos que pueden ser usadas en Tareas.

<code>Lock</code>	Un bloqueo mutex.
<code>Event</code>	Un objeto de evento.
<code>Condition</code>	Un objeto de condición.
<code>Semaphore</code>	Un semáforo.
<code>BoundedSemaphore</code>	Un semáforo acotado.

Ejemplos

- *Usando `asyncio.Event`.*
- Ver también la documentación de las *primitivas de sincronización* de `asyncio`.

Excepciones

<code>asyncio.TimeoutError</code>	Lanzado en tiempos de expiración por funciones como <code>wait_for()</code> . Ten en mente que <code>asyncio.TimeoutError</code> no está relacionada con la excepción predefinida <code>TimeoutError</code> .
<code>asyncio.CancelledError</code>	Lanzada cuando una Tarea es cancelada. Ver también <code>Task.cancel()</code> .

Ejemplos

- *Gestionando `CancelledError` para ejecutar código en petición de cancelación.*
- Ver también la lista completa de *excepciones específicas de `asyncio`*.

18.1.13 Índice de API de bajo nivel

Esta página enumera todas las APIs de `asyncio` de bajo nivel.

Obtención del bucle de eventos

<code>asyncio.get_running_loop()</code>	La función preferida para obtener el bucle de eventos en ejecución.
<code>asyncio.get_event_loop()</code>	Obtiene una instancia del bucle de eventos (actual o mediante la política del bucle).
<code>asyncio.set_event_loop()</code>	Establece el bucle de eventos como actual a través de la política del bucle.
<code>asyncio.new_event_loop()</code>	Crea un nuevo bucle de eventos.

Ejemplos

- Usando `asyncio.get_running_loop()`.

Métodos del bucle de eventos

Consulte también la sección de la documentación principal sobre los *métodos del bucle de eventos*.

Ciclo de vida

<code>loop.run_until_complete()</code>	Ejecuta un Future/Tarea/aguadable (<i>awaitable</i>) hasta que se complete.
<code>loop.run_forever()</code>	Ejecuta el bucle de eventos para siempre.
<code>loop.stop()</code>	Detiene el bucle de eventos.
<code>loop.close()</code>	Cierra el bucle de eventos.
<code>loop.is_running()</code>	Retorna <code>True</code> si el bucle de eventos se está ejecutando.
<code>loop.is_closed()</code>	Retorna <code>True</code> si el bucle de eventos está cerrado.
<code>await loop.shutdown_asyncgens()</code>	Cierra generadores asíncronos.

Depuración

<code>loop.set_debug()</code>	Habilita o deshabilita el modo de depuración.
<code>loop.get_debug()</code>	Obtiene el modo de depuración actual.

Programación de devoluciones de llamada

<code>loop.call_soon()</code>	Invoca una devolución de llamada <i>soon</i> .
<code>loop.call_soon_threadsafe()</code>	Una variante segura para subprocesos de <code>loop.call_soon()</code> .
<code>loop.call_later()</code>	Invoca una devolución de llamada <i>después</i> del tiempo especificado.
<code>loop.call_at()</code>	Invoca una devolución de llamada <i>en</i> el tiempo especificado.

Hilo/Grupo de procesos

<code>await loop.run_in_executor()</code>	Ejecuta una función de bloqueo vinculada a la CPU o de otro tipo en un ejecutor <code>concurrent.futures</code> .
<code>loop.set_default_executor()</code>	Establece el ejecutor predeterminado para <code>loop.run_in_executor()</code> .

Tareas y Futures

<code>loop.create_future()</code>	Crea un objeto <i>Future</i> .
<code>loop.create_task()</code>	Programa una corrutina como <i>Task</i> .
<code>loop.set_task_factory()</code>	Establece una fábrica utilizada por <code>loop.create_task()</code> para crear <i>Tareas</i> .
<code>loop.get_task_factory()</code>	Obtiene la fábrica <code>loop.create_task()</code> que se usa para crear <i>Tareas</i> .

DNS

<code>await loop.getaddrinfo()</code>	Versión asincrónica de <code>socket.getaddrinfo()</code> .
<code>await loop.getnameinfo()</code>	Versión asincrónica de <code>socket.getnameinfo()</code> .

Redes e IPC

<code>await loop.create_connection()</code>	Abre una conexión TCP.
<code>await loop.create_server()</code>	Crea un servidor TCP.
<code>await loop.create_unix_connection()</code>	Abre una conexión de socket Unix.
<code>await loop.create_unix_server()</code>	Crea un servidor socket de Unix.
<code>await loop.connect_accepted_socket()</code>	Envuelve un <code>socket</code> en un par (transport, protocol).
<code>await loop.create_datagram_endpoint()</code>	Abre una conexión de datagramas (UDP).
<code>await loop.sendfile()</code>	Envía un archivo a través del transporte.
<code>await loop.start_tls()</code>	Actualiza una conexión existente a TLS.
<code>await loop.connect_read_pipe()</code>	Envuelve el fin de lectura de <i>pipe</i> en un par (transport, protocol).
<code>await loop.connect_write_pipe()</code>	Envuelve el fin de escritura de <i>pipe</i> en un par (transport, protocol).

Sockets

<code>await loop.sock_recv()</code>	Recibe datos de <code>socket</code> .
<code>await loop.sock_recv_into()</code>	Recibe datos de <code>socket</code> en un buffer.
<code>await loop.sock_sendall()</code>	Envía datos a <code>socket</code> .
<code>await loop.sock_connect()</code>	Conecta con <code>socket</code> .
<code>await loop.sock_accept()</code>	Acepta una conexión <code>socket</code> .
<code>await loop.sock_sendfile()</code>	Envía un archivo a través de <code>socket</code> .
<code>loop.add_reader()</code>	Comienza a monitorear un descriptor de archivo para ver la disponibilidad de lectura.
<code>loop.remove_reader()</code>	Detiene el monitoreo del descriptor de archivo para ver la disponibilidad de lectura.
<code>loop.add_writer()</code>	Comienza a monitorear un descriptor de archivo para ver la disponibilidad de escritura.
<code>loop.remove_writer()</code>	Detiene el monitoreo del descriptor de archivo para ver la disponibilidad de escritura.

Señales Unix

<code>loop.add_signal_handler()</code>	Añade un gestor para <i>signal</i> .
<code>loop.remove_signal_handler()</code>	Elimina un gestor para <i>signal</i> .

Subprocesos

<code>loop.subprocess_exec()</code>	Genera un subproceso.
<code>loop.subprocess_shell()</code>	Genera un subproceso desde un comando de shell.

Gestor de errores

<code>loop.call_exception_handler()</code>	Invoca al gestor de excepciones.
<code>loop.set_exception_handler()</code>	Establece un nuevo gestor de excepciones.
<code>loop.get_exception_handler()</code>	Obtiene el gestor de excepciones actual.
<code>loop.default_exception_handler()</code>	La implementación predetermina del gestor de excepciones.

Ejemplos

- Usando `asyncio.get_event_loop()` y `loop.run_forever()`.
- Usando `loop.call_later()`.
- Usando `loop.create_connection()` para implementar *un cliente de eco*.
- Usando `loop.create_connection()` para *conectar a un socket*.
- Usando `add_reader()` para mirar un FD y leer eventos.
- Usando `loop.add_signal_handler()`.
- Usando `loop.subprocess_exec()`.

Transportes

Todos los transportes implementan los siguientes métodos:

<code>transport.close()</code>	Cierra el transporte.
<code>transport.is_closing()</code>	Retorna True si el transporte está cerrado o se está cerrando.
<code>transport.get_extra_info()</code>	Solicita información sobre el transporte.
<code>transport.set_protocol()</code>	Establece un nuevo protocolo.
<code>transport.get_protocol()</code>	Retorna el protocolo actual.

Transportes que pueden recibir datos (conexiones TCP y Unix, *pipes*, etc). Retornan de métodos como `loop.create_connection()`, `loop.create_unix_connection()`, `loop.connect_read()`, etc:

Leer transportes

<code>transport.is_reading()</code>	Retorna <code>True</code> si el transporte está recibiendo.
<code>transport.pause_reading()</code>	Pausa la recepción.
<code>transport.resume_reading()</code>	Reanuda la recepción.

Transportes que pueden enviar datos (conexiones TCP y Unix, *pipes*, etc). Retornan de métodos como `loop.create_connection()`, `loop.create_unix_connection()`, `loop.connect_write_pipe()`, etc:

Escribir transportes

<code>transport.write()</code>	Escribe datos en el transporte.
<code>transport.writelines()</code>	Escribe búferes en el transporte.
<code>transport.can_write_eof()</code>	Retorna <code>True</code> si el transporte admite el envío de EOF.
<code>transport.write_eof()</code>	Cierra y envía EOF después de vaciar los datos almacenados en búfer.
<code>transport.abort()</code>	Cierra el transporte inmediatamente.
<code>transport.get_write_buffer_size()</code>	Retorna los límites superior e inferior para controlar el flujo de escritura.
<code>transport.set_write_buffer_limits()</code>	Establece nuevos límites superior e inferior para el control del flujo de escritura.

Transportes retornados por `loop.create_datagram_endpoint()`:

Transportes de datagramas

<code>transport.sendto()</code>	Envía datos al par remoto.
<code>transport.abort()</code>	Cierra el transporte inmediatamente.

Abstracción de transporte de bajo nivel sobre subprocessos. Retornado por `loop.subprocess_exec()` y `loop.subprocess_shell()`:

Transportes de subprocessos

<code>transport.get_pid()</code>	Retorna el id de proceso del subprocesso.
<code>transport.get_pipe_transport()</code>	Retorna el transporte para la <i>pipe</i> de comunicación solicitada (<i>stdin</i> , <i>stdout</i> o <i>stderr</i>).
<code>transport.get_returncode()</code>	Retorna el código de retorno del subprocesso.
<code>transport.kill()</code>	Mata el subprocesso.
<code>transport.send_signal()</code>	Envía una señal al subprocesso.
<code>transport.terminate()</code>	Detiene el subprocesso.
<code>transport.close()</code>	Mata el subprocesso y cierra todas las <i>pipes</i> .

Protocolos

Las clases de protocolo pueden implementar los siguientes **métodos de devolución de llamada**:

<code>callback <i>connection_made</i>()</code>	Se llama cuando se establece una conexión.
<code>callback <i>connection_lost</i>()</code>	Se llama cuando la conexión se pierde o cierra.
<code>callback <i>pause_writing</i>()</code>	Se llama cuando el búfer del transporte excede el límite superior.
<code>callback <i>resume_writing</i>()</code>	Se llama cuando el búfer del transporte se vacía por debajo del límite inferior.

Protocolos de streaming (TCP, Unix Sockets, Pipes)

<code>callback <i>data_received</i>()</code>	Se llama cuando se reciben algunos datos.
<code>callback <i>eof_received</i>()</code>	Se llama cuando se recibe un EOF.

Protocolos de streaming en búfer

<code>callback <i>get_buffer</i>()</code>	Se llama para asignar un nuevo búfer de recepción.
<code>callback <i>buffer_updated</i>()</code>	Se llama cuando el búfer se actualizó con los datos recibidos.
<code>callback <i>eof_received</i>()</code>	Se llama cuando se recibe un EOF.

Protocolos de datagramas

<code>callback <i>datagram_received</i>()</code>	Se llama cuando se recibe un datagrama.
<code>callback <i>error_received</i>()</code>	Se llama cuando una operación de envío o recepción anterior genera un <i>OSError</i> .

Protocolos de subprocessos

<code>callback <i>pipe_data_received</i>()</code>	Se llama cuando el proceso hijo escribe datos en su <i>pipe stdout</i> o <i>stderr</i> .
<code>callback <i>pipe_connection_lost</i>()</code>	Se llama cuando se cierra un <i>pipe</i> que se comunica con el proceso hijo.
<code>callback <i>process_exited</i>()</code>	Se llama cuando el proceso hijo ha finalizado.

Políticas de bucle de eventos

Las políticas son un mecanismo de bajo nivel para alterar el comportamiento de funciones como `asyncio.get_event_loop()`. Vea también la sección principal *políticas* para más detalles.

Acceso a políticas

<code>asyncio.get_event_loop_policy()</code>	Retorna la política actual en todo el proceso.
<code>asyncio.set_event_loop_policy()</code>	Establece una nueva política para todo el proceso.
<code>AbstractEventLoopPolicy</code>	Clase base para objetos de política.

18.1.14 Desarrollando con asyncio

La programación asincrónica es diferente a la programación «secuencial» clásica.

Esta página enumera errores y trampas comunes y explica cómo evitarlos.

Modo Depuración

Por defecto asyncio se ejecuta en modo producción. Para facilitar el desarrollo asyncio tiene un *modo depuración*.

Hay varias maneras de habilitar el modo depuración de asyncio:

- Definiendo la variable de entorno `PYTHONASYNCIODEBUG` a 1.
- Usando la opción `-X dev` de la línea de comandos de Python.
- Pasando `debug=True` a `asyncio.run()`.
- Invocando `loop.set_debug()`.

Además de habilitar el modo depuración, considere también:

- definir el nivel de log del *asyncio logger* a `logging.DEBUG`, por ejemplo el siguiente fragmento de código puede ser ejecutado al inicio de la aplicación:

```
logging.basicConfig(level=logging.DEBUG)
```

- configurando el módulo *warnings* para mostrar advertencias *ResourceWarning*. Una forma de hacerlo es usando la opción `-W default` de la línea de comandos.

Cuando el modo depuración está habilitado:

- asyncio comprueba las *corrutinas que no son esperadas* y las registra; esto mitiga la dificultad de las «esperas olvidadas».
- Muchas APIs asyncio que no son seguras para hilos (como los métodos `loop.call_soon()` y `loop.call_at()`) generan una excepción si son llamados desde un hilo equivocado.
- El tiempo de ejecución del selector E/S es registrado si tarda demasiado tiempo en realizar una operación E/S.
- Los callbacks que tardan más de 100ms son registrados. El atributo `loop.slow_callback_duration` puede ser usado para definir la duración mínima de ejecución en segundos considerada como «lenta».

Concurrencia y Multihilo

Un bucle de eventos se ejecuta en un hilo (generalmente el hilo principal) y ejecuta todos los callbacks y las Tareas en su hilo. Mientras una Tarea está ejecutándose en el bucle de eventos, ninguna otra Tarea puede ejecutarse en el mismo hilo. Cuando una Tarea ejecuta una expresión `await`, la Tarea en ejecución se suspende y el bucle de eventos ejecuta la siguiente Tarea.

To schedule a *callback* from another OS thread, the `loop.call_soon_threadsafe()` method should be used. Example:

```
loop.call_soon_threadsafe(callback, *args)
```

Casi ningún objeto `asyncio` es seguro entre hilos (*thread safe*), lo cual generalmente no es un problema a no ser que haya código que trabaje con ellos desde fuera de una Tarea o un callback. Si tal código necesita llamar a la API de `asyncio` de bajo nivel, se debe usar el método `loop.call_soon_threadsafe()`, por ejemplo:

```
loop.call_soon_threadsafe(fut.cancel)
```

Para programar un objeto de corrutina desde una hilo diferente del sistema operativo se debe usar la función `run_coroutine_threadsafe()`. Esta retorna un `concurrent.futures.Future` para acceder al resultado:

```
async def coro_func():
    return await asyncio.sleep(1, 42)

# Later in another OS thread:

future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
# Wait for the result:
result = future.result()
```

Para manejar señales y ejecutar subprocessos, el bucle de eventos debe ser ejecutado en el hilo principal.

El método `loop.run_in_executor()` puede ser usado con un `concurrent.futures.ThreadPoolExecutor` para ejecutar código bloqueante en un hilo diferente del sistema operativo sin bloquear el hilo del sistema operativo en el que el bucle de eventos está siendo ejecutado.

There is currently no way to schedule coroutines or callbacks directly from a different process (such as one started with *multiprocessing*). The *Event Loop Methods* section lists APIs that can read from pipes and watch file descriptors without blocking the event loop. In addition, `asyncio`'s *Subprocess* APIs provide a way to start a process and communicate with it from the event loop. Lastly, the aforementioned `loop.run_in_executor()` method can also be used with a `concurrent.futures.ProcessPoolExecutor` to execute code in a different process.

Ejecutando Código Bloqueante

Código bloqueante (dependiente de la CPU) no debe ser ejecutado directamente. Por ejemplo, si una función realiza un cálculo intensivo de CPU durante 1 segundo, todas las Tareas y operaciones de Entrada/Salida (IO) concurrentes se retrasarían 1 segundo.

Un ejecutor puede ser usado para ejecutar una tarea en un hilo diferente o incluso en un proceso diferente para evitar bloquear el hilo del sistema operativo con el bucle de eventos. Consulte el método `loop.run_in_executor()` para más detalles.

Logueando

`asyncio` usa el módulo `logging` y todo el logueo es realizado mediante el logger "`asyncio`".

El nivel de log por defecto es `logging.INFO`, el cual puede ser fácilmente ajustado:

```
logging.getLogger("asyncio").setLevel(logging.WARNING)
```

Detectar corrutinas no esperadas

Cuando una función de corrutina es invocada, pero no esperada (por ejemplo `coro()` en lugar de `await coro()`) o la corrutina no es programada con `asyncio.create_task()`, `asyncio` emitirá una `RuntimeWarning`:

```
import asyncio

async def test():
    print("never scheduled")

async def main():
    test()

asyncio.run(main())
```

Salida:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
    test()
```

Salida en modo depuración:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
Coroutine created at (most recent call last)
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

< .. >

File "../t.py", line 7, in main
    test()
    test()
```

La solución habitual es esperar la corrutina o llamar a la función `asyncio.create_task()`:

```
async def main():
    await test()
```

Detectar excepciones nunca recuperadas

Si un `Future.set_exception()` es invocado pero el objeto Futuro nunca es esperado, la excepción nunca será propagada al código del usuario. En este caso, `asyncio` emitiría un mensaje de registro cuando el objeto Futuro fuera recolectado como basura.

Ejemplo de una excepción no manejada:

```
import asyncio

async def bug():
    raise Exception("not consumed")

async def main():
    asyncio.create_task(bug())

asyncio.run(main())
```

Salida:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
    exception=Exception('not consumed')>
```

(continué en la próxima página)

(proviene de la página anterior)

```
Traceback (most recent call last):
  File "test.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

Habilita el modo depuración para obtener el seguimiento de pila (*traceback*) donde la tarea fue creada:

```
asyncio.run(main(), debug=True)
```

Salida en modo depuración:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
      exception=Exception('not consumed') created at asyncio/tasks.py:321>

source_traceback: Object created at (most recent call last):
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

< .. >

Traceback (most recent call last):
  File "../t.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

Nota: El código fuente para `asyncio` puede encontrarse en [Lib/asyncio/](#).

18.2 socket — Low-level networking interface

Source code: [Lib/socket.py](#)

This module provides access to the BSD *socket* interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

Nota: Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

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.

Ver también:

Module `socketserver` Classes that simplify writing network servers.

Module `ssl` A TLS/SSL wrapper for socket objects.

18.2.1 Socket families

Depending on the system and the build options, various socket families are supported by this module.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows:

- The address of an `AF_UNIX` socket bound to a file system node is represented as a string, using the file system encoding and the 'surrogateescape' error handler (see [PEP 383](#)). An address in Linux's abstract namespace is returned as a *bytes-like object* with an initial null byte; note that sockets in this namespace can communicate with normal file system sockets, so programs intended to run on Linux may need to deal with both types of address. A string or bytes-like object can be used for either type of address when passing it as an argument.

Distinto en la versión 3.3: Previously, `AF_UNIX` socket paths were assumed to use UTF-8 encoding.

Distinto en la versión 3.5: Writable *bytes-like object* is now accepted.

- 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 IPv4 addresses, two special forms are accepted instead of a host address: '' represents `INADDR_ANY`, which is used to bind to all interfaces, and the string '<broadcast>' represents `INADDR_BROADCAST`. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.
- For `AF_INET6` address family, a four-tuple (`host`, `port`, `flowinfo`, `scopeid`) is used, where `flowinfo` and `scopeid` represent the `sin6_flowinfo` and `sin6_scope_id` members 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.

Distinto en la versión 3.7: For multicast addresses (with `scopeid` meaningful) `address` may not contain `%scope` (or `zone id`) part. This information is superfluous and may be safely omitted (recommended).

- `AF_NETLINK` sockets are represented as pairs (`pid`, `groups`).
- Linux-only support for TIPC is 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.

- A tuple (`interface`,) is used for the `AF_CAN` address family, where `interface` is a string representing a network interface name like 'can0'. The network interface name '' can be used to receive packets from all network interfaces of this family.
 - `CAN_ISOTP` protocol require a tuple (`interface`, `rx_addr`, `tx_addr`) where both additional parameters are unsigned long integer that represent a CAN identifier (standard or extended).
- A string or a tuple (`id`, `unit`) is used for the `SYSPROTO_CONTROL` protocol of the `PF_SYSTEM` family. The string is the name of a kernel control using a dynamically-assigned ID. The tuple can be used if ID and unit number of the kernel control are known or if a registered ID is used.

Nuevo en la versión 3.3.

- `AF_BLUETOOTH` supports the following protocols and address formats:
 - `BTPROTO_L2CAP` accepts `(bdaddr, psm)` where `bdaddr` is the Bluetooth address as a string and `psm` is an integer.
 - `BTPROTO_RFCOMM` accepts `(bdaddr, channel)` where `bdaddr` is the Bluetooth address as a string and `channel` is an integer.
 - `BTPROTO_HCI` accepts `(device_id,)` where `device_id` is either an integer or a string with the Bluetooth address of the interface. (This depends on your OS; NetBSD and DragonFlyBSD expect a Bluetooth address while everything else expects an integer.)

Distinto en la versión 3.2: NetBSD and DragonFlyBSD support added.

 - `BTPROTO_SCO` accepts `bdaddr` where `bdaddr` is a *bytes* object containing the Bluetooth address in a string format. (ex. `b'12:23:34:45:56:67'`) This protocol is not supported under FreeBSD.

- `AF_ALG` is a Linux-only socket based interface to Kernel cryptography. An algorithm socket is configured with a tuple of two to four elements `(type, name [, feat [, mask]])`, where:
 - *type* is the algorithm type as string, e.g. `aead`, `hash`, `skcipher` or `rng`.
 - *name* is the algorithm name and operation mode as string, e.g. `sha256`, `hmac (sha256)`, `cbc (aes)` or `drbg_nopr_ctr_aes256`.
 - *feat* and *mask* are unsigned 32bit integers.

Availability: Linux 2.6.38, some algorithm types require more recent Kernels.

Nuevo en la versión 3.6.

- `AF_VSOCK` allows communication between virtual machines and their hosts. The sockets are represented as a `(CID, port)` tuple where the context ID or CID and port are integers.

Availability: Linux >= 4.8 QEMU >= 2.8 ESX >= 4.0 ESX Workstation >= 6.5.

Nuevo en la versión 3.7.

- `AF_PACKET` is a low-level interface directly to network devices. The packets are represented by the tuple `(ifname, proto[, pkttype[, hatype[, addr]])` where:
 - *ifname* - String specifying the device name.
 - *proto* - An in network-byte-order integer specifying the Ethernet protocol number.
 - *pkttype* - Optional integer specifying the packet type:
 - * `PACKET_HOST` (the default) - Packet addressed to the local host.
 - * `PACKET_BROADCAST` - Physical-layer broadcast packet.
 - * `PACKET_MULTICAST` - Packet sent to a physical-layer multicast address.
 - * `PACKET_OTHERHOST` - Packet to some other host that has been caught by a device driver in promiscuous mode.
 - * `PACKET_OUTGOING` - Packet originating from the local host that is looped back to a packet socket.
 - *hatype* - Optional integer specifying the ARP hardware address type.
 - *addr* - Optional bytes-like object specifying the hardware physical address, whose interpretation depends on the device.
- `AF_QIPCRTR` is a Linux-only socket based interface for communicating with services running on co-processors in Qualcomm platforms. The address family is represented as a `(node, port)` tuple where the *node* and *port* are non-negative integers.

Nuevo en la versión 3.8.

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.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; starting from Python 3.3, errors related to socket or address semantics raise *OSError* or one of its subclasses (they used to raise *socket.error*).

Non-blocking mode is supported through *setblocking()*. A generalization of this based on timeouts is supported through *settimeout()*.

18.2.2 Module contents

The module *socket* exports the following elements.

Exceptions

exception *socket.error*

A deprecated alias of *OSError*.

Distinto en la versión 3.3: Following [PEP 3151](#), this class was made an alias of *OSError*.

exception *socket.herror*

A subclass of *OSError*, this exception is raised for address-related errors, i.e. for functions that use *h_errno* in the POSIX C API, including *gethostbyname_ex()* and *gethostbyaddr()*. The accompanying value is a pair (*h_errno*, *string*) representing an error returned by a library call. *h_errno* is a numeric value, while *string* represents the description of *h_errno*, as returned by the *hstrerror()* C function.

Distinto en la versión 3.3: This class was made a subclass of *OSError*.

exception *socket.gaierror*

A subclass of *OSError*, this exception is raised for address-related errors by *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 numeric *error* value will match one of the *EAI_** constants defined in this module.

Distinto en la versión 3.3: This class was made a subclass of *OSError*.

exception *socket.timeout*

A subclass of *OSError*, this exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to *settimeout()* (or implicitly through *setdefaulttimeout()*). The accompanying value is a string whose value is currently always «timed out».

Distinto en la versión 3.3: This class was made a subclass of *OSError*.

Constants

The *AF_** and *SOCK_** constants are now *AddressFamily* and *SocketKind* *IntEnum* collections.

Nuevo en la versión 3.4.

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. More constants may be available depending on the system.

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()`. More constants may be available depending on the system. (Only `SOCK_STREAM` and `SOCK_DGRAM` appear to be generally useful.)

socket.SOCK_CLOEXEC**socket.SOCK_NONBLOCK**

These two constants, if defined, can be combined with the socket types and allow you to set some flags atomically (thus avoiding possible race conditions and the need for separate calls).

Ver también:

[Secure File Descriptor Handling](#) for a more thorough explanation.

Availability: Linux >= 2.6.27.

Nuevo en la versión 3.2.

SO_***socket.SOMAXCONN****MSG_*****SOL_*****SCM_*****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.

Distinto en la versión 3.6: `SO_DOMAIN`, `SO_PROTOCOL`, `SO_PEERSEC`, `SO_PASSSEC`, `TCP_USER_TIMEOUT`, `TCP_CONGESTION` were added.

Distinto en la versión 3.6.5: On Windows, `TCP_FASTOPEN`, `TCP_KEEPCNT` appear if run-time Windows supports.

Distinto en la versión 3.7: `TCP_NOTSENT_LOWAT` was added.

On Windows, `TCP_KEEPIRL`, `TCP_KEEPIRLVL` appear if run-time Windows supports.

socket.AF_CAN**socket.PF_CAN****SOL_CAN_*****CAN_***

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.25.

Nuevo en la versión 3.3.

socket.CAN_BCM**CAN_BCM_***

`CAN_BCM`, in the CAN protocol family, is the broadcast manager (BCM) protocol. Broadcast manager constants, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.25.

Nota: The `CAN_BCM_CAN_FD_FRAME` flag is only available on Linux ≥ 4.8 .

Nuevo en la versión 3.4.

`socket.CAN_RAW_FD_FRAMES`

Enables CAN FD support in a `CAN_RAW` socket. This is disabled by default. This allows your application to send both CAN and CAN FD frames; however, you must accept both CAN and CAN FD frames when reading from the socket.

This constant is documented in the Linux documentation.

Availability: Linux ≥ 3.6 .

Nuevo en la versión 3.5.

`socket.CAN_ISOTP`

`CAN_ISOTP`, in the CAN protocol family, is the ISO-TP (ISO 15765-2) protocol. ISO-TP constants, documented in the Linux documentation.

Availability: Linux $\geq 2.6.25$.

Nuevo en la versión 3.7.

`socket.AF_PACKET`

`socket.PF_PACKET`

PACKET_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux ≥ 2.2 .

`socket.AF_RDS`

`socket.PF_RDS`

`socket.SOL_RDS`

RDS_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux $\geq 2.6.30$.

Nuevo en la versión 3.3.

`socket.SIO_RCVALL`

`socket.SIO_KEEPA_LIVE_VALS`

`socket.SIO_LOOPBACK_FAST_PATH`

RCVALL_*

Constants for Windows' `WSAIocctl()`. The constants are used as arguments to the `ioctl()` method of socket objects.

Distinto en la versión 3.6: `SIO_LOOPBACK_FAST_PATH` was added.

TIPC_*

TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for more information.

`socket.AF_ALG`

`socket.SOL_ALG`

ALG_*

Constants for Linux Kernel cryptography.

Availability: Linux $\geq 2.6.38$.

Nuevo en la versión 3.6.

`socket.AF_VSOCK`

`socket.IOCTL_VM_SOCKETS_GET_LOCAL_CID`

VMADDR*

SO_VM*

Constants for Linux host/guest communication.

Availability: Linux >= 4.8.

Nuevo en la versión 3.7.

socket.AF_LINK

Availability: BSD, OSX.

Nuevo en la versión 3.4.

socket.has_ipv6

This constant contains a boolean value which indicates if IPv6 is supported on this platform.

socket.BDADDR_ANY**socket.BDADDR_LOCAL**

These are string constants containing Bluetooth addresses with special meanings. For example, *BDADDR_ANY* can be used to indicate any address when specifying the binding socket with *BTPROTO_RFCOMM*.

socket.HCI_FILTER**socket.HCI_TIME_STAMP****socket.HCI_DATA_DIR**

For use with *BTPROTO_HCI*. *HCI_FILTER* is not available for NetBSD or DragonFlyBSD. *HCI_TIME_STAMP* and *HCI_DATA_DIR* are not available for FreeBSD, NetBSD, or DragonFlyBSD.

socket.AF_QIPCRTR

Constant for Qualcomm's IPC router protocol, used to communicate with service providing remote processors.

Availability: Linux >= 4.7.

Functions

Creating sockets

The following functions all create *socket objects*.

socket.socket (*family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None*)

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*, *AF_UNIX*, *AF_CAN*, *AF_PACKET*, or *AF_RDS*. The socket type should be *SOCK_STREAM* (the default), *SOCK_DGRAM*, *SOCK_RAW* or perhaps one of the other *SOCK_** constants. The protocol number is usually zero and may be omitted or in the case where the address family is *AF_CAN* the protocol should be one of *CAN_RAW*, *CAN_BCM* or *CAN_ISOTP*.

If *fileno* is specified, the values for *family*, *type*, and *proto* are auto-detected from the specified file descriptor. Auto-detection can be overruled by calling the function with explicit *family*, *type*, or *proto* arguments. This only affects how Python represents e.g. the return value of *socket.getpeername()* but not the actual OS resource. Unlike *socket.fromfd()*, *fileno* will return the same socket and not a duplicate. This may help close a detached socket using *socket.close()*.

The newly created socket is *non-inheritable*.

Raises an *auditing event* *socket.__new__* with arguments *self*, *family*, *type*, *protocol*.

Distinto en la versión 3.3: The *AF_CAN* family was added. The *AF_RDS* family was added.

Distinto en la versión 3.4: The *CAN_BCM* protocol was added.

Distinto en la versión 3.4: The returned socket is now non-inheritable.

Distinto en la versión 3.7: The *CAN_ISOTP* protocol was added.

Distinto en la versión 3.7: When *SOCK_NONBLOCK* or *SOCK_CLOEXEC* bit flags are applied to *type* they are cleared, and *socket.type* will not reflect them. They are still passed to the underlying system *socket()* call. Therefore,

```
sock = socket.socket (
    socket.AF_INET,
    socket.SOCK_STREAM | socket.SOCK_NONBLOCK)
```

will still create a non-blocking socket on OSes that support `SOCK_NONBLOCK`, but `sock.type` will be set to `socket.SOCK_STREAM`.

`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`.

The newly created sockets are *non-inheritable*.

Distinto en la versión 3.2: The returned socket objects now support the whole socket API, rather than a subset.

Distinto en la versión 3.4: The returned sockets are now non-inheritable.

Distinto en la versión 3.5: Windows support added.

`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.

Distinto en la versión 3.2: *source_address* was added.

`socket.create_server(address, *, family=AF_INET, backlog=None, reuse_port=False, dualstack_ipv6=False)`

Convenience function which creates a TCP socket bound to *address* (a 2-tuple (host, port)) and return the socket object.

family should be either `AF_INET` or `AF_INET6`. *backlog* is the queue size passed to `socket.listen()`; when 0 a default reasonable value is chosen. *reuse_port* dictates whether to set the `SO_REUSEPORT` socket option.

If *dualstack_ipv6* is true and the platform supports it the socket will be able to accept both IPv4 and IPv6 connections, else it will raise `ValueError`. Most POSIX platforms and Windows are supposed to support this functionality. When this functionality is enabled the address returned by `socket.getpeername()` when an IPv4 connection occurs will be an IPv6 address represented as an IPv4-mapped IPv6 address. If *dualstack_ipv6* is false it will explicitly disable this functionality on platforms that enable it by default (e.g. Linux). This parameter can be used in conjunction with `has_dualstack_ipv6()`:

```
import socket

addr = ("", 8080) # all interfaces, port 8080
if socket.has_dualstack_ipv6():
    s = socket.create_server(addr, family=socket.AF_INET6, dualstack_ipv6=True)
else:
    s = socket.create_server(addr)
```

Nota: On POSIX platforms the `SO_REUSEADDR` socket option is set in order to immediately reuse previous sockets which were bound on the same *address* and remained in `TIME_WAIT` state.

Nuevo en la versión 3.8.

`socket.has_dualstack_ipv6()`

Return `True` if the platform supports creating a TCP socket which can handle both IPv4 and IPv6 connections.

Nuevo en la versión 3.8.

`socket.fromfd(fd, family, type, proto=0)`

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.

The newly created socket is *non-inheritable*.

Distinto en la versión 3.4: The returned socket is now non-inheritable.

`socket.fromshare(data)`

Instantiate a socket from data obtained from the `socket.share()` method. The socket is assumed to be in blocking mode.

Availability: Windows.

Nuevo en la versión 3.3.

`socket.SocketType`

This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

Other functions

The `socket` module also offers various network-related services:

`socket.close(fd)`

Close a socket file descriptor. This is like `os.close()`, but for sockets. On some platforms (most noticeable Windows) `os.close()` does not work for socket file descriptors.

Nuevo en la versión 3.7.

`socket.getaddrinfo(host, port, family=0, type=0, proto=0, flags=0)`

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*, *type* and *proto* arguments can be optionally specified in order to narrow the list of addresses returned. Passing zero as a value for each of these arguments selects the full range of results. The *flags* argument can be one or several of the `AI_*` constants, and will influence how results are computed and returned. 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, type, proto, canonname, sockaddr)
```

In these tuples, *family*, *type*, *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.

Raises an *auditing event* `socket.getaddrinfo` with arguments *host*, *port*, *family*, *type*, *protocol*.

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, proto=socket.IPPROTO_TCP)
[(<AddressFamily.AF_INET6: 10>, <SocketType.SOCK_STREAM: 1>,
  6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (<AddressFamily.AF_INET: 2>, <SocketType.SOCK_STREAM: 1>,
  6, '', ('93.184.216.34', 80))]
```

Distinto en la versión 3.2: parameters can now be passed using keyword arguments.

Distinto en la versión 3.7: for IPv6 multicast addresses, string representing an address will not contain %scope part.

`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.

`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.

Raises an *auditing event* `socket.gethostbyname` with argument *hostname*.

`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.

Raises an *auditing event* `socket.gethostbyname` with argument *hostname*.

`socket.gethostname()`

Return a string containing the hostname of the machine where the Python interpreter is currently executing.

Raises an *auditing event* `socket.gethostname` with no arguments.

Note: `gethostname()` doesn't always return the fully qualified domain name; use `getfqdn()` for that.

`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.

Raises an *auditing event* `socket.gethostbyaddr` with argument *ip_address*.

`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.

For IPv6 addresses, %scope is appended to the host part if *sockaddr* contains meaningful *scopeid*. Usually this happens for multicast addresses.

For more information about *flags* you can consult `getnameinfo(3)`.

Raises an *auditing event* `socket.getnameinfo` with argument *sockaddr*.

`socket.getprotobyne(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.

Raises an *auditing event* `socket.getservbyname` with arguments `servicename, protocolname`.

`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.

Raises an *auditing event* `socket.getservbyport` with arguments `port, protocolname`.

`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.

Obsoleto desde la versión 3.7: In case *x* does not fit in 16-bit unsigned integer, but does fit in a positive C int, it is silently truncated to 16-bit unsigned integer. This silent truncation feature is deprecated, and will raise an exception in future versions of Python.

`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.

Obsoleto desde la versión 3.7: In case *x* does not fit in 16-bit unsigned integer, but does fit in a positive C int, it is silently truncated to 16-bit unsigned integer. This silent truncation feature is deprecated, and will raise an exception in future versions of Python.

`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 bytes object 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, *OSError* 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 *bytes-like object* four bytes 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 byte sequence passed to this function is not exactly 4 bytes in length, *OSError* will be raised. *inet_ntoa()* does not support IPv6, and *inet_ntop()* should be used instead for IPv4/v6 dual stack support.

Distinto en la versión 3.5: Writable *bytes-like object* is now accepted.

`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, `OSError` 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), Windows.

Distinto en la versión 3.4: Windows support added

`socket.inet_ntop(address_family, packed_ip)`

Convert a packed IP address (a *bytes-like object* of some number of bytes) 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 bytes object `packed_ip` is not the correct length for the specified address family, `ValueError` will be raised. `OSError` is raised for errors from the call to `inet_ntop()`.

Availability: Unix (maybe not all platforms), Windows.

Distinto en la versión 3.4: Windows support added

Distinto en la versión 3.5: Writable *bytes-like object* is now accepted.

`socket.CMSG_LEN(length)`

Return the total length, without trailing padding, of an ancillary data item with associated data of the given `length`. This value can often be used as the buffer size for `recvmsg()` to receive a single item of ancillary data, but **RFC 3542** requires portable applications to use `CMSG_SPACE()` and thus include space for padding, even when the item will be the last in the buffer. Raises `OverflowError` if `length` is outside the permissible range of values.

Availability: most Unix platforms, possibly others.

Nuevo en la versión 3.3.

`socket.CMSG_SPACE(length)`

Return the buffer size needed for `recvmsg()` to receive an ancillary data item with associated data of the given `length`, along with any trailing padding. The buffer space needed to receive multiple items is the sum of the `CMSG_SPACE()` values for their associated data lengths. Raises `OverflowError` if `length` is outside the permissible range of values.

Note that some systems might support ancillary data without providing this function. Also note that setting the buffer size using the results of this function may not precisely limit the amount of ancillary data that can be received, since additional data may be able to fit into the padding area.

Availability: most Unix platforms, possibly others.

Nuevo en la versión 3.3.

`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`.

`socket.setdefaulttimeout(timeout)`

Set the default timeout in seconds (float) for new socket objects. When the socket module is first imported, the default is `None`. See `settimeout()` for possible values and their respective meanings.

`socket.sethostname(name)`

Set the machine's hostname to `name`. This will raise an `OSError` if you don't have enough rights.

Raises an *auditing event* `socket.sethostname` with argument `name`.

Availability: Unix.

Nuevo en la versión 3.3.

`socket.if_nameindex()`

Return a list of network interface information (index int, name string) tuples. *OSError* if the system call fails.

Availability: Unix, Windows.

Nuevo en la versión 3.3.

Distinto en la versión 3.8: Windows support was added.

Nota: On Windows network interfaces have different names in different contexts (all names are examples):

- UUID: {FB605B73-AAC2-49A6-9A2F-25416AEA0573}
- name: ethernet_32770
- friendly name: vEthernet (nat)
- description: Hyper-V Virtual Ethernet Adapter

This function returns names of the second form from the list, ethernet_32770 in this example case.

`socket.if_nameindex(if_name)`

Return a network interface index number corresponding to an interface name. *OSError* if no interface with the given name exists.

Availability: Unix, Windows.

Nuevo en la versión 3.3.

Distinto en la versión 3.8: Windows support was added.

Ver también:

«Interface name» is a name as documented in `if_nameindex()`.

`socket.if_indexoname(if_index)`

Return a network interface name corresponding to an interface index number. *OSError* if no interface with the given index exists.

Availability: Unix, Windows.

Nuevo en la versión 3.3.

Distinto en la versión 3.8: Windows support was added.

Ver también:

«Interface name» is a name as documented in `if_nameindex()`.

18.2.3 Socket Objects

Socket objects have the following methods. Except for `makefile()`, these correspond to Unix system calls applicable to sockets.

Distinto en la versión 3.2: Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling `close()`.

`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.

The newly created socket is *non-inheritable*.

Distinto en la versión 3.4: The socket is now non-inheritable.

Distinto en la versión 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see [PEP 475](#) for the rationale).

`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.)

Raises an *auditing event* `socket.bind` with arguments `self`, `address`.

`socket.close()`

Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from `makefile()` are closed. Once that happens, 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, but it is recommended to `close()` them explicitly, or to use a `with` statement around them.

Distinto en la versión 3.6: `OSError` is now raised if an error occurs when the underlying `close()` call is made.

Nota: `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.)

If the connection is interrupted by a signal, the method waits until the connection completes, or raise a `socket.timeout` on timeout, if the signal handler doesn't raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an `InterruptedError` exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

Raises an *auditing event* `socket.connect` with arguments `self`, `address`.

Distinto en la versión 3.5: The method now waits until the connection completes instead of raising an `InterruptedError` exception if the connection is interrupted by a signal, the signal handler doesn't raise an exception and the socket is blocking or has a timeout (see the [PEP 475](#) for the rationale).

`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.

Raises an *auditing event* `socket.connect` with arguments `self`, `address`.

`socket.detach()`

Put the socket object into closed state without actually closing the underlying file descriptor. The file descriptor is returned, and can be reused for other purposes.

Nuevo en la versión 3.2.

`socket.dup()`

Duplicate the socket.

The newly created socket is *non-inheritable*.

Distinto en la versión 3.4: The socket is now non-inheritable.

`socket.fileno()`

Return the socket's file descriptor (a small integer), or -1 on failure. 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.get_inheritable()`

Get the *inheritable flag* of the socket's file descriptor or socket's handle: `True` if the socket can be inherited in child processes, `False` if it cannot.

Nuevo en la versión 3.4.

`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 bytes object. 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 byte strings).

`socket.getblocking()`

Return `True` if socket is in blocking mode, `False` if in non-blocking.

This is equivalent to checking `socket.gettimeout() == 0`.

Nuevo en la versión 3.7.

`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()*.

`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.

Currently only the following control codes are supported: `SIO_RCVALL`, `SIO_KEEPA_LIVE_VALS`, and `SIO_LOOPBACK_FAST_PATH`.

Distinto en la versión 3.6: `SIO_LOOPBACK_FAST_PATH` was added.

`socket.listen([backlog])`

Enable a server to accept connections. If *backlog* is specified, it must be at least 0 (if it is lower, it is set to 0); it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

Distinto en la versión 3.5: The *backlog* parameter is now optional.

`socket.makefile(mode='r', buffering=None, *, encoding=None, errors=None, newline=None)`

Return a *file object* associated with the socket. The exact returned type depends on the arguments given to *makefile()*. These arguments are interpreted the same way as by the built-in *open()* function, except the only supported *mode* values are `'r'` (default), `'w'` and `'b'`.

The socket must be in blocking mode; it can have a timeout, but the file object's internal buffer may end up in an inconsistent state if a timeout occurs.

Closing the file object returned by *makefile()* won't close the original socket unless all other file objects have been closed and *socket.close()* has been called on the socket object.

Nota: 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 bytes object 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.

Nota: For best match with hardware and network realities, the value of *bufsize* should be a relatively small power of 2, for example, 4096.

Distinto en la versión 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see [PEP 475](#) for the rationale).

`socket.recvfrom(bufsize[, flags])`

Receive data from the socket. The return value is a pair (*bytes*, *address*) where *bytes* is a bytes object 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.)

Distinto en la versión 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see [PEP 475](#) for the rationale).

Distinto en la versión 3.7: For multicast IPv6 address, first item of *address* does not contain %scope part anymore. In order to get full IPv6 address use `getnameinfo()`.

`socket.recvmsg(bufsize[, ancbufsize[, flags]])`

Receive normal data (up to *bufsize* bytes) and ancillary data from the socket. The *ancbufsize* argument sets the size in bytes of the internal buffer used to receive the ancillary data; it defaults to 0, meaning that no ancillary data will be received. Appropriate buffer sizes for ancillary data can be calculated using `CMSG_SPACE()` or `CMSG_LEN()`, and items which do not fit into the buffer might be truncated or discarded. The *flags* argument defaults to 0 and has the same meaning as for `recv()`.

The return value is a 4-tuple: (*data*, *ancdata*, *msg_flags*, *address*). The *data* item is a *bytes* object holding the non-ancillary data received. The *ancdata* item is a list of zero or more tuples (*cmsg_level*, *cmsg_type*, *cmsg_data*) representing the ancillary data (control messages) received: *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a *bytes* object holding the associated data. The *msg_flags* item is the bitwise OR of various flags indicating conditions on the received message; see your system documentation for details. If the receiving socket is unconnected, *address* is the address of the sending socket, if available; otherwise, its value is unspecified.

On some systems, `sendmsg()` and `recvmsg()` can be used to pass file descriptors between processes over an `AF_UNIX` socket. When this facility is used (it is often restricted to `SOCK_STREAM` sockets), `recvmsg()` will return, in its ancillary data, items of the form (`socket.SOL_SOCKET`, `socket.SCM_RIGHTS`, *fds*), where *fds* is a *bytes* object representing the new file descriptors as a binary array of the native C `int` type. If `recvmsg()` raises an exception after the system call returns, it will first attempt to close any file descriptors received via this mechanism.

Some systems do not indicate the truncated length of ancillary data items which have been only partially received. If an item appears to extend beyond the end of the buffer, `recvmsg()` will issue a `RuntimeWarning`, and will return the part of it which is inside the buffer provided it has not been truncated before the start of its associated data.

On systems which support the `SCM_RIGHTS` mechanism, the following function will receive up to *maxfds* file descriptors, returning the message data and a list containing the descriptors (while ignoring unexpected conditions such as unrelated control messages being received). See also `sendmsg()`.

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")    # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds *
    ↪ fds.itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS:
            # Append data, ignoring any truncated integers at the end.
            fds.frombytes(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.
    ↪ itemsize)])
    return msg, list(fds)
```

Availability: most Unix platforms, possibly others.

Nuevo en la versión 3.3.

Distinto en la versión 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see [PEP 475](#) for the rationale).

`socket.recvmsg_into(buffers[, ancbufsize[, flags]])`

Receive normal data and ancillary data from the socket, behaving as *recvmsg()* would, but scatter the non-ancillary data into a series of buffers instead of returning a new bytes object. The *buffers* argument must be an iterable of objects that export writable buffers (e.g. *bytearray* objects); these will be filled with successive chunks of the non-ancillary data until it has all been written or there are no more buffers. The operating system may set a limit (*sysconf()* value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancbufsize* and *flags* arguments have the same meaning as for *recvmsg()*.

The return value is a 4-tuple: (*nbytes*, *ancdata*, *msg_flags*, *address*), where *nbytes* is the total number of bytes of non-ancillary data written into the buffers, and *ancdata*, *msg_flags* and *address* are the same as for *recvmsg()*.

Example:

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'-----')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
```

Availability: most Unix platforms, possibly others.

Nuevo en la versión 3.3.

`socket.recvfrom_into(buffer[, nbytes[, flags]])`

Receive data from the socket, writing it into *buffer* instead of creating a new bytestring. 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.)

`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 bytestring. 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.

`socket.send(bytes[, 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 topic, consult the socket-howto.

Distinto en la versión 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see [PEP 475](#) for the rationale).

`socket.sendall(bytes[, 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 *bytes* 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.

Distinto en la versión 3.5: The socket timeout is no more reset each time data is sent successfully. The socket timeout is now the maximum total duration to send all data.

Distinto en la versión 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see [PEP 475](#) for the rationale).

`socket.sendto(bytes, address)`

`socket.sendto(bytes, 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.)

Raises an `auditing event` `socket.sendto` with arguments `self, address`.

Distinto en la versión 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see [PEP 475](#) for the rationale).

`socket.sendmsg(bufers[, ancdata[, flags[, address]]])`

Send normal and ancillary data to the socket, gathering the non-ancillary data from a series of buffers and concatenating it into a single message. The *bufers* argument specifies the non-ancillary data as an iterable of *bytes-like objects* (e.g. `bytes` objects); the operating system may set a limit (`sysconf()` value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancdata* argument specifies the ancillary data (control messages) as an iterable of zero or more tuples (`cmsg_level`, `cmsg_type`, `cmsg_data`), where `cmsg_level` and `cmsg_type` are integers specifying the protocol level and protocol-specific type respectively, and `cmsg_data` is a bytes-like object holding the associated data. Note that some systems (in particular, systems without `MSG_SPACE()`) might support sending only one control message per call. The *flags* argument defaults to 0 and has the same meaning as for `send()`. If *address* is supplied and not `None`, it sets a destination address for the message. The return value is the number of bytes of non-ancillary data sent.

The following function sends the list of file descriptors *fds* over an `AF_UNIX` socket, on systems which support the `SCM_RIGHTS` mechanism. See also `recvmsg()`.

```
import socket, array

def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SCM_RIGHTS, array.
    ↪array("i", fds))])
```

Availability: most Unix platforms, possibly others.

Raises an `auditing event` `socket.sendmsg` with arguments `self, address`.

Nuevo en la versión 3.3.

Distinto en la versión 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see [PEP 475](#) for the rationale).

`socket.sendmsg_afalg([msg], *, op[, iv[, assoclen[, flags]]])`

Specialized version of `sendmsg()` for `AF_ALG` socket. Set mode, IV, AEAD associated data length and flags for `AF_ALG` socket.

Availability: Linux >= 2.6.38.

Nuevo en la versión 3.6.

`socket.sendfile(file, offset=0, count=None)`

Send a file until EOF is reached by using high-performance `os.sendfile` and return the total number of bytes which were sent. `file` must be a regular file object opened in binary mode. If `os.sendfile` is not available (e.g. Windows) or `file` is not a regular file `send()` will be used instead. `offset` tells from where to start reading the file. If specified, `count` is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is updated on return or also in case of error in which case `file.tell()` can be used to figure out the number of bytes which were sent. The socket must be of `SOCK_STREAM` type. Non-blocking sockets are not supported.

Nuevo en la versión 3.5.

`socket.set_inheritable(inheritable)`

Set the *inheritable flag* of the socket's file descriptor or socket's handle.

Nuevo en la versión 3.4.

`socket.setblocking(flag)`

Set blocking or non-blocking mode of the socket: if `flag` is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain `settimeout()` calls:

- `sock.setblocking(True)` is equivalent to `sock.settimeout(None)`
- `sock.setblocking(False)` is equivalent to `sock.settimeout(0.0)`

Distinto en la versión 3.7: The method no longer applies `SOCK_NONBLOCK` flag on `socket.type`.

`socket.settimeout(value)`

Set a timeout on blocking socket operations. The `value` argument can be a nonnegative floating point number expressing seconds, or `None`. If a non-zero value is given, subsequent socket operations will raise a *timeout* exception if the timeout period `value` has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If `None` is given, the socket is put in blocking mode.

For further information, please consult the *notes on socket timeouts*.

Distinto en la versión 3.7: The method no longer toggles `SOCK_NONBLOCK` flag on `socket.type`.

`socket.setsockopt(level, optname, value: int)`

`socket.setsockopt(level, optname, value: buffer)`

`socket.setsockopt(level, optname, None, optlen: int)`

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, `None` or a *bytes-like object* representing a buffer. In the later case it is up to the caller to ensure that the bytearray contains the proper bits (see the optional built-in module `struct` for a way to encode C structures as bytestrings). When `value` is set to `None`, `optlen` argument is required. It's equivalent to call `setsockopt()` C function with `optval=NULL` and `optlen=optlen`.

Distinto en la versión 3.5: Writable *bytes-like object* is now accepted.

Distinto en la versión 3.6: `setsockopt(level, optname, None, optlen: int)` form added.

`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.

`socket.share(process_id)`

Duplicate a socket and prepare it for sharing with a target process. The target process must be provided with `process_id`. The resulting bytes object can then be passed to the target process using some form of interprocess

communication and the socket can be recreated there using `fromshare()`. Once this method has been called, it is safe to close the socket since the operating system has already duplicated it for the target process.

Availability: Windows.

Nuevo en la versión 3.3.

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.

`socket.type`

The socket type.

`socket.proto`

The socket protocol.

18.2.4 Notes on socket timeouts

A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are by default always created in blocking mode, but this can be changed by calling `setdefaulttimeout()`.

- 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: functions from the `select` can be used to know when and whether a socket is available for reading or writing.
- In *timeout mode*, operations fail if they cannot be completed within the timeout specified for the socket (they raise a `timeout` exception) or if the system returns an error.

Nota: At the operating system level, sockets in *timeout mode* are internally set in non-blocking mode. Also, the blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. This implementation detail can have visible consequences if e.g. you decide to use the `fileno()` of a socket.

Timeouts and the `connect` method

The `connect()` operation is also 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()`. However, the system network stack may also return a connection timeout error of its own regardless of any Python socket timeout setting.

Timeouts and the `accept` method

If `getdefaulttimeout()` is not `None`, sockets returned by the `accept()` method inherit that timeout. Otherwise, the behaviour depends on settings of the listening socket:

- if the listening socket is in *blocking mode* or in *timeout mode*, the socket returned by `accept()` is in *blocking mode*;
- if the listening socket is in *non-blocking mode*, whether the socket returned by `accept()` is in blocking or non-blocking mode is operating system-dependent. If you want to ensure cross-platform behaviour, it is recommended you manually override this setting.

18.2.5 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
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl' # The remote host
PORT = 50007           # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
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 OSError as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
```

(continué en la próxima página)

(proviene de la página anterior)

```

        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)

```

```

# Echo client program
import socket
import sys

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 OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))

```

The next 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)

```

(continué en la próxima página)

(proviene de la página anterior)

```
# receive a package
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

The next example shows how to use the socket interface to communicate to a CAN network using the raw socket protocol. To use CAN with the broadcast manager protocol instead, open a socket with:

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

After binding (`CAN_RAW`) or connecting (`CAN_BCM`) the socket, you can use the `socket.send()`, and the `socket.recv()` operations (and their counterparts) on the socket object as usual.

This last example might require special privileges:

```
import socket
import struct

# CAN frame packing/unpacking (see 'struct can_frame' in <linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])

# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

Running an example several times with too small delay between executions, could lead to this error:

```
OSError: [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.

Ver también:

For an introduction to socket programming (in C), see the following papers:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*, by Stuart Sechrest
- *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.

18.3 `ssl` — TLS/SSL wrapper for socket objects

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.

Nota: 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.

Advertencia: 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.

Distinto en la versión 3.5.3: Updated to support linking with OpenSSL 1.1.0

Distinto en la versión 3.6: OpenSSL 0.9.8, 1.0.0 and 1.0.1 are deprecated and no longer supported. In the future the `ssl` module will require at least OpenSSL 1.0.2 or 1.1.0.

18.3.1 Functions, Constants, and Exceptions

Socket creation

Since Python 3.2 and 2.7.9, it is recommended to use the `SSLContext.wrap_socket()` of an `SSLContext` instance to wrap sockets as `SSLSocket` objects. The helper functions `create_default_context()` returns a new context with secure default settings. The old `wrap_socket()` function is deprecated since it is both inefficient and has no support for server name indication (SNI) and hostname matching.

Client socket example with default context and IPv4/IPv6 dual stack:

```
import socket
import ssl

hostname = 'www.python.org'
context = ssl.create_default_context()

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

Client socket example with custom context and IPv4:

```
hostname = 'www.python.org'
# PROTOCOL_TLS_CLIENT requires valid cert chain and hostname
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('path/to/cabundle.pem')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

Server socket example listening on localhost IPv4:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain('/path/to/certchain.pem', '/path/to/private.key')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    sock.bind(('127.0.0.1', 8443))
    sock.listen(5)
    with context.wrap_socket(sock, server_side=True) as ssock:
        conn, addr = ssock.accept()
        ...
```

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_TLS`, `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.

When `keylog_filename` is supported and the environment variable `SSLKEYLOGFILE` is set, `create_default_context()` enables key logging.

Nota: 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.

Nota: 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
```

Nuevo en la versión 3.4.

Distinto en la versión 3.4.4: RC4 was dropped from the default cipher string.

Distinto en la versión 3.6: ChaCha20/Poly1305 was added to the default cipher string.

3DES was dropped from the default cipher string.

Distinto en la versión 3.8: Support for key logging to `SSLKEYLOGFILE` was added.

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 `OSError`. The error code and message of `SSLError` instances are provided by the OpenSSL library.

Distinto en la versión 3.3: `SSLError` used to be a subtype of `socket.error`.

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.

Nuevo en la versión 3.3.

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.

Nuevo en la versión 3.3.

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.

Nuevo en la versión 3.3.

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.

Nuevo en la versión 3.3.

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.

Nuevo en la versión 3.3.

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.

Nuevo en la versión 3.3.

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.

Nuevo en la versión 3.3.

exception `ssl.SSLCertVerificationError`

A subclass of `SSLError` raised when certificate validation has failed.

Nuevo en la versión 3.7.

verify_code

A numeric error number that denotes the verification error.

verify_message

A human readable string of the verification error.

exception `ssl.CertificateError`

An alias for `SSLCertVerificationError`.

Distinto en la versión 3.7: The exception is now an alias for `SSLCertVerificationError`.

Random generation

`ssl.RAND_bytes (num)`

Return *num* cryptographically strong pseudo-random bytes. Raises an `SSLError` if the PRNG has not been seeded with enough data or if the operation is not supported by the current RAND method. `RAND_status()` can be used to check the status of the PRNG and `RAND_add()` can be used to seed the PRNG.

For almost all applications `os.urandom()` is preferable.

Read the Wikipedia article, [Cryptographically secure pseudorandom number generator \(CSPRNG\)](#), to get the requirements of a cryptographically generator.

Nuevo en la versión 3.3.

`ssl.RAND_pseudo_bytes (num)`

Return (bytes, is_cryptographic): bytes are *num* pseudo-random bytes, is_cryptographic is `True` if the bytes generated are cryptographically strong. Raises an `SSLError` if the operation is not supported by the current RAND method.

Generated pseudo-random byte sequences will be unique if they are of sufficient length, but are not necessarily unpredictable. They can be used for non-cryptographic purposes and for certain purposes in cryptographic protocols, but usually not for key generation etc.

For almost all applications `os.urandom()` is preferable.

Nuevo en la versión 3.3.

Obsoleto desde la versión 3.6: 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.

Distinto en la versión 3.5: Writable *bytes-like object* is now accepted.

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](#), [RFC 5280](#) and [RFC 6125](#). 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'
```

Nuevo en la versión 3.2.

Distinto en la versión 3.3.3: The function now follows [RFC 6125](#), section 6.4.3 and does neither match multiple wildcards (e.g. `*.*.com` or `*a*.example.org`) nor a wildcard inside an internationalized domain names (IDN) fragment. IDN A-labels such as `www*.xn--python-kva.org` are still supported, but `x*.python.org` no longer matches `xn--tda.python.org`.

Distinto en la versión 3.5: Matching of IP addresses, when present in the `subjectAltName` field of the certificate, is now supported.

Distinto en la versión 3.7: The function is no longer used to TLS connections. Hostname matching is now performed by OpenSSL.

Allow wildcard when it is the leftmost and the only character in that segment. Partial wildcards like `www*.example.com` are no longer supported.

Obsoleto desde la versión 3.7.

`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](#)).

Distinto en la versión 3.5: 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_TLS, 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 `SSLContext.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.

Distinto en la versión 3.3: This function is now IPv6-compatible.

Distinto en la versión 3.5: The default `ssl_version` is changed from `PROTOCOL_SSLv3` to `PROTOCOL_TLS` 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`.

Nuevo en la versión 3.4.

`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.

Nuevo en la versión 3.4.

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.

Nuevo en la versión 3.4.

ssl.wrap_socket (*sock*, *keyfile=None*, *certfile=None*, *server_side=False*, *cert_reqs=CERT_NONE*, *ssl_version=PROTOCOL_TLS*, *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.

Internally, function creates a `SSLContext` with protocol *ssl_version* and `SSLContext.options` set to *cert_reqs*. If parameters *keyfile*, *certfile*, *ca_certs* or *ciphers* are set, then the values are passed to `SSLContext.load_cert_chain()`, `SSLContext.load_verify_locations()`, and `SSLContext.set_ciphers()`.

The arguments *server_side*, *do_handshake_on_connect*, and *suppress_ragged_eofs* have the same meaning as `SSLContext.wrap_socket()`.

Obsoleto desde la versión 3.7: Since Python 3.2 and 2.7.9, it is recommended to use the `SSLContext.wrap_socket()` instead of `wrap_socket()`. The top-level function is limited and creates an insecure client socket without server name indication or hostname matching.

Constants

All constants are now `enum.IntEnum` or `enum.IntFlag` collections.

Nuevo en la versión 3.6.

ssl.CERT_NONE

Possible value for `SSLContext.verify_mode`, or the *cert_reqs* parameter to `wrap_socket()`. Except for `PROTOCOL_TLS_CLIENT`, it is the default mode. With client-side sockets, just about any cert is accepted. Validation errors, such as untrusted or expired cert, are ignored and do not abort the TLS/SSL handshake.

In server mode, no certificate is requested from the client, so the client does not send any for client cert authentication.

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 client mode, `CERT_OPTIONAL` has the same meaning as `CERT_REQUIRED`. It is recommended to use `CERT_REQUIRED` for client-side sockets instead.

In server mode, a client certificate request is sent to the client. The client may either ignore the request or send a certificate in order perform TLS client cert authentication. If the client chooses to send a certificate, it is verified. Any verification error immediately aborts the TLS handshake.

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. This mode is **not** sufficient to verify a certificate in client mode as it does not match hostnames. `check_hostname` must be enabled as well to verify the authenticity of a cert. `PROTOCOL_TLS_CLIENT` uses `CERT_REQUIRED` and enables `check_hostname` by default.

With server socket, this mode provides mandatory TLS client cert authentication. A client certificate request is sent to the client and the client must provide a valid and trusted certificate.

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()`.

`class ssl.VerifyMode`

`enum.IntEnum` collection of `CERT_*` constants.

Nuevo en la versión 3.6.

`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.

Nuevo en la versión 3.4.

`ssl.VERIFY_CRL_CHECK_LEAF`

Possible value for `SSLContext.verify_flags`. In this mode, only the peer cert is checked but none 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 CRL has been loaded with `SSLContext.load_verify_locations`, validation will fail.

Nuevo en la versión 3.4.

`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.

Nuevo en la versión 3.4.

`ssl.VERIFY_X509_STRICT`

Possible value for `SSLContext.verify_flags` to disable workarounds for broken X.509 certificates.

Nuevo en la versión 3.4.

`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.

Nuevo en la versión 3.4.4.

`class ssl.VerifyFlags`

`enum.IntFlag` collection of `VERIFY_*` constants.

Nuevo en la versión 3.6.

`ssl.PROTOCOL_TLS`

Selects the highest protocol version that both the client and server support. Despite the name, this option can select both «SSL» and «TLS» protocols.

Nuevo en la versión 3.6.

`ssl.PROTOCOL_TLS_CLIENT`

Auto-negotiate the highest protocol version like `PROTOCOL_TLS`, but only support client-side `SSLSocket` connections. The protocol enables `CERT_REQUIRED` and `check_hostname` by default.

Nuevo en la versión 3.6.

ssl.PROTOCOL_TLS_SERVER

Auto-negotiate the highest protocol version like *PROTOCOL_TLS*, but only support server-side *SSLSocket* connections.

Nuevo en la versión 3.6.

ssl.PROTOCOL_SSLv23

Alias for *PROTOCOL_TLS*.

Obsoleto desde la versión 3.6: 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.

Advertencia: SSL version 2 is insecure. Its use is highly discouraged.

Obsoleto desde la versión 3.6: 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.

Advertencia: SSL version 3 is insecure. Its use is highly discouraged.

Obsoleto desde la versión 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* with flags like *OP_NO_SSLv3* instead.

ssl.PROTOCOL_TLSv1

Selects TLS version 1.0 as the channel encryption protocol.

Obsoleto desde la versión 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* 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+.

Nuevo en la versión 3.4.

Obsoleto desde la versión 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* 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+.

Nuevo en la versión 3.4.

Obsoleto desde la versión 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* 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.

Nuevo en la versión 3.2.

ssl.OP_NO_SSLv2

Prevents an SSLv2 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing SSLv2 as the protocol version.

Nuevo en la versión 3.2.

Obsoleto desde la versión 3.6: SSLv2 is deprecated

ssl.OP_NO_SSLv3

Prevents an SSLv3 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing SSLv3 as the protocol version.

Nuevo en la versión 3.2.

Obsoleto desde la versión 3.6: SSLv3 is deprecated

ssl.OP_NO_TLSv1

Prevents a TLSv1 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing TLSv1 as the protocol version.

Nuevo en la versión 3.2.

Obsoleto desde la versión 3.7: The option is deprecated since OpenSSL 1.1.0, use the new `SSLContext.minimum_version` and `SSLContext.maximum_version` instead.

ssl.OP_NO_TLSv1_1

Prevents a TLSv1.1 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing TLSv1.1 as the protocol version. Available only with openssl version 1.0.1+.

Nuevo en la versión 3.4.

Obsoleto desde la versión 3.7: The option is deprecated since OpenSSL 1.1.0.

ssl.OP_NO_TLSv1_2

Prevents a TLSv1.2 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing TLSv1.2 as the protocol version. Available only with openssl version 1.0.1+.

Nuevo en la versión 3.4.

Obsoleto desde la versión 3.7: The option is deprecated since OpenSSL 1.1.0.

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.

Nuevo en la versión 3.7.

Obsoleto desde la versión 3.7: The option is deprecated since OpenSSL 1.1.0. It was added to 2.7.15, 3.6.3 and 3.7.0 for backwards compatibility with OpenSSL 1.0.2.

ssl.OP_NO_RENEGOTIATION

Disable all renegotiation in TLSv1.2 and earlier. Do not send HelloRequest messages, and ignore renegotiation requests via ClientHello.

This option is only available with OpenSSL 1.1.0h and later.

Nuevo en la versión 3.7.

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.

Nuevo en la versión 3.3.

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.

Nuevo en la versión 3.3.

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.

Nuevo en la versión 3.3.

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.

Nuevo en la versión 3.8.

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.

Nuevo en la versión 3.3.

class ssl.Options

enum.IntFlag collection of OP_* constants.

ssl.OP_NO_TICKET

Prevent client side from requesting a session ticket.

Nuevo en la versión 3.6.

ssl.OP_IGNORE_UNEXPECTED_EOF

Ignore unexpected shutdown of TLS connections.

This option is only available with OpenSSL 3.0.0 and later.

Nuevo en la versión 3.10.

ssl.HAS_ALPN

Whether the OpenSSL library has built-in support for the *Application-Layer Protocol Negotiation* TLS extension as described in [RFC 7301](#).

Nuevo en la versión 3.5.

ssl.HAS_NEVER_CHECK_COMMON_NAME

Whether the OpenSSL library has built-in support not checking subject common name and *SSLContext.hostname_checks_common_name* is writeable.

Nuevo en la versión 3.7.

ssl.HAS_ECDH

Whether the OpenSSL library has built-in support for the Elliptic Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor.

Nuevo en la versión 3.3.

ssl.HAS_SNI

Whether the OpenSSL library has built-in support for the *Server Name Indication* extension (as defined in [RFC 6066](#)).

Nuevo en la versión 3.2.

ssl.HAS_NPN

Whether the OpenSSL library has built-in support for the *Next Protocol Negotiation* as described in the [Application Layer Protocol Negotiation](#). When true, you can use the *SSLContext.set_npn_protocols()* method to advertise which protocols you want to support.

Nuevo en la versión 3.3.

ssl.HAS_SSLv2

Whether the OpenSSL library has built-in support for the SSL 2.0 protocol.

Nuevo en la versión 3.7.

ssl.HAS_SSLv3

Whether the OpenSSL library has built-in support for the SSL 3.0 protocol.

Nuevo en la versión 3.7.

`ssl.HAS_TLSv1`

Whether the OpenSSL library has built-in support for the TLS 1.0 protocol.

Nuevo en la versión 3.7.

`ssl.HAS_TLSv1_1`

Whether the OpenSSL library has built-in support for the TLS 1.1 protocol.

Nuevo en la versión 3.7.

`ssl.HAS_TLSv1_2`

Whether the OpenSSL library has built-in support for the TLS 1.2 protocol.

Nuevo en la versión 3.7.

`ssl.HAS_TLSv1_3`

Whether the OpenSSL library has built-in support for the TLS 1.3 protocol.

Nuevo en la versión 3.7.

`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()`.

Nuevo en la versión 3.3.

`ssl.OPENSSSL_VERSION`

The version string of the OpenSSL library loaded by the interpreter:

```
>>> ssl.OPENSSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

Nuevo en la versión 3.2.

`ssl.OPENSSSL_VERSION_INFO`

A tuple of five integers representing version information about the OpenSSL library:

```
>>> ssl.OPENSSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

Nuevo en la versión 3.2.

`ssl.OPENSSSL_VERSION_NUMBER`

The raw version number of the OpenSSL library, as a single integer:

```
>>> ssl.OPENSSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSSL_VERSION_NUMBER)
'0x100020bf'
```

Nuevo en la versión 3.2.

`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()`.

Nuevo en la versión 3.4.

`class ssl.AlertDescription`

`enum.IntEnum` collection of `ALERT_DESCRIPTION_*` constants.

Nuevo en la versión 3.6.

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).

Nuevo en la versión 3.4.

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).

Nuevo en la versión 3.4.

class ssl.SSLErrorNumber

`enum.IntEnum` collection of `SSL_ERROR_*` constants.

Nuevo en la versión 3.6.

class ssl.TLSVersion

`enum.IntEnum` collection of SSL and TLS versions for `SSLContext.maximum_version` and `SSLContext.minimum_version`.

Nuevo en la versión 3.7.

TLSVersion.MINIMUM_SUPPORTED**TLSVersion.MAXIMUM_SUPPORTED**

The minimum or maximum supported SSL or TLS version. These are magic constants. Their values don't reflect the lowest and highest available TLS/SSL versions.

TLSVersion.SSLv3**TLSVersion.TLSv1****TLSVersion.TLSv1_1****TLSVersion.TLSv1_2****TLSVersion.TLSv1_3**

SSL 3.0 to TLS 1.3.

18.3.2 SSL Sockets

class ssl.SSLSocket (*socket.socket*)

SSL sockets provide the following methods of *Socket Objects*:

- `accept()`
- `bind()`
- `close()`
- `connect()`
- `detach()`
- `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)
- `sendfile()` (but `os.sendfile` will be used for plain-text sockets only, else `send()` will be used)
- `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*.

Instances of `SSLSocket` must be created using the `SSLContext.wrap_socket()` method.

Distinto en la versión 3.5: The `sendfile()` method was added.

Distinto en la versión 3.5: The `shutdown()` does not reset the socket timeout each time bytes are received or sent. The socket timeout is now to maximum total duration of the shutdown.

Obsoleto desde la versión 3.6: It is deprecated to create a `SSLSocket` instance directly, use `SSLContext.wrap_socket()` to wrap a socket.

Distinto en la versión 3.7: `SSLSocket` instances must to created with `wrap_socket()`. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

SSL sockets also have the following additional methods and attributes:

`SSLSocket.read(len=1024, buffer=None)`

Read up to `len` bytes of data from the SSL socket and return the result as a `bytes` instance. If `buffer` is specified, then read into the buffer instead, and return the number of bytes read.

Raise `SSLWantReadError` or `SSLWantWriteError` if the socket is *non-blocking* and the read would block.

As at any time a re-negotiation is possible, a call to `read()` can also cause write operations.

Distinto en la versión 3.5: The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration to read up to `len` bytes.

Obsoleto desde la versión 3.6: Use `recv()` instead of `read()`.

`SSLSocket.write(buf)`

Write `buf` to the SSL socket and return the number of bytes written. The `buf` argument must be an object supporting the buffer interface.

Raise `SSLWantReadError` or `SSLWantWriteError` if the socket is *non-blocking* and the write would block.

As at any time a re-negotiation is possible, a call to `write()` can also cause read operations.

Distinto en la versión 3.5: The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration to write `buf`.

Obsoleto desde la versión 3.6: Use `send()` instead of `write()`.

Nota: The `read()` and `write()` methods are the low-level methods that read and write unencrypted, application-level data and decrypt/encrypt it to encrypted, wire-level data. These methods require an active SSL connection, i.e. the handshake was completed and `SSLSocket.unwrap()` was not called.

Normally you should use the socket API methods like `recv()` and `send()` instead of these methods.

`SSLSocket.do_handshake()`

Perform the SSL setup handshake.

Distinto en la versión 3.4: The handshake method also performs `match_hostname()` when the `check_hostname` attribute of the socket's `context` is true.

Distinto en la versión 3.5: The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration of the handshake.

Distinto en la versión 3.7: Hostname or IP address is matched by OpenSSL during handshake. The function `match_hostname()` is no longer used. In case OpenSSL refuses a hostname or IP address, the handshake is aborted early and a TLS alert message is send to the peer.

`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}
```

Nota: 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`).

Distinto en la versión 3.2: The returned dictionary includes additional items such as `issuer` and `notBefore`.

Distinto en la versión 3.4: `ValueError` is raised when the handshake isn't done. The returned dictionary includes additional X509v3 extension items such as `crlDistributionPoints`, `caIssuers` and `OCSP` URIs.

Distinto en la versión 3.8.1: IPv6 address strings no longer have a trailing new line.

`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.shared_ciphers()

Return the list of ciphers shared by the client during the handshake. Each entry of the returned list is a three-value tuple containing the name of the cipher, the version of the SSL protocol that defines its use, and the number of secret bits the cipher uses. `shared_ciphers()` returns `None` if no connection has been established or the socket is a client socket.

Nuevo en la versión 3.5.

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.

Nuevo en la versión 3.3.

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.

Nuevo en la versión 3.3.

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.

Nuevo en la versión 3.5.

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`.

Nuevo en la versión 3.3.

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.verify_client_post_handshake()

Requests post-handshake authentication (PHA) from a TLS 1.3 client. PHA can only be initiated for a TLS 1.3 connection from a server-side socket, after the initial TLS handshake and with PHA enabled on both sides, see `SSLContext.post_handshake_auth`.

The method does not perform a cert exchange immediately. The server-side sends a `CertificateRequest` during the next write event and expects the client to respond with a certificate on the next read event.

If any precondition isn't met (e.g. not TLS 1.3, PHA not enabled), an `SSL_ERROR` is raised.

Nota: Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the method raises `NotImplementedError`.

Nuevo en la versión 3.8.

SSLSocket.version()

Return the actual SSL protocol version negotiated by the connection as a string, or `None` is no secure con-

nection 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.

Nuevo en la versión 3.5.

`SSLSocket.pending()`

Returns the number of already decrypted bytes available for read, pending on the connection.

`SSLSocket.context`

The `SSLContext` object this SSL socket is tied to. If the SSL socket was created using the deprecated `wrap_socket()` function (rather than `SSLContext.wrap_socket()`), this is a custom context object created for this SSL socket.

Nuevo en la versión 3.2.

`SSLSocket.server_side`

A boolean which is `True` for server-side sockets and `False` for client-side sockets.

Nuevo en la versión 3.2.

`SSLSocket.server_hostname`

Hostname of the server: `str` type, or `None` for server-side socket or if the hostname was not specified in the constructor.

Nuevo en la versión 3.2.

Distinto en la versión 3.7: The attribute is now always ASCII text. When `server_hostname` is an internationalized domain name (IDN), this attribute now stores the A-label form ("`xn--pythn-mua.org`"), rather than the U-label form ("`pythön.org`").

`SSLSocket.session`

The `SSLSession` for this SSL connection. The session is available for client and server side sockets after the TLS handshake has been performed. For client sockets the session can be set before `do_handshake()` has been called to reuse a session.

Nuevo en la versión 3.6.

`SSLSocket.session_reused`

Nuevo en la versión 3.6.

18.3.3 SSL Contexts

Nuevo en la versión 3.2.

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=PROTOCOL_TLS*)

Create a new SSL context. You may pass *protocol* which must be one of the `PROTOCOL_*` constants defined in this module. The parameter 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_TLS`; 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):

<i>client / server</i>	SSLv2	SSLv3	TLS ³	TLSv1	TLSv1.1	TLSv1.2
SSLv2	yes	no	no ¹	no	no	no
SSLv3	no	yes	no ²	no	no	no
TLS (SSLv23) ³	no ¹	no ²	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

Ver también:

`create_default_context()` lets the `ssl` module choose security settings for a given purpose.

Distinto en la versión 3.6: 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}
```

Nuevo en la versión 3.4.

`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.

Distinto en la versión 3.3: New optional argument `password`.

`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.

³ TLS 1.3 protocol will be available with `PROTOCOL_TLS` in OpenSSL \geq 1.1.1. There is no dedicated PROTOCOL constant for just TLS 1.3.

¹ `SSLContext` disables SSLv2 with `OP_NO_SSLv2` by default.

² `SSLContext` disables SSLv3 with `OP_NO_SSLv3` by default.

Nuevo en la versión 3.4.

`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.

Distinto en la versión 3.4: New optional argument *cadata*

`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.

Nota: Certificates in a *capath* directory aren't loaded unless they have been used at least once.

Nuevo en la versión 3.4.

`SSLContext.get_ciphers` ()

Get a list of enabled ciphers. The list is in order of cipher priority. See `SSLContext.set_ciphers()`.

Example:

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDH')
>>> ctx.get_ciphers() # OpenSSL 1.0.x
[{'alg_bits': 256,
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH      Au=RSA  '
                  'Enc=AESGCM(256) Mac=AEAD',
  'id': 50380848,
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 256},
 {'alg_bits': 128,
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH      Au=RSA  '
                  'Enc=AESGCM(128) Mac=AEAD',
  'id': 50380847,
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 128}]
```

On OpenSSL 1.1 and newer the cipher dict contains additional fields:

```
>>> ctx.get_ciphers() # OpenSSL 1.1+
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH      Au=RSA  '
                  'Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
```

(continué en la próxima página)

(proviene de la página anterior)

```

'id': 50380848,
'kea': 'kx-ecdhe',
'name': 'ECDHE-RSA-AES256-GCM-SHA384',
'protocol': 'TLSv1.2',
'strength_bits': 256,
'symmetric': 'aes-256-gcm'},
{'aead': True,
'alg_bits': 128,
'auth': 'auth-rsa',
'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH      Au=RSA      '
                'Enc=AESGCM(128) Mac=AEAD',
'digest': None,
'id': 50380847,
'kea': 'kx-ecdhe',
'name': 'ECDHE-RSA-AES128-GCM-SHA256',
'protocol': 'TLSv1.2',
'strength_bits': 128,
'symmetric': 'aes-128-gcm'}}]

```

Availability: OpenSSL 1.0.2+.

Nuevo en la versión 3.6.

`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.

Nota: 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.

Nuevo en la versión 3.5.

`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 [Application Layer Protocol Negotiation](#). 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.

Nuevo en la versión 3.3.

`SSLContext.sni_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 `sni_callback` is set to `None` then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function 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 text. For internationalized domain name, the server name is an IDN A-label (`"xn--pythn-mua.org"`).

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 `sni_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 an exception is raised from the `sni_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.

Nuevo en la versión 3.7.

`SSLContext.set_servername_callback(server_name_callback)`

This is a legacy API retained for backwards compatibility. When possible, you should use `sni_callback` instead. The given `server_name_callback` is similar to `sni_callback`, except that when the server hostname is an IDN-encoded internationalized domain name, the `server_name_callback` receives a decoded U-label (`"python.org"`).

If there is an decoding error on the server name, the TLS connection will terminate with an `ALERT_DESCRIPTION_INTERNAL_ERROR` fatal TLS alert message to the client.

Nuevo en la versión 3.4.

`SSLContext.load_dh_params(dhfile)`

Load the key generation parameters for Diffie-Hellman (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.

Nuevo en la versión 3.3.

`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`.

Nuevo en la versión 3.3.

Ver también:

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, session=None)`
Wrap an existing Python socket `sock` and return an instance of `SSLContext.sslsocket_class` (default `SSLSocket`). The returned SSL socket is tied to the context, its settings and certificates. `sock` must be a `SOCK_STREAM` socket; other socket types are unsupported.

The parameter `server_side` is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

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. The method may raise `SSL_ERROR`.

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.

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.recv()` 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.

`session`, see `session`.

Distinto en la versión 3.5: Always allow a `server_hostname` to be passed, even if OpenSSL does not have SNI.

Distinto en la versión 3.6: `session` argument was added.

Distinto en la versión 3.7: The method returns on instance of `SSLContext.sslsocket_class` instead of hard-coded `SSLSocket`.

`SSLContext.sslsocket_class`

The return type of `SSLContext.wrap_socket()`, defaults to `SSLSocket`. The attribute can be overridden on instance of class in order to return a custom subclass of `SSLSocket`.

Nuevo en la versión 3.7.

`SSLContext.wrap_bio(incoming, outgoing, server_side=False, server_hostname=None, session=None)`

Wrap the BIO objects `incoming` and `outgoing` and return an instance of `SSLContext.sslobject_class` (default `SSLObject`). The SSL routines will read input data from the incoming BIO and write data to the outgoing BIO.

The `server_side`, `server_hostname` and `session` parameters have the same meaning as in `SSLContext.wrap_socket()`.

Distinto en la versión 3.6: `session` argument was added.

Distinto en la versión 3.7: The method returns on instance of `SSLContext.sslobject_class` instead of hard-coded `SSLObject`.

`SSLContext.sslobject_class`

The return type of `SSLContext.wrap_bio()`, defaults to `SSLObject`. The attribute can be overridden on instance of class in order to return a custom subclass of `SSLObject`.

Nuevo en la versión 3.7.

`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`

Whether to match the peer cert's 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. Enabling hostname checking automatically sets `verify_mode` from `CERT_NONE` to `CERT_REQUIRED`. It cannot be set back to `CERT_NONE` as long as hostname checking is enabled. The `PROTOCOL_TLS_CLIENT` protocol enables hostname checking by default. With other protocols, hostname checking must be enabled explicitly.

Example:

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
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))
```

Nuevo en la versión 3.4.

Distinto en la versión 3.7: `verify_mode` is now automatically changed to `CERT_REQUIRED` when hostname checking is enabled and `verify_mode` is `CERT_NONE`. Previously the same operation would have failed with a `ValueError`.

Nota: This features requires OpenSSL 0.9.8f or newer.

`SSLContext.keylog_filename`

Write TLS keys to a keylog file, whenever key material is generated or received. The keylog file is designed for debugging purposes only. The file format is specified by NSS and used by many traffic analyzers such as Wireshark. The log file is opened in append-only mode. Writes are synchronized between threads, but not between processes.

Nuevo en la versión 3.8.

Nota: This features requires OpenSSL 1.1.1 or newer.

`SSLContext.maximum_version`

A `TLSVersion` enum member representing the highest supported TLS version. The value defaults to `TLSVersion.MAXIMUM_SUPPORTED`. The attribute is read-only for protocols other than `PROTOCOL_TLS`, `PROTOCOL_TLS_CLIENT`, and `PROTOCOL_TLS_SERVER`.

The attributes `maximum_version`, `minimum_version` and `SSLContext.options` all affect the supported SSL and TLS versions of the context. The implementation does not prevent invalid combination. For example a context with `OP_NO_TLSv1_2` in `options` and `maximum_version` set to `TLSVersion.TLSv1_2` will not be able to establish a TLS 1.2 connection.

Nota: This attribute is not available unless the ssl module is compiled with OpenSSL 1.1.0g or newer.

Nuevo en la versión 3.7.

`SSLContext.minimum_version`

Like `SSLContext.maximum_version` except it is the lowest supported version or `TLSVersion.MINIMUM_SUPPORTED`.

Nota: This attribute is not available unless the ssl module is compiled with OpenSSL 1.1.0g or newer.

Nuevo en la versión 3.7.

`SSLContext.num_tickets`

Control the number of TLS 1.3 session tickets of a `TLS_PROTOCOL_SERVER` context. The setting has no impact on TLS 1.0 to 1.2 connections.

Nota: This attribute is not available unless the ssl module is compiled with OpenSSL 1.1.1 or newer.

Nuevo en la versión 3.8.

`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.

Nota: 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`.

Distinto en la versión 3.6: `SSLContext.options` returns `Options` flags:

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

`SSLContext.post_handshake_auth`

Enable TLS 1.3 post-handshake client authentication. Post-handshake auth is disabled by default and a server can only request a TLS client certificate during the initial handshake. When enabled, a server may request a TLS client certificate at any time after the handshake.

When enabled on client-side sockets, the client signals the server that it supports post-handshake authentication.

When enabled on server-side sockets, `SSLContext.verify_mode` must be set to `CERT_OPTIONAL` or `CERT_REQUIRED`, too. The actual client cert exchange is delayed until `SSLSocket.verify_client_post_handshake()` is called and some I/O is performed.

Nota: Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the property value is `None` and can't be modified

Nuevo en la versión 3.8.

`SSLContext.protocol`

The protocol version chosen when constructing the context. This attribute is read-only.

`SSLContext.hostname_checks_common_name`

Whether `check_hostname` falls back to verify the cert's subject common name in the absence of a subject alternative name extension (default: `true`).

Nota: Only writeable with OpenSSL 1.1.0 or higher.

Nuevo en la versión 3.7.

Distinto en la versión 3.9.3: The flag had no effect with OpenSSL before version 1.1.1k. Python 3.8.9, 3.9.3, and 3.10 include workarounds for previous versions.

`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+.

Nuevo en la versión 3.4.

Distinto en la versión 3.6: `SSLContext.verify_flags` returns `VerifyFlags` flags:

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

`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`.

Distinto en la versión 3.6: `SSLContext.verify_mode` returns `VerifyMode` enum:

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

18.3.4 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.
```

(continué en la próxima página)

(proviene de la página anterior)

```

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.

18.3.5 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:
    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_CLIENT)
>>> 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)

The `PROTOCOL_TLS_CLIENT` protocol configures the context for cert validation and hostname verification. `verify_mode` is set to `CERT_REQUIRED` and `check_hostname` is set to `True`. All other protocols create SSL contexts with insecure defaults.

When you use the context to connect to a server, `CERT_REQUIRED` and `check_hostname` validate the server certificate: it ensures that the server certificate was signed with one of the CA certificates, checks the signature for correctness, and verifies other properties like validity and identity of the hostname:

```
>>> 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'),),
                (('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.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',
```

(continué en la próxima página)

(proviene de la página anterior)

```
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)
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.recv(1024)
    # empty 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.recv(1024)
    # 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).

18.3.6 Notes on non-blocking sockets

SSL sockets behave slightly different than regular sockets in non-blocking mode. When working with non-blocking sockets, there are thus several things you need to be aware of:

- Most `SSLSocket` methods will raise either `SSLWantWriteError` or `SSLWantReadError` instead of `BlockingIOError` if an I/O operation would block. `SSLWantReadError` will be raised if a read operation on the underlying socket is necessary, and `SSLWantWriteError` for a write operation on the underlying socket. Note that attempts to *write* to an SSL socket may require *reading* from the underlying socket first, and attempts to *read* from the SSL socket may require a prior *write* to the underlying socket.

Distinto en la versión 3.5: In earlier Python versions, the `SSLSocket.send()` method returned zero instead of raising `SSLWantWriteError` or `SSLWantReadError`.

- 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], [])
```

Ver también:

The `asyncio` module supports *non-blocking SSL sockets* and provides a higher level API. It polls for events using the `selectors` module and handles `SSLWantWriteError`, `SSLWantReadError` and `BlockingIOError` exceptions. It runs the SSL handshake asynchronously as well.

18.3.7 Memory BIO Support

Nuevo en la versión 3.5.

Ever since the SSL module was introduced in Python 2.6, the `SSLSocket` class has provided two related but distinct areas of functionality:

- SSL protocol handling
- Network IO

The network IO API is identical to that provided by `socket.socket`, from which `SSLSocket` also inherits. This allows an SSL socket to be used as a drop-in replacement for a regular socket, making it very easy to add SSL support to an existing application.

Combining SSL protocol handling and network IO usually works well, but there are some cases where it doesn't. An example is async IO frameworks that want to use a different IO multiplexing model than the «select/poll on a file descriptor» (readiness based) model that is assumed by `socket.socket` and by the internal OpenSSL socket IO routines. This is mostly relevant for platforms like Windows where this model is not efficient. For this purpose, a reduced scope variant of `SSLSocket` called `SSLObject` is provided.

class `ssl.SSLObject`

A reduced-scope variant of `SSLSocket` representing an SSL protocol instance that does not contain any network IO methods. This class is typically used by framework authors that want to implement asynchronous IO for SSL through memory buffers.

This class implements an interface on top of a low-level SSL object as implemented by OpenSSL. This object captures the state of an SSL connection but does not provide any network IO itself. IO needs to be performed through separate «BIO» objects which are OpenSSL's IO abstraction layer.

This class has no public constructor. An `SSLObject` instance must be created using the `wrap_bio()` method. This method will create the `SSLObject` instance and bind it to a pair of BIOs. The *incoming* BIO is used to pass data from Python to the SSL protocol instance, while the *outgoing* BIO is used to pass data the other way around.

The following methods are available:

- `context`
- `server_side`
- `server_hostname`
- `session`
- `session_reused`
- `read()`
- `write()`
- `getpeercert()`
- `selected_alpn_protocol()`
- `selected_npn_protocol()`
- `cipher()`
- `shared_ciphers()`
- `compression()`
- `pending()`
- `do_handshake()`
- `verify_client_post_handshake()`
- `unwrap()`
- `get_channel_binding()`
- `version()`

When compared to `SSLSocket`, this object lacks the following features:

- Any form of network IO; `recv()` and `send()` read and write only to the underlying `MemoryBIO` buffers.
- There is no `do_handshake_on_connect` machinery. You must always manually call `do_handshake()` to start the handshake.
- There is no handling of `suppress_ragged_eofs`. All end-of-file conditions that are in violation of the protocol are reported via the `SSLEOFError` exception.
- The method `unwrap()` call does not return anything, unlike for an SSL socket where it returns the underlying socket.
- The `server_name_callback` callback passed to `SSLContext.set_servername_callback()` will get an `SSLObject` instance instead of a `SSLSocket` instance as its first parameter.

Some notes related to the use of `SSLObject`:

- All IO on an `SSLObject` is *non-blocking*. This means that for example `read()` will raise an `SSLWantReadError` if it needs more data than the incoming BIO has available.
- There is no module-level `wrap_bio()` call like there is for `wrap_socket()`. An `SSLObject` is always created via an `SSLContext`.

Distinto en la versión 3.7: `SSLObject` instances must be created with `wrap_bio()`. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

An `SSLObject` communicates with the outside world using memory buffers. The class `MemoryBIO` provides a memory buffer that can be used for this purpose. It wraps an OpenSSL memory BIO (Basic IO) object:

class `ssl.MemoryBIO`

A memory buffer that can be used to pass data between Python and an SSL protocol instance.

pending

Return the number of bytes currently in the memory buffer.

eof

A boolean indicating whether the memory BIO is current at the end-of-file position.

read (*n=-1*)

Read up to *n* bytes from the memory buffer. If *n* is not specified or negative, all bytes are returned.

write (*buf*)

Write the bytes from *buf* to the memory BIO. The *buf* argument must be an object supporting the buffer protocol.

The return value is the number of bytes written, which is always equal to the length of *buf*.

write_eof ()

Write an EOF marker to the memory BIO. After this method has been called, it is illegal to call `write()`. The attribute `eof` will become true after all data currently in the buffer has been read.

18.3.8 SSL session

Nuevo en la versión 3.6.

class `ssl.SSLSession`

Session object used by `session`.

id

time

timeout

ticket_lifetime_hint

has_ticket

18.3.9 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.

For example, here is how you would use the `smtplib.SMTP` class to create a trusted, secure connection to a SMTP server:

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

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.

Distinto en la versión 3.7: Hostname matchings is now performed by OpenSSL. Python no longer uses `match_hostname()`.

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.

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_TLS_CLIENT` or `PROTOCOL_TLS_SERVER` as the protocol version. SSLv2 and SSLv3 are disabled by default.

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.options |= ssl.OP_NO_TLSv1
>>> client_context.options |= ssl.OP_NO_TLSv1_1
```

The SSL context created above will only allow TLSv1.2 and later (if supported by your system) connections to a server. `PROTOCOL_TLS_CLIENT` implies certificate validation and hostname checks by default. You have to load certificates into the context.

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 3.2.3, 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 `SSLContext.get_ciphers()` or 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.

18.3.10 TLS 1.3

Nuevo en la versión 3.7.

Python has provisional and experimental support for TLS 1.3 with OpenSSL 1.1.1. The new protocol behaves slightly differently than previous version of TLS/SSL. Some new TLS 1.3 features are not yet available.

- TLS 1.3 uses a disjunct set of cipher suites. All AES-GCM and ChaCha20 cipher suites are enabled by default. The method `SSLContext.set_ciphers()` cannot enable or disable any TLS 1.3 ciphers yet, but `SSLContext.get_ciphers()` returns them.
- Session tickets are no longer sent as part of the initial handshake and are handled differently. `SSLSocket.session` and `SSLSession` are not compatible with TLS 1.3.
- Client-side certificates are also no longer verified during the initial handshake. A server can request a certificate at any time. Clients process certificate requests while they send or receive application data from the server.
- TLS 1.3 features like early data, deferred TLS client cert request, signature algorithm configuration, and rekeying are not supported yet.

18.3.11 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 \geq 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.

Ver también:

Class `socket.socket` Documentation of underlying `socket` class

SSL/TLS Strong Encryption: An Introduction Intro from the Apache HTTP Server documentation

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management
Steve Kent

RFC 4086: Randomness Requirements for Security Donald E., Jeffrey I. Schiller

RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile
D. Cooper

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 (DTLS)
IETF

Mozilla's Server Side TLS recommendations Mozilla

18.4 `select` — Esperando la finalización de E/S

Este módulo proporciona acceso a las funciones `select()` y `poll()` disponibles en la mayoría de los sistemas operativos, `devpoll()` disponible en Solaris y derivados, `epoll()` disponible en Linux 2.5+ y `kqueue()` disponible en la mayoría de BSD. Tenga en cuenta que en Windows, solo funciona para sockets; en otros sistemas operativos, también funciona para otros tipos de archivos (en particular, en Unix, funciona en tuberías). No se puede usar en archivos normales para determinar si un archivo ha crecido desde la última lectura.

Nota: El módulo `selectors` permite la multiplexación de E/S de alto nivel y eficiente, construida sobre las primitivas del módulo `select`. Se alienta a los usuarios a utilizar el módulo `selectors` en su lugar, a menos que quieran un control preciso sobre las primitivas utilizadas de nivel OS.

El módulo define lo siguiente:

exception `select.error`

Un alias en desuso de `OSError`.

Distinto en la versión 3.3: Siguiendo [PEP 3151](#), esta clase se convirtió en un alias de `OSError`.

`select.devpoll()`

(Solo se admite en Solaris y derivados). Retorna un objeto de sondeo `/dev/poll`; vea la sección [Objetos de sondeo /dev/poll](#) a continuación para conocer los métodos admitidos por los objetos `devpoll`.

Los objetos `devpoll()` están vinculados a la cantidad de descriptores de archivo permitidos en el momento de la creación de instancias. Si su programa reduce este valor, `devpoll()` fallará. Si su programa aumenta este valor, `devpoll()` puede retornar una lista incompleta de descriptores de archivos activos.

El nuevo descriptor del archivo es *non-inheritable*.

Nuevo en la versión 3.3.

Distinto en la versión 3.4: El nuevo descriptor de archivo ahora no es heredable.

`select.epoll(sizehint=-1, flags=0)`

(Solo se admite en Linux 2.5.44 y versiones posteriores). Retorna un objeto de sondeo de borde, que se puede usar como interfaz de disparo de nivel o de borde para eventos de E/S.

`sizehint` informa a `epoll` sobre el número esperado de eventos a ser registrados. Debe ser positivo o `-1` para usar el valor predeterminado. Solo se usa en sistemas más antiguos donde `epoll_create1()` no está disponible; de lo contrario no tiene ningún efecto (aunque su valor aún está marcado).

`flags` está en desuso y se ignora por completo. Sin embargo, cuando se proporciona, su valor debe ser `0` o `select.EPOLL_CLOEXEC`, de lo contrario, se generará `OSError`.

Consulte la sección [Objetos de sondeo de Edge y Level Trigger \(epoll\)](#) a continuación para conocer los métodos admitidos por los objetos `epolling`.

Los objetos `epoll` admiten el protocolo *context management*: cuando se usa en una declaración `with`, el nuevo descriptor de archivo se cierra automáticamente al final del bloque.

El nuevo descriptor del archivo es *non-inheritable*.

Distinto en la versión 3.3: Se agregó el parámetro `flags`.

Distinto en la versión 3.4: Se agregó soporte para la declaración `with`. El nuevo descriptor de archivo ahora no es heredable.

Obsoleto desde la versión 3.4: El parámetro `flags`. `select.EPOLL_CLOEXEC` se usa por defecto ahora. Use `os.set_inheritable()` para hacer que el descriptor de archivo sea heredable.

`select.poll()`

(No es compatible con todos los sistemas operativos). Retorna un objeto de sondeo, que admite registrar y

anular el registro de descriptores de archivo, y luego sondearlos para eventos de I/O; vea la sección *Sondeo de objetos* a continuación para conocer los métodos admitidos por los objetos de sondeo.

`select.kqueue()`

(Solo se admite en BSD). Retorna un objeto de cola del kernel; vea la sección *Objetos Kqueue* a continuación para conocer los métodos admitidos por los objetos kqueue.

El nuevo descriptor del archivo es *non-inheritable*.

Distinto en la versión 3.4: El nuevo descriptor de archivo ahora no es heredable.

`select.kevent(ident, filter=KQ_FILTER_READ, flags=KQ_EV_ADD, fflags=0, data=0, udata=0)`

(Solo se admite en BSD). Retorna un objeto de evento del kernel; vea la sección *Objetos Kevent* a continuación para conocer los métodos admitidos por los objetos kevent.

`select.select(rlist, wlist, xlist[, timeout])`

This is a straightforward interface to the Unix `select()` system call. The first three arguments are iterables of “waitable objects”: either integers representing file descriptors or objects with a parameterless method named `fileno()` returning such an integer:

- *rlist*: espera hasta que esté listo para leer
- *wlist*: espera hasta que esté listo para escribir
- *xlist*: espera una «condición excepcional» (consulte la página del manual para ver lo que su sistema considera tal condición)

Empty iterables are allowed, but acceptance of three empty iterables 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.

El valor de retorno es un triple de listas de objetos que están listos: subconjuntos de los primeros tres argumentos. Cuando se alcanza el tiempo de espera sin que esté listo un descriptor de archivo, se retornan tres listas vacías.

Among the acceptable object types in the iterables 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).

Nota: Los objetos de archivo en Windows no son admisibles, pero los sockets sí. En Windows, la función subyacente `select()` es proporcionada por la biblioteca WinSock, y no maneja los descriptores de archivo que no se originan en WinSock.

Distinto en la versión 3.5: La función ahora se vuelve a intentar con un tiempo de espera (*timeout*) recalculado cuando se interrumpe por una señal, excepto si el controlador de señal genera una excepción (ver **PEP 475** para la justificación), en lugar de generar *InterruptedError*.

`select.PIPE_BUF`

El número mínimo de bytes que se pueden escribir sin bloquear en una tubería cuando la tubería ha sido reportada como lista para escribir por `select()`, `poll()` u otra interfaz en este módulo. Esto no se aplica a otro tipo de objetos similares a archivos, como los sockets.

Este valor es garantizado por POSIX para ser menor a 512.

Availability: Unix

Nuevo en la versión 3.2.

18.4.1 Objetos de sondeo /dev/poll

Solaris y derivados tienen /dev/poll. Mientras que `select()` es $O(\text{descriptor de archivo más alto})$ y `poll()` es $O(\text{número de descriptores de archivo})$, /dev/poll es $O(\text{descriptores de archivo activos})$.

El comportamiento /dev/poll está muy cerca del estándar objeto `poll()`.

`devpoll.close()`

Cierra el descriptor de archivo del objeto de sondeo.

Nuevo en la versión 3.4.

`devpoll.closed`

True si el objeto de sondeo está cerrado.

Nuevo en la versión 3.4.

`devpoll.fileno()`

Retorna el número de descriptor de archivo del objeto de sondeo.

Nuevo en la versión 3.4.

`devpoll.register(fd[, eventmask])`

Registra un descriptor de archivo con el objeto de sondeo. Las futuras llamadas al método `poll()` comprobarán si el descriptor del archivo tiene algún evento I/O pendiente. `fd` puede ser un entero o un objeto con un método `fileno()` que retorna un entero. Los objetos de archivo implementan `Fileno()`, por lo que también pueden usarse como argumento.

`eventmask` es una máscara de bits opcional que describe el tipo de eventos que desea verificar. Las constantes son las mismas que con el objeto `poll()`. El valor predeterminado es una combinación de las constantes `POLLIN`, `POLLPRI` y `POLLOUT`.

Advertencia: Registra un descriptor de archivo que ya está registrado no es un error, pero el resultado no está definido. La acción apropiada es anular el registro o modificarlo primero. Esta es una diferencia importante en comparación con `poll()`.

`devpoll.modify(fd[, eventmask])`

Este método hace un `unregister()` seguido de `register()`. Es (un poco) más eficiente que hacer lo mismo explícitamente.

`devpoll.unregister(fd)`

Elimina un descriptor de archivo que está siendo rastreado por un objeto de sondeo. Al igual que el método `register()`, `fd` puede ser un entero o un objeto con un método `fileno()` que retorna un entero.

Al intentar eliminar un descriptor de archivo que nunca se registró se ignora de forma segura.

`devpoll.poll([timeout])`

Sondea el conjunto de descriptores de archivos registrados y retorna una lista posiblemente vacía que contiene `(fd, event)` 2 tuplas para los descriptores que tienen eventos o errores que informar. `fd` es el descriptor de archivo, y `event` es una máscara de bits con bits establecidos para los eventos informados para ese descriptor — `POLLIN` para la entrada en espera, `POLLOUT` para indicar que el descriptor puede ser escrito, y así sucesivamente. Una lista vacía indica que se agotó el tiempo de espera de la llamada y que ningún descriptor de archivos tuvo ningún evento que informar. Si se da `timeout`, especifica el período de tiempo en milisegundos que el sistema esperará por los eventos antes de regresar. Si se omite `timeout`, es -1 o es `None`, la llamada se bloqueará hasta que haya un evento para este objeto de encuesta.

Distinto en la versión 3.5: La función ahora se vuelve a intentar con un tiempo de espera (`timeout`) recalculado cuando se interrumpe por una señal, excepto si el controlador de señal genera una excepción (ver [PEP 475](#) para la justificación), en lugar de generar `InterruptedError`.

18.4.2 Objetos de sondeo de *Edge* y *Level Trigger* (*epoll*)

<https://linux.die.net/man/4/epoll>

eventmask

Constan- te	Significado
EPOLLIN	Disponible para lectura
EPOLLOUT	Disponible para escritura
EPOLLPRI	Urgente para lectura
EPOLLERR	La condición de error ocurrió en la asociación. fd
EPOLLHUP	Se colgó en la asociación. fd
EPOLLET	Establece el comportamiento en <i>Edge Trigger</i> , el valor predeterminado es <i>Level Trigger</i>
EPOLLONESHOT	Establece el comportamiento en <i>one-shot</i> . Después de que se retira un evento, el fd se deshabilita internamente
EPOLLEXCLUSIVE	Despierta solo un objeto epoll cuando el fd asociado tiene un evento. El valor predeterminado (si este flag no está configurado) es activar todos los objetos epoll que sondean en un fd.
EPOLLRDHUP	Socket de flujo de conexión cerrada por pares o apagado escribiendo la mitad de la conexión.
EPOLLRDNONBLOCK	Equivalente a EPOLLIN
EPOLLRDDEADLINE	Una banda de datos de prioridad se puede leer.
EPOLLWRNONBLOCK	Equivalente a EPOLLOUT
EPOLLWRBAND	Se pueden escribir datos de prioridad.
EPOLLMMSG	Ignorado.

Nuevo en la versión 3.6: EPOLLEXCLUSIVE fue agregado. Solo es compatible con Linux Kernel 4.5 o posterior.

`epoll.close()`

Cierra el descriptor del archivo de control del objeto epoll.

`epoll.closed`

True si el objeto epoll está cerrado.

`epoll.fileno()`

Retorna el número de descriptor de archivo del control fd.

`epoll.fromfd(fd)`

Crea un objeto epoll a partir de un descriptor de archivo dado.

`epoll.register(fd[, eventmask])`

Registra un descriptor fd con el objeto epoll.

`epoll.modify(fd, eventmask)`

Modifica un descriptor de archivo registrado.

`epoll.unregister(fd)`

Elimina un descriptor de archivo registrado del objeto epoll.

`epoll.poll(timeout=None, maxevents=-1)`

Espera los eventos. tiempo de espera (*timeout*) en segundos (float)

Distinto en la versión 3.5: La función ahora se vuelve a intentar con un tiempo de espera (*timeout*) recalculado cuando se interrumpe por una señal, excepto si el controlador de señal genera una excepción (ver [PEP 475](#) para la justificación), en lugar de generar *InterruptedError*.

18.4.3 Sondeo de objetos

La llamada al sistema: `poll()`, compatible con la mayoría de los sistemas Unix, proporciona una mejor escalabilidad para los servidores de red que dan servicio a muchos, muchos clientes al mismo tiempo. `poll()` escala mejor porque la llamada al sistema solo requiere enumerar los descriptors de archivo de interés, mientras que `select()` construye un mapa de bits, activa bits para los fds de interés, y luego todo el mapa de bits debe escanearse linealmente nuevamente. `select()` es $O(\text{descriptor de archivo más alto})$, mientras que `poll()` es $O(\text{número de descriptors de archivo})$.

`poll.register(fd[, eventmask])`

Registra un descriptor de archivo con el objeto de sondeo. Las futuras llamadas al método `poll()` comprobarán si el descriptor del archivo tiene algún evento I/O pendiente. `fd` puede ser un entero o un objeto con un método `fileno()` que retorna un entero. Los objetos de archivo implementan `Fileno()`, por lo que también pueden usarse como argumento.

`eventmask` es una máscara de bits opcional que describe el tipo de eventos que se desea verificar y puede ser una combinación de las constantes `POLLIN`, `POLLPRI`, y `POLLOUT`, descrito en la mesa de abajo. Si no se especifica, el valor predeterminado utilizado verificará los 3 tipos de eventos.

Constante	Significado
<code>POLLIN</code>	Hay datos para leer
<code>POLLPRI</code>	Hay datos urgentes para leer
<code>POLLOUT</code>	Lista para la salida: la escritura no bloqueará
<code>POLLERR</code>	Condición de error de algún tipo
<code>POLLHUP</code>	Colgado
<code>POLLRDHUP</code>	Socket de flujo de conexión cerrada por pares, o apagado escribiendo la mitad de la conexión
<code>POLLNVAL</code>	Solicitud no válida: descriptor no abierto

Al registrar un descriptor de archivo que ya está registrado no es un error y tiene el mismo efecto que registrar el descriptor exactamente una vez.

`poll.modify(fd, eventmask)`

Modifica un fd ya registrado. Esto tiene el mismo efecto que `register(fd, eventmask)`. Si se intenta modificar un descriptor de archivo que nunca se registró, se genera una excepción `OSError` con `errno` `ENOENT`.

`poll.unregister(fd)`

Elimina un descriptor de archivo que está siendo rastreado por un objeto de sondeo. Al igual que el método `register()`, `fd` puede ser un entero o un objeto con un método `fileno()` que retorna un entero.

Intenta eliminar un descriptor de archivo que nunca se registró provoca una excepción `KeyError`.

`poll.poll([timeout])`

Sondea el conjunto de descriptors de archivos registrados y retorna una lista posiblemente vacía que contiene `(fd, event)` y 2 tuplas para los descriptors que tienen eventos o errores que informar. `fd` es el descriptor de archivo, y `event` es una máscara de bits con bits establecidos para los eventos informados para ese descriptor — `POLLIN` para la entrada en espera, `POLLOUT` para indicar que el descriptor puede ser escrito, y así sucesivamente. Una lista vacía indica que se agotó el tiempo de espera de la llamada y que ningún descriptor de archivos tuvo ningún evento que informar. Si se da `timeout`, especifica el período de tiempo en milisegundos que el sistema esperará por los eventos antes de regresar. Si se omite `timeout`, es negativo, o `None`, la llamada se bloqueará hasta que haya un evento para este objeto de encuesta.

Distinto en la versión 3.5: La función ahora se vuelve a intentar con un tiempo de espera (`timeout`) recalculado cuando se interrumpe por una señal, excepto si el controlador de señal genera una excepción (ver [PEP 475](#) para la justificación), en lugar de generar `InterruptedError`.

18.4.4 Objetos Kqueue

`kqueue.close()`

Cierra el descriptor del archivo de control del objeto `kqueue`.

`kqueue.closed`

True si el objeto `kqueue` está cerrado.

`kqueue.fileno()`

Retorna el número de descriptor de archivo del control fd.

`kqueue.fromfd(fd)`

Crea un objeto `kqueue` a partir de un descriptor de archivo dado.

`kqueue.control(changelist, max_events[, timeout])` → eventlist

Interfaz de bajo nivel para kevent

- la lista de cambios (*changelist*) debe ser iterable de objetos kevent o None
- max_events debe ser 0 o un entero positivo
- tiempo de espera (*timeout*) en segundos (posible con *floats*); el valor predeterminado es None, para esperar para siempre

Distinto en la versión 3.5: La función ahora se vuelve a intentar con un tiempo de espera (*timeout*) recalculado cuando se interrumpe por una señal, excepto si el controlador de señal genera una excepción (ver [PEP 475](#) para la justificación), en lugar de generar `InterruptedError`.

18.4.5 Objetos Kevent

<https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2>

`kevent.ident`

Valor utilizado para identificar el evento. La interpretación depende del filtro, pero generalmente es el descriptor de archivo. En el constructor, `ident` puede ser un int o un objeto con un método `fileno()`. `kevent` almacena el entero internamente.

`kevent.filter`

Nombre del filtro del kernel.

Constante	Significado
<code>KQ_FILTER_READ</code>	Toma un descriptor y retorna cada vez que hay datos disponibles para leer
<code>KQ_FILTER_WRITE</code>	Toma un descriptor y retorna cada vez que hay datos disponibles para escribir
<code>KQ_FILTER_AIO</code>	Solicitudes de AIO
<code>KQ_FILTER_VNODE</code>	Retorna cuando ocurre uno o más de los eventos solicitados observados en <i>fflag</i>
<code>KQ_FILTER_PROC</code>	Vigila los eventos en un id de proceso
<code>KQ_FILTER_NETDEV</code>	Vigila los eventos en un dispositivo de red [no disponible en Mac OS X]
<code>KQ_FILTER_SIGNAL</code>	Retorna cada vez que la señal observada se entrega al proceso
<code>KQ_FILTER_TIMER</code>	Establece un temporizador arbitrario

`kevent.flags`

Acción de filtro.

Constante	Significado
KQ_EV_ADD	Agrega o modifica un evento
KQ_EV_DELETE	Elimina un evento de la cola
KQ_EV_ENABLE	Permitscontrol() para retornar el evento
KQ_EV_DISABLE	Disableevent
KQ_EV_ONESHOT	Elimina evento después de la primera aparición
KQ_EV_CLEAR	Restablece el estado después de recuperar un evento
KQ_EV_SYSFLAGS	evento interno
KQ_EV_FLAG1	evento interno
KQ_EV_EOF	Filtrar la condición específica de EOF
KQ_EV_ERROR	Ver valores de retorno

`kevent.flags`

Filtrar flags específicas.

KQ_FILTER_READ y KQ_FILTER_WRITE bandera de filtro:

Constante	Significado
KQ_NOTE_LOWAT	marca de agua baja de un <i>socket buffer</i>

KQ_FILTER_VNODE bandera de filtro:

Constante	Significado
KQ_NOTE_DELETE	<i>unlink()</i> fue llamado
KQ_NOTE_WRITE	una escritura ha ocurrido
KQ_NOTE_EXTEND	el archivo fue extendido
KQ_NOTE_ATTRIB	un atributo fue cambiado
KQ_NOTE_LINK	el recuento de enlaces ha cambiado
KQ_NOTE_RENAME	el archivo fue renombrado
KQ_NOTE_REVOKE	se revocó el acceso al archivo

KQ_FILTER_PROC bandera de filtro:

Constante	Significado
KQ_NOTE_EXIT	el proceso ha terminado (<i>exited</i>)
KQ_NOTE_FORK	el proceso ha llamado a <i>fork()</i>
KQ_NOTE_EXEC	el proceso ha ejecutado un nuevo proceso
KQ_NOTE_PCTRLMASK	flag de filtro interno
KQ_NOTE_PDATAMASK	flag de filtro interno
KQ_NOTE_TRACK	sigue un proceso a través de <i>fork()</i>
KQ_NOTE_CHILD	retornado en el proceso hijo para <i>NOTE_TRACK</i>
KQ_NOTE_TRACKERR	incapaz de adjuntar a un proceso hijo

KQ_FILTER_NETDEV banderas de filtro (no disponible en Mac OS X):

Constante	Significado
KQ_NOTE_LINKUP	el enlace está funcionando
KQ_NOTE_LINKDOWN	el enlace está caído
KQ_NOTE_LINKINV	el estado del enlace es invalido

`kevent.data`

Filtrar datos específicos.

`kevent.udata`

Valor definido por el usuario.

18.5 selectors — Multiplexación de E/S de alto nivel

Nuevo en la versión 3.4.

Código fuente: [Lib/selectors.py](#)

18.5.1 Introducción

Este módulo permite multiplexación de E/S eficiente y de alto nivel, basada en los primitivos del módulo *select*. Se recomienda a los usuarios utilizar este módulo en su lugar, a menos que deseen un control preciso sobre los primitivos a nivel de sistema operativo utilizados.

Define una clase base abstracta *BaseSelector*, junto con varias implementaciones concretas (*KqueueSelector*, *EpollSelector*...), que se pueden utilizar para esperar la notificación de disponibilidad para E/S en varios objetos de archivo. De aquí en adelante, «objeto de archivo» hace referencia a cualquier objeto con un método *fileno()* o un descriptor de archivo sin procesar. Véase *file object*.

DefaultSelector es un alias de la implementación más eficiente disponible en la plataforma actual: esta debería ser la opción predeterminada para la mayoría de los usuarios.

Nota: El tipo de objetos de archivo admitidos depende de la plataforma: en Windows, se admiten sockets, pero no *pipes*, mientras que en Unix, ambos son compatibles (también se pueden admitir algunos otros tipos, como FIFOs o dispositivos de archivo especiales).

Ver también:

select Módulo de multiplexación de E/S de bajo nivel.

18.5.2 Clases

Jerarquía de clases:

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

De aquí en adelante, *events* es una máscara bit a bit que indica qué eventos de E/S se deben esperar en un objeto de archivo determinado. Puede ser cualquier combinación de las siguientes constantes de módulo:

Constante	Significado
EVENT_READ	Disponible para lectura
EVENT_WRITE	Disponible para escritura

class selectors.SelectorKey

La clase *SelectorKey* es una *namedtuple* que se utiliza para asociar un objeto de archivo a su descriptor de archivo subyacente, máscara de evento seleccionada y datos adjuntos. Es retornada por varios métodos *BaseSelector*.

fileobj

Objeto de archivo registrado.

fd

Descriptor de archivo subyacente.

events

Eventos que se deben esperar en este objeto de archivo.

data

Datos opacos opcionales asociados a este objeto de archivo: por ejemplo, podría usarse para almacenar un ID de sesión por cliente.

class selectors.BaseSelector

Un *BaseSelector* se usa para esperar a que el evento de E/S esté listo en varios objetos de archivo. Admite el registro y la cancelación del registro de secuencias de archivos y un método para esperar eventos de E/S en esas secuencias, con un tiempo de espera opcional. Es una clase base abstracta, por lo que no se puede crear una instancia. Use *DefaultSelector* en su lugar, o un *SelectSelector*, *KqueueSelector*, etc. si deseas usar específicamente una implementación y su plataforma la admite. *BaseSelector* y sus implementaciones concretas soportan el protocolo *context manager*.

abstractmethod register (*fileobj*, *events*, *data=None*)

Registra un objeto de archivo para su selección y lo monitoriza en busca de eventos de E/S.

fileobj es el objeto de archivo a monitorear. Puede ser un descriptor de archivo de tipo entero o un objeto con un método `fileno()`. *events* es una máscara bit a bit de eventos para monitorear. *data* es un objeto opaco.

Esto retorna una nueva instancia *SelectorKey*, o lanza un *ValueError* en caso de una máscara de evento o un descriptor de archivo no válidos, o *KeyError* si el objeto de archivo ya está registrado.

abstractmethod unregister (*fileobj*)

Anula el registro de un objeto de archivo de la selección y lo elimina del monitoreo. Se anulará el registro de un objeto de archivo antes de cerrarlo.

fileobj debe ser un objeto de archivo previamente registrado.

Esto retorna la instancia *SelectorKey* asociada, o lanza un *KeyError* si *fileobj* no está registrado. Se lanzará un *ValueError* si *fileobj* no es válido (por ejemplo, no tiene el método `fileno()` o su método `fileno()` tiene un valor de retorno no válido).

modify (*fileobj*, *events*, *data=None*)

Cambia los eventos monitorizados de un objeto de archivo registrado o los datos adjuntos.

Esto es equivalente a `BaseSelector.unregister(fileobj)()` seguido de `BaseSelector.register(fileobj, events, data)()`, excepto que se puede implementar de manera más eficiente.

Esto retorna una nueva instancia de *SelectorKey*, o lanza un *ValueError* en caso de una máscara de evento o un descriptor de archivo no válidos, o *KeyError* si el objeto de archivo no está registrado.

abstractmethod select (*timeout=None*)

Espera hasta que algunos objetos de archivo registrados estén listos o el tiempo de espera expire.

Si *timeout* > 0, esto especifica el tiempo máximo de espera, en segundos. Si *timeout* ≤ 0, la llamada no se bloqueará y reportará los objetos de archivo actualmente listos. Si *timeout* es *None*, la llamada se bloqueará hasta que un objeto de archivo monitorizado esté listo.

Retorna una lista de tuplas (*key*, *events*), una por cada objeto de archivo listo.

key es la instancia *SelectorKey* correspondiente a un objeto de archivo listo. *events* es una máscara de bits de eventos listos en este objeto de archivo.

Nota: Este método puede regresar antes de que cualquier objeto de archivo esté listo o haya transcurrido el tiempo de espera si el proceso actual recibe una señal: en este caso, se retornará una lista vacía.

Distinto en la versión 3.5: El selector ahora se reintenta con un tiempo de espera recalculado cuando es interrumpido por una señal si el gestor de señales no lanzó una excepción (ver [PEP 475](#) para la justificación), en lugar de retornar una lista vacía de eventos antes del tiempo de espera.

close()

Cierra el selector.

Se debe llamar para asegurarse de que se libera cualquier recurso subyacente. El selector no se utilizará una vez cerrado.

get_key() (*fileobj*)

Retorna la clave asociada con un objeto de archivo registrado.

Esto retorna la instancia `SelectorKey` asociada a este objeto de archivo, o lanza un `KeyError` si el objeto de archivo no está registrado.

abstractmethod get_map()

Retorna un mapeo asociando objetos de archivo a las llaves del selector.

Retorna una instancia de `Mapping` mapeando objetos de archivo registrados a su instancia `SelectorKey` asociada

class selectors.DefaultSelector

La clase de selector predeterminada, utiliza la implementación más eficiente disponible en la plataforma actual. Esta debería ser la opción predeterminada para la mayoría de los usuarios.

class selectors.SelectSelector

Selector basado en `select.select()`.

class selectors.PollSelector

Selector basado en `select.poll()`.

class selectors.EpollSelector

Selector basado en `select.epoll()`.

fileno()

Esto retorna el descriptor de archivo utilizado por el objeto `select.epoll()` subyacente.

class selectors.DevpollSelector

Selector basado en `select.devpoll()`.

fileno()

Esto retorna el descriptor de archivo utilizado por el objeto `select.devpoll()` subyacente.

Nuevo en la versión 3.5.

class selectors.KqueueSelector

Selector basado en `select.kqueue()`.

fileno()

Esto retorna el descriptor de archivo utilizado por el objeto `select.kqueue()` subyacente.

18.5.3 Ejemplos

Aquí hay una implementación simple de un servidor de eco:

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept() # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000) # Should be ready
```

(continué en la próxima página)

(proviene de la página anterior)

```

if data:
    print('echoing', repr(data), 'to', conn)
    conn.send(data) # Hope it won't block
else:
    print('closing', conn)
    sel.unregister(conn)
    conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)

```

18.6 `asyncore` — controlador de socket asincrónico

Código fuente: [Lib/asyncore.py](#)

Obsoleto desde la versión 3.6: Por favor, utilice `asyncio` en su lugar.

Nota: Este módulo solo existe para compatibilidad con versiones anteriores. Para el nuevo código recomendamos usar `asyncio`.

Este módulo proporciona la infraestructura básica para escribir servicio de socket asincrónicos, clientes y servidores.

Sólo hay dos maneras de que un programa en un solo procesador haga «más de una cosa a la vez». La programación multiproceso es la forma más sencilla y popular de hacerlo, pero hay otra técnica muy diferente, que le permite tener casi todas las ventajas de multiproceso, sin usar realmente varios subprocesos. Es realmente sólo práctico si su programa está en gran parte limitado por el I/O. Si el programa está limitado por el procesador, los subprocesos programados preventivos son probablemente lo que realmente necesita. Sin embargo, los servidores de red rara vez están limitado al procesador.

Si su sistema operativo es compatible con la llamada del sistema `select()` en su biblioteca de I/O (y casi todos lo son), puede usarla para hacer malabares con varios canales de comunicación a la vez; haciendo otro trabajo mientras su I/O está teniendo lugar en el «fondo». Aunque esta estrategia puede parecer extraña y compleja, especialmente al principio, es en muchos sentidos más fácil de entender y controlar que la programación multiproceso. El módulo `asyncore` resuelve muchos de los problemas difíciles para usted, haciendo que la tarea de construir sofisticados servidores de red de alto rendimiento y clientes sea fácil. Para aplicaciones y protocolos «conversacionales», el módulo complementario `asynchat` es invaluable.

La idea básica detrás de ambos módulos es crear uno o más *canales* de red, instancias de clase `asyncore.dispatcher` y `asynchat.async_chat`. La creación de los canales los agrega a un mapa global, utilizado por la función `loop()` si no lo proporciona con su propio `map`.

Una vez creados los canales iniciales, llamar a la función `loop()` activa el servicio de canal, que continúa hasta que se cierra el último canal (incluido el que se ha agregado al mapa durante el servicio asincrónico).

`asyncore.loop([timeout[, use_poll[, map[, count]]]])`

Ingresa un bucle de sondeo que termina después de que se hayan cerrado los pases de conteo o todos los canales abiertos. Todos los argumentos son opcionales. El parámetro `count` tiene como valor predeterminado `None`, lo que da como resultado que el bucle termine solo cuando se hayan cerrado todos los canales. El argumento

`timeout` establece el parámetro de tiempo de espera para la llamada adecuada a `select()` o `poll()`, medida en segundos; el valor predeterminado es 30 segundos. El parámetro `use_poll`, si es `true`, indica que `poll()` debe utilizarse en lugar de `select()` (el valor predeterminado es `False`).

El parámetro `map` es un diccionario cuyos elementos son los canales a observar. A medida que se cierran los canales, se eliminan del mapa. Si se omite `map`, se utiliza un mapa global. Los canales (instancias de `asyncore.dispatcher`, `asyncchat.async_chat` y subclases de los mismos) se pueden mezclar libremente en el mapa.

class `asyncore.dispatcher`

La clase `dispatcher` es un contenedor fino alrededor de un objeto de socket de bajo nivel. Para hacerlo más útil, tiene algunos métodos para el control de eventos que se llaman desde el bucle asincrónico. De lo contrario, se puede tratar como un objeto de socket normal sin bloqueo.

La activación de eventos de bajo nivel en determinados momentos o en determinados estados de conexión indica al bucle asincrónico que se han producido ciertos eventos de nivel superior. Por ejemplo, si hemos pedido un socket para conectarse a otro host, sabemos que la conexión se ha realizado cuando el socket se vuelve *grabable* por primera vez (en este punto sabe que puede escribir a él con la expectativa de éxito). Los eventos de nivel superior implícitos son:

Evento	Descripción
<code>handle_connect()</code>	Implícito en el primer proceso de lectura o escritura
<code>handle_close()</code>	Implícito en un evento de lectura sin datos disponibles
<code>handle_accepted()</code>	Implícito en un evento de lectura en un socket de escucha

Durante el procesamiento asincrónico, se utilizan los métodos `readable()` y `writable()` de cada canal asignado para determinar si el socket del canal deberían ser agregados a la lista de canales seleccionados o encuestados para eventos de lectura y escritura.

Por lo tanto, el conjunto de eventos de canal es mayor que los eventos básicos del socket. El conjunto completo de métodos que se pueden invalidar en la subclase es el siguiente:

handle_read()

Se llama cuando el bucle asincrónico detecta que una llamada `read()` en el socket del canal tendrá éxito.

handle_write()

Se llama cuando el bucle asincrónico detecta que se puede escribir un socket grabable. A menudo, este método implementará el almacenamiento en búfer necesario para el rendimiento. Por ejemplo:

```
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

handle_expt()

Se llama cuando hay datos fuera de banda (OOB) para una conexión de socket. Esto casi nunca sucederá, ya que OOB es tenuemente compatible y rara vez se utiliza.

handle_connect()

Se llama cuando el socket del abridor activo realmente hace una conexión. Puede enviar un banner de «bienvenido» o iniciar una negociación de protocolo con el punto de conexión remoto, por ejemplo.

handle_close()

Se llama cuando el socket está cerrado.

handle_error()

Se llama cuando se genera una excepción y no se controla de otro modo. La versión predeterminada imprime un *traceback* condensado.

handle_accept()

Se llama en los canales de escucha (abridores pasivos) cuando se puede establecer una conexión con un nuevo punto de conexión remoto que ha emitido una llamada `connect()` para el punto de conexión local. En desuso en la versión 3.2; use `handle_accepted()` en su lugar.

Obsoleto desde la versión 3.2.

handle_accepted (*sock*, *addr*)

Se llama en los canales de escucha (abridores pasivos) cuando se ha establecido una conexión con un nuevo punto de conexión remoto que ha emitido una llamada `connect()` para el punto de conexión local. *sock* es un objeto de socket *new* utilizable para enviar y recibir datos en la conexión, y *addr* es la dirección enlazada al socket en el otro extremo de la conexión.

Nuevo en la versión 3.2.

readable ()

Se llama en cada momento alrededor del bucle asíncronico para determinar si se debe agregar el socket de un canal a la lista en la que se pueden producir eventos de lectura. El método predeterminado simplemente retorna `True`, lo que indica que, de forma predeterminada, todos los canales estarán interesados en eventos de lectura.

writable ()

Se llama cada vez alrededor del bucle asíncronico para determinar si se debe agregar el socket de un canal a la lista en la que se pueden producir eventos de escritura. El método predeterminado simplemente retorna `True`, lo que indica que, de forma predeterminada, todos los canales estarán interesados en eventos de escritura.

Además, cada canal delega o extiende muchos de los métodos de socket. La mayoría de estos son casi idénticos a sus socios de socket.

create_socket (*family*=`socket.AF_INET`, *type*=`socket.SOCK_STREAM`)

Esto es idéntico a la creación de un socket normal y usará las mismas opciones para la creación. Consulte la documentación `socket` para obtener información sobre la creación de sockets.

Distinto en la versión 3.3: Los argumentos *family* y *type* se pueden omitir.

connect (*address*)

Al igual que con el objeto de socket normal, *address* es una tupla con el primer elemento al que se va a conectar el host y el segundo el número de puerto.

send (*data*)

Envía *data* al punto final remoto del socket.

recv (*buffer_size*)

Lee como máximo los bytes *buffer_size* desde el punto final remoto del socket. Un objeto bytes vacío implica que el canal se ha cerrado desde el otro extremo.

Tenga en cuenta que `recv()` puede elevar `BlockingIOError`, aunque `select.select()` o `select.poll()` ha informado del socket listo para la lectura.

listen (*backlog*)

Escucha las conexiones realizadas al socket. El argumento *backlog* especifica el número máximo de conexiones en cola y debe ser al menos 1; el valor máximo depende del sistema (normalmente 5).

bind (*address*)

Enlaza el socket a *address*. El socket no debe estar enlazado ya. (El formato de *address* depende de la familia de direcciones — consulte la documentación `socket` para obtener más información.) Para marcar el socket como *reutilizable* (estableciendo la opción `SO_REUSEADDR`), llame al método `set_reuse_addr()` del objeto *dispatcher*.

accept ()

Acepta una conexión. El socket debe estar enlazado a una dirección y escuchar las conexiones. El valor retornado puede ser `None` o un par (*conn*, *address*) donde *conn* es un objeto de socket *new* utilizable para enviar y recibir datos en la conexión, y *address* es la dirección enlazada al socket en el otro extremo de la conexión. Cuando se retorna `None` significa que la conexión no se llevó a cabo, en cuyo caso el servidor debe ignorar este evento y seguir escuchando otras conexiones entrantes.

close ()

Cierra el socket. Se producirá un error en todas las operaciones futuras en el objeto de socket. El punto final remoto no recibirá más datos (después de vaciar los datos en cola). Los sockets se cierran automáticamente cuando se recogen como elementos no utilizados.

class `asyncore.dispatcher_with_send`

Una subclase `dispatcher` que agrega capacidad de salida almacenada en búfer simple, útil para clientes simples. Para un uso más sofisticado, utilice `asyncchat.async_chat`.

class `asyncore.file_dispatcher`

`file_dispatcher` toma un descriptor de archivo o *file object* junto con un argumento de mapa opcional y lo ajusta para su uso con las funciones `poll()` o `loop()`. Si se proporciona un objeto de archivo o cualquier cosa con un método `fileno()`, ese método se llamará y se pasará al constructor `file_wrapper`.

Disponibilidad: Unix.

class `asyncore.file_wrapper`

`file_wrapper` toma un descriptor de archivo entero y llama a `os.dup()` para duplicar el identificador de modo que el identificador original se pueda cerrar independientemente del `file_wrapper`. Esta clase implementa métodos suficientes emulando un socket para su uso por la clase `file_dispatcher`.

Disponibilidad: Unix.

18.6.1 Ejemplo asyncore de cliente HTTP básico

Aquí hay una llamada básica al cliente HTTP que usa la clase `dispatcher` para implementar su controlador de socket:

```
import asyncore

class HTTPClient(asyncore.dispatcher):

    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.connect((host, 80))
        self.buffer = bytes('GET %s HTTP/1.0\r\nHost: %s\r\n\r\n' %
                             (path, host), 'ascii')

    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()
```

18.6.2 Ejemplo asyncore de servidor de eco básico

Aquí hay un servidor de eco básico que utiliza la clase `dispatcher` para aceptar conexiones y distribuye las conexiones entrantes a un controlador:

```
import asyncore

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()
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)

    def handle_accepted(self, sock, addr):
        print('Incoming connection from %s' % repr(addr))
        handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```

18.7 `asyncchat` — Asynchronous socket command/response handler

Source code: [Lib/asyncchat.py](#)

Obsoleto desde la versión 3.6: Please use `asyncio` instead.

Nota: This module exists for backwards compatibility only. For new code we recommend using `asyncio`.

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. `asyncchat` 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 `asyncchat.async_chat`, can freely be mixed in the channel map. Typically an `asyncore.dispatcher` server channel generates new `asyncchat.async_chat` channel objects as it receives incoming connection requests.

`class` `asyncchat.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 FIFO queue 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 bytes object. At this point the `async_chat` object removes the producer from the queue and starts using the next producer, if any. When the producer queue 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 queue. When this producer is popped off the queue 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 queue.

`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 queue 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 queue 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
<i>string</i>	Will call <code>found_terminator()</code> when the string is found in the input stream
<i>integer</i>	Will call <code>found_terminator()</code> when the indicated number of characters have been received
<code>None</code>	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.

18.7.1 `asyncchat` 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.

```
import asyncchat

class http_request_handler(asyncchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asyncchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = b""
        self.set_terminator(b"\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(b"".join(self.ibuffer))
            self.ibuffer = []
            if self.op.upper() == b"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, b"".join(self.ibuffer))
            self.handling = True
            self.ibuffer = []
            self.handle_request()
```

18.8 `signal` — Set handlers for asynchronous events

This module provides mechanisms to use signal handlers in Python.

18.8.1 General rules

The `signal.signal()` function allows defining custom handlers to be executed when a signal is received. A small number of default handlers are installed: `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 if the parent process has not changed it.

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.

Execution of Python signal handlers

A Python signal handler does not get executed inside the low-level (C) signal handler. Instead, the low-level signal handler sets a flag which tells the *virtual machine* to execute the corresponding Python signal handler at a later point (for example at the next *bytecode* instruction). This has consequences:

- It makes little sense to catch synchronous errors like `SIGFPE` or `SIGSEGV` that are caused by an invalid operation in C code. Python will return from the signal handler to the C code, which is likely to raise the same signal again, causing Python to apparently hang. From Python 3.3 onwards, you can use the `faulthandler` module to report on synchronous errors.
- A long-running calculation implemented purely in C (such as regular expression matching on a large body of text) may run uninterrupted for an arbitrary amount of time, regardless of any signals received. The Python signal handlers will be called when the calculation finishes.

Signals and threads

Python signal handlers are always executed in the main Python thread, even if the signal was received in another thread. This means that signals can't be used as a means of inter-thread communication. You can use the synchronization primitives from the `threading` module instead.

Besides, only the main thread is allowed to set a new signal handler.

18.8.2 Module contents

Distinto en la versión 3.5: `signal` (`SIG*`), `handler` (`SIG_DFL`, `SIG_IGN`) and `sigmask` (`SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`) related constants listed below were turned into `enums`. `getsignal()`, `pthread_sigmask()`, `sigpending()` and `sigwait()` functions return human-readable `enums`.

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.

`signal.SIGABRT`

Abort signal from `abort(3)`.

signal.SIGALRM

Timer signal from *alarm(2)*.

Availability: Unix.

signal.SIGBREAK

Interrupt from keyboard (CTRL + BREAK).

Availability: Windows.

signal.SIGBUS

Bus error (bad memory access).

Availability: Unix.

signal.SIGCHLD

Child process stopped or terminated.

Availability: Unix.

signal.SIGCLD

Alias to *SIGCHLD*.

signal.SIGCONT

Continue the process if it is currently stopped

Availability: Unix.

signal.SIGFPE

Floating-point exception. For example, division by zero.

Ver también:

ZeroDivisionError is raised when the second argument of a division or modulo operation is zero.

signal.SIGHUP

Hangup detected on controlling terminal or death of controlling process.

Availability: Unix.

signal.SIGILL

Illegal instruction.

signal.SIGINT

Interrupt from keyboard (CTRL + C).

Default action is to raise *KeyboardInterrupt*.

signal.SIGKILL

Kill signal.

It cannot be caught, blocked, or ignored.

Availability: Unix.

signal.SIGPIPE

Broken pipe: write to pipe with no readers.

Default action is to ignore the signal.

Availability: Unix.

signal.SIGSEGV

Segmentation fault: invalid memory reference.

signal.SIGTERM

Termination signal.

signal.SIGUSR1

User-defined signal 1.

Availability: Unix.

`signal.SIGUSR2`

User-defined signal 2.

Availability: Unix.

`signal.SIGWINCH`

Window resize signal.

Availability: Unix.

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.

Nuevo en la versión 3.2.

`signal.CTRL_BREAK_EVENT`

The signal corresponding to the Ctrl+Break keystroke event. This signal can only be used with `os.kill()`.

Availability: Windows.

Nuevo en la versión 3.2.

`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.

`signal.SIG_BLOCK`

A possible value for the *how* parameter to `pthread_sigmask()` indicating that signals are to be blocked.

Nuevo en la versión 3.3.

`signal.SIG_UNBLOCK`

A possible value for the *how* parameter to `pthread_sigmask()` indicating that signals are to be unblocked.

Nuevo en la versión 3.3.

`signal.SIG_SETMASK`

A possible value for the *how* parameter to `pthread_sigmask()` indicating that the signal mask is to be replaced.

Nuevo en la versión 3.3.

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 `OSError`.

Nuevo en la versión 3.3: This error used to be a subtype of `IOError`, which is now an alias of `OSError`.

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.

Availability: Unix. See the man page `alarm(2)` for further information.

`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.strsignal(signalnum)`

Return the system description of the signal *signalnum*, such as «Interrupt», «Segmentation fault», etc. Returns `None` if the signal is not recognized.

Nuevo en la versión 3.8.

`signal.valid_signals()`

Return the set of valid signal numbers on this platform. This can be less than `range(1, NSIG)` if some signals are reserved by the system for internal use.

Nuevo en la versión 3.8.

`signal.pause()`

Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing.

Availability: Unix. See the man page `signal(2)` for further information.

See also `sigwait()`, `sigwaitinfo()`, `sigtimedwait()` and `sigpending()`.

`signal.raise_signal(signum)`

Sends a signal to the calling process. Returns nothing.

Nuevo en la versión 3.8.

`signal.pthread_kill(thread_id, signalnum)`

Send the signal *signalnum* to the thread *thread_id*, another thread in the same process as the caller. The target thread can be executing any code (Python or not). However, if the target thread is executing the Python interpreter, the Python signal handlers will be *executed by the main thread*. Therefore, the only point of sending a signal to a particular Python thread would be to force a running system call to fail with `InterruptedError`.

Use `threading.get_ident()` or the `ident` attribute of `threading.Thread` objects to get a suitable value for *thread_id*.

If *signalnum* is 0, then no signal is sent, but error checking is still performed; this can be used to check if the target thread is still running.

Raises an *auditing event* `signal.pthread_kill` with arguments `thread_id`, `signalnum`.

Availability: Unix. See the man page `pthread_kill(3)` for further information.

See also `os.kill()`.

Nuevo en la versión 3.3.

`signal.pthread_sigmask(how, mask)`

Fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller. Return the old signal mask as a set of signals.

The behavior of the call is dependent on the value of *how*, as follows.

- `SIG_BLOCK`: The set of blocked signals is the union of the current set and the *mask* argument.
- `SIG_UNBLOCK`: The signals in *mask* are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.

- `SIG_SETMASK`: The set of blocked signals is set to the *mask* argument.

mask is a set of signal numbers (e.g. `{signal.SIGINT, signal.SIGTERM}`). Use `valid_signals()` for a full mask including all signals.

For example, `signal.pthread_sigmask(signal.SIG_BLOCK, [])` reads the signal mask of the calling thread.

`SIGKILL` and `SIGSTOP` cannot be blocked.

Availability: Unix. See the man page `sigprocmask(3)` and `pthread_sigmask(3)` for further information.

See also `pause()`, `sigpending()` and `sigwait()`.

Nuevo en la versión 3.3.

`signal.setitimer(which, seconds, interval=0.0)`

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 (if *interval* is non-zero). 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.

`signal.getitimer(which)`

Returns current value of a given interval timer specified by *which*.

Availability: Unix.

`signal.set_wakeup_fd(fd, *, warn_on_full_buffer=True)`

Set the wakeup file descriptor to *fd*. When a signal is received, the signal number is written as a single byte into 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.

There are two common ways to use this function. In both approaches, you use the fd to wake up when a signal arrives, but then they differ in how they determine *which* signal or signals have arrived.

In the first approach, we read the data out of the fd's buffer, and the byte values give you the signal numbers. This is simple, but in rare cases it can run into a problem: generally the fd will have a limited amount of buffer space, and if too many signals arrive too quickly, then the buffer may become full, and some signals may be lost. If you use this approach, then you should set `warn_on_full_buffer=True`, which will at least cause a warning to be printed to stderr when signals are lost.

In the second approach, we use the wakeup fd *only* for wakeups, and ignore the actual byte values. In this case, all we care about is whether the fd's buffer is empty or non-empty; a full buffer doesn't indicate a problem at all. If you use this approach, then you should set `warn_on_full_buffer=False`, so that your users are not confused by spurious warning messages.

Distinto en la versión 3.5: On Windows, the function now also supports socket handles.

Distinto en la versión 3.7: Added `warn_on_full_buffer` parameter.

`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.

`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)* for further information.)

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*, *SIGTERM*, or *SIGBREAK*. A *ValueError* will be raised in any other case. Note that not all systems define the same set of signal names; an *AttributeError* will be raised if a signal name is not defined as *SIG** module level constant.

`signal.sigpending ()`

Examine the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). Return the set of the pending signals.

Availability: Unix. See the man page *sigpending (2)* for further information.

See also *pause()*, *pthread_sigmask()* and *sigwait()*.

Nuevo en la versión 3.3.

`signal.sigwait (sigset)`

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal (removes it from the pending list of signals), and returns the signal number.

Availability: Unix. See the man page *sigwait (3)* for further information.

See also *pause()*, *pthread_sigmask()*, *sigpending()*, *sigwaitinfo()* and *sigtimedwait()*.

Nuevo en la versión 3.3.

`signal.sigwaitinfo (sigset)`

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal and removes it from the pending list of signals. If one of the signals in *sigset* is already pending for the calling thread, the function will return immediately with information about that signal. The signal handler is not called for the delivered signal. The function raises an *InterruptedError* if it is interrupted by a signal that is not in *sigset*.

The return value is an object representing the data contained in the *siginfo_t* structure, namely: *si_signo*, *si_code*, *si_errno*, *si_pid*, *si_uid*, *si_status*, *si_band*.

Availability: Unix. See the man page *sigwaitinfo (2)* for further information.

See also *pause()*, *sigwait()* and *sigtimedwait()*.

Nuevo en la versión 3.3.

Distinto en la versión 3.5: The function is now retried if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see [PEP 475](#) for the rationale).

`signal.sigtimedwait (sigset, timeout)`

Like `sigwaitinfo()`, but takes an additional *timeout* argument specifying a timeout. If *timeout* is specified as 0, a poll is performed. Returns *None* if a timeout occurs.

Availability: Unix. See the man page `sigtimedwait(2)` for further information.

See also `pause()`, `sigwait()` and `sigwaitinfo()`.

Nuevo en la versión 3.3.

Distinto en la versión 3.5: The function is now retried with the recomputed *timeout* if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see [PEP 475](#) for the rationale).

18.8.3 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 OSError("Couldn't open device!")

# 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
```

18.8.4 Note on SIGPIPE

Piping output of your program to tools like `head(1)` will cause a `SIGPIPE` signal to be sent to your process when the receiver of its standard output closes early. This results in an exception like `BrokenPipeError: [Errno 32] Broken pipe`. To handle this case, wrap your entry point to catch this exception as follows:

```
import os
import sys

def main():
    try:
        # simulate large output (your code replaces this loop)
        for x in range(10000):
            print("y")
        # flush output here to force SIGPIPE to be triggered
        # while inside this try block.
        sys.stdout.flush()
    except BrokenPipeError:
        # Python flushes standard streams on exit; redirect remaining output
        # to devnull to avoid another BrokenPipeError at shutdown
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1) # Python exits with error code 1 on EPIPE

if __name__ == '__main__':
    main()
```

Do not set `SIGPIPE`'s disposition to `SIG_DFL` in order to avoid `BrokenPipeError`. Doing that would cause your program to exit unexpectedly also whenever any socket connection is interrupted while your program is still writing to it.

18.9 mmap — Soporte de archivos mapeados en memoria

Memory-mapped file objects behave like both `bytearray` and like `file objects`. You can use `mmap` objects in most places where `bytearray` are expected; for example, you can use the `re` module to search through a memory-mapped file. You can also change a single byte by doing `obj[index] = 97`, or change a subsequence by assigning to a slice: `obj[i1:i2] = b'...'`. You can also read and write data starting at the current file position, and `seek()` through the file to different positions.

Un archivo mapeado en memoria se crea con el constructor `mmap`, que es diferente en Unix y en Windows. En cualquier caso, debes proporcionar un descriptor de archivo para un archivo abierto para la actualización. Si deseas mapear un objeto archivo de Python existente, use su método `fileno()` para obtener el valor correcto para el parámetro `fileno`. De otra manera, puedes abrir el archivo usando la función `os.open()`, que retorna un descriptor de archivo directamente (el archivo aún necesita ser cerrado cuando hayas terminado).

Nota: Si quieres crear un mapeado en memoria para un archivo con permisos de escritura y en el búfer, debes ejecutar la función `flush()`. Es necesario para asegurar que las modificaciones locales a los búfer estén realmente disponible para el mapeado.

Para las versiones del constructor de tanto Unix como de Windows, `access` puede ser especificado como un parámetro nombrado opcional. `access` acepta uno de cuatro valores: `ACCESS_READ`, `ACCESS_WRITE`, o `ACCESS_DEFAULT` para especificar una memoria de sólo lectura, *write-through*, o *copy-on-write* respectivamente, o `ACCESS_DEFAULT` para deferir a `prot`. El parámetro `access` se puede usar tanto en Unix como en Windows. Si `access` no es especificado, el `mmap` de Windows retorna un mapeado *write-through*. Los valores de la memoria inicial para los tres tipos de acceso son tomados del archivo especificado. La asignación a una mapa de memoria `ACCESS_READ` lanza una excepción `TypeError`. La asignación a un mapa de memoria `ACCESS_WRITE` afecta tanto a la memoria como al archivo subyacente. La asignación a un mapa de memoria `ACCESS_COPY` afecta a la memoria pero no actualiza el archivo subyacente.

Distinto en la versión 3.7: Se añadió la constante `ACCESS_DEFAULT`.

Para mapear memoria anónima, se debe pasar `-1` como el `fileno` junto con la longitud.

class `mmap.mmap` (`fileno`, `length`, `tagname=None`, `access=ACCESS_DEFAULT` [, `offset`])

(En la versión de Windows) Mapea `length` bytes desde el archivo especificado por el gestor de archivo `fileno`, y crea un objeto `mmap`. Si `length` es más largo que el tamaño actual del archivo, el archivo es extendido para contener `length` bytes. Si `length` es 0, la longitud máxima del map es la tamaño actual del archivo, salvo que si el archivo está vacío Windows lanza una excepción (no puedes crear un mapeado vacío en Windows)

`tagname`, si está especifico y no es `None`, es una cadena que proporciona el nombre de la etiqueta para el mapeado. Windows te permite tener varios mapeados diferentes del mismo archivo. Si especificas el nombre de una etiqueta existente, la etiqueta se abre, de otro modo una crea una nueva etiqueta. Si este parámetro se omite o es `None`, el mapeado es creado sin un nombre. Evitar el uso del parámetro etiqueta te ayudará a mantener tu código portable entre Unix y Windows.

`offset` puede ser especificado como un `offset` entero no negativo. las referencias de `mmap` serán relativas al `offset` desde el comienzo del archivo. `offset` es por defecto 0. `offset` debe ser un múltiplo de `ALLOCATIONGRANULARITY`.

Lanza un *evento de inspección* `mmap.__new__` con los argumentos `fileno`, `length`, `access`, `offset`.

class `mmap.mmap` (`fileno`, `length`, `flags=MAP_SHARED`, `prot=PROT_WRITE|PROT_READ`, `access=ACCESS_DEFAULT` [, `offset`])

(En la versión de Unix) Mapea `length` bytes desde el archivo especificado por el descriptor de archivo `fileno`,

y retorna un objeto *mmap*. Si *length* es 0, la longitud máxima del map será el tamaño actual del archivo cuando *mmap* sea llamado.

flags especifica la naturaleza del mapeado. `MAP_PRIVATE` crea un mapeado *copy-on-write* privado, por lo que los cambios al contenido del objeto *mmap* serán privados para este proceso, y `MAP_SHARED` crea un mapeado que es compartido con todos los demás procesos que mapean las mismas áreas del archivo. El valor por defecto es `MAP_SHARED`.

prot, si se especifica, proporciona la protección de memoria deseado; los dos valores más útiles son `PROT_READ` y `PROT_WRITE`, para especificar que las páginas puedan ser escritas o leídas. *prot* es por defecto `PROT_READ | PROT_WRITE`.

access puede ser especificado en lugar de *flags* y *prot* como un parámetro nombrado opcional. Es un error especificar tanto *flags*, *prot* como *access*. Véase la descripción de *access* arriba por información de cómo usar este parámetro.

offset puede ser especificado como un *offset* entero no negativo. Las referencias serán relativas al *offset* desde el comienzo del archivo. *offset* por defecto es 0. *offset* debe ser un múltiplo de `ALLOCATIONGRANULARITY` que es igual a `PAGESIZE` en los sistemas Unix.

Para asegurar la validez del mapeado en memoria creado el archivo especificado por el descriptor *fileno* es internamente y automáticamente sincronizado con la memoria de respaldo en Mac OS X y OpenVMS.

Este ejemplo muestra un forma simple de usar *mmap*:

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"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 b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5]) # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline()) # prints b"Hello world!\n"
    # close the map
    mm.close()
```

mmap también puede ser usado como un gestor de contexto en una sentencia *with*

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

Nuevo en la versión 3.2: Soporte del Gestor de Contexto.

El siguiente ejemplo demuestra como crear un mapa anónimo y cambiar los datos entre los procesos padre e hijo:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")
```

(continué en la próxima página)

(proviene de la página anterior)

```

pid = os.fork()

if pid == 0: # In a child process
    mm.seek(0)
    print(mm.readline())

    mm.close()

```

Lanza un *evento de inspección* `mmap.__new__` con los argumentos `fileno`, `length`, `access`, `offset`.

Los objetos de archivos mapeados en memoria soportan los siguiente métodos:

close()

Cierra el *mmap*. Las llamadas posteriores a otros métodos del objeto resultarán en que se lance una excepción *ValueError*. Esto no cerrará el archivo abierto.

closed

True si el archivo está cerrado.

Nuevo en la versión 3.2.

find(sub[, start[, end]])

Retorna el índice mínimo en el objeto donde la subsecuencia *sub* es hallada, tal que *sub* este contenido en el rango `[start, end]`. Los argumentos opcionales *start* y *end* son interpretados como en una notación de rebanada. Retorna `-1` si falla.

Distinto en la versión 3.5: Ahora el objeto *bytes-like object* con permisos de escritura se acepta.

flush([offset[, size]])

Transmite los cambios hechos a la copia en memoria de una archivo de vuelta al archivo. Sin el uso de esta llamada no hay garantía que los cambios sean escritos de vuelta antes de que los objetos sean destruidos. Si *offset* y *size* son especificados, sólo los cambios al rango de bytes dado serán transmitidos al disco; de otra forma, la extensión completa al mapeado se transmite. *offset* debe ser un múltiplo de la constante `PAGESIZE` o `ALLOCATIONGRANULARITY`.

Se retorna `None` para indicar éxito. Una excepción es lanzada cuando la llamada falla.

Distinto en la versión 3.8: Anteriormente, se retornaba un valor diferente de cero cuando era exitoso; se retornaba cero cuando pasaba un error en Windows. Se retornaba un valor de cero cuando era exitoso; se lanzaba una excepción cuando pasaba un error en Unix.

madvice(option[, start[, length]])

Envía un aviso *option* al kernel sobre la región de la memoria que comienza con *start* y se extiende *length* bytes. *option* debe ser una de las *constantes* `MADV_*` disponibles en el sistema. Si *start* y *end* se omiten, se abarca al mapeo entero. En algunos sistemas (incluyendo Linux), *start* debe ser un múltiplo de `PAGESIZE`.

Disponibilidad: Sistemas con la llamada al sistema `madvice()`.

Nuevo en la versión 3.8.

move(dest, src, count)

Copia los *count* bytes empezando en el *offset* *src* al índice de destino *dest*. Si el *mmap* fue creado con `ACCESS_READ`, entonces las llamadas lanzaran una excepción *TypeError*.

read([n])

Retorna una clase *bytes* que contiene hasta *n* bytes empezando desde la posición del archivo actual. Si se omite el argumento, es `None` o negativo, retorna todos los bytes desde la posición actual del archivo hasta el final del mapeado. Se actualiza la posición del archivo para apuntar después de los bytes que se retornaron.

Distinto en la versión 3.3: El argumento puede ser omitido o ser `None`.

read_byte()

Retorna un byte en la posición actual del archivo como un entero, y avanza la posición del archivo por 1.

readline()

Retorna una sola línea, empezando desde la posición actual del archivo y hasta la siguiente nueva línea.

resize(newsize)

Redimensiona el mapa y el archivo subyacente, si lo hubiera. Si el *mmap* fue creado con `ACCESS_READ` o `ACCESS_COPY`, redimensionar el mapa lanzará una excepción *TypeError*.

rfind(sub[, start[, end]])

Retorna el índice más alto en el objeto donde la subsecuencia *sub* se encuentre, tal que *sub* sea contenido en el rango *[start, end]*. Los argumentos opcionales *start* y *end* son interpretados como una notación de rebanada. Retorna `-1` si falla.

Distinto en la versión 3.5: Ahora el objeto *bytes-like object* con permisos de escritura se acepta.

seek(pos[, whence])

Establece la posición actual del archivo. El argumento *whence* es opcional y es por defecto `os.SEEK_SET` o `0` (posicionamiento del archivo absoluto); otros valores son `os.SEEK_CUR` o `1` (búsqueda relativa a la posición actual) y `os.SEEK_END` o `2` (búsqueda relativa al final del archivo).

size()

Retorna el tamaño del archivo, que puede ser más grande que el tamaño del área mapeado en memoria.

tell()

Retorna la posición actual del puntero del archivo.

write(bytes)

Escribe los bytes en *bytes* en memoria en la posición actual del puntero del archivo y retorna el número de bytes escritos (nunca menos que `len(bytes)`), ya que si la escritura falla, una excepción *ValueError* será lanzada). La posición del archivo es actualizada para apuntar después de los bytes escritos. Si el *mmap* fue creado con `ACCESS_READ`, entonces escribirlo lanzará una excepción *TypeError*.

Distinto en la versión 3.5: Ahora el objeto *bytes-like object* con permisos de escritura se acepta.

Distinto en la versión 3.6: Ahora se retorna el número de bytes escritos.

write_byte(byte)

Escribe el entero *byte* en la memoria en la posición actual del puntero del archivo; se avanza la posición del archivo por 1. Si el *mmap* es creado con `ACCESS_READ`, entonces escribirlo hará que se lance la excepción *TypeError*.

18.9.1 Constantes `MADV_*`

```
mmap.MADV_NORMAL
mmap.MADV_RANDOM
mmap.MADV_SEQUENTIAL
mmap.MADV_WILLNEED
mmap.MADV_DONTNEED
mmap.MADV_REMOVE
mmap.MADV_DONTFORK
mmap.MADV_DOFORK
mmap.MADV_HWPOISON
mmap.MADV_MERGEABLE
mmap.MADV_UNMERGEABLE
mmap.MADV_SOFT_OFFLINE
mmap.MADV_HUGEPAGE
mmap.MADV_NOHUGEPAGE
mmap.MADV_DONTDUMP
mmap.MADV_DODUMP
mmap.MADV_FREE
mmap.MADV_NOSYNC
```


`mmap.MADV_AUTOSYNC`

`mmap.MADV_NOCORE`

`mmap.MADV_CORE`

`mmap.MADV_PROTECT`

Se pueden pasar estas opciones al método `mmap.madvise()`. No todas las opciones estarán presentes en todos los sistemas.

Disponibilidad: Sistemas con la llamada al sistema `madvise()`.

Nuevo en la versión 3.8.

Manejo de Datos de Internet

Este capítulo describe los módulos que admiten el manejo de formatos de datos que se usan comúnmente en Internet.

19.1 `email` — Un paquete de manejo de correo electrónico y MIME

Código fuente `Lib/email/__init__.py`

The `email` package is a library for managing email messages. 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 **RFC 5322** and **RFC 6532**, as well as such MIME-related RFCs as **RFC 2045**, **RFC 2046**, **RFC 2047**, **RFC 2183**, and **RFC 2231**.

La estructura general del paquete de correo electrónico se puede dividir en tres componentes principales, más un cuarto componente que controla el comportamiento de los otros componentes.

El componente central del paquete es un «modelo de objetos» que representa los mensajes de correo electrónico. Una aplicación interactúa con el paquete principalmente a través de la interfaz del modelo de objetos definida en el submódulo `message`. La aplicación puede usar esta API para hacer preguntas sobre un correo electrónico existente, para construir un nuevo correo electrónico o para agregar o eliminar subcomponentes de correo electrónico que utilizan la misma interfaz de modelo de objetos. Es decir, siguiendo la naturaleza de los mensajes de correo electrónico y sus subcomponentes MIME, el modelo de objetos de correo electrónico es una estructura de árbol de objetos que proporcionan la API `EmailMessage`.

Los otros dos componentes principales del paquete son `parser` y `generator`. El parser toma la versión serializada de un mensaje de correo electrónico (una secuencia de bytes) y la convierte en un árbol de objetos `EmailMessage`. El generador toma un `EmailMessage` y lo convierte de nuevo en un flujo de bytes serializado. (El analizador y el generador también manejan flujos de caracteres de texto, pero se desaconseja este uso ya que es demasiado fácil terminar con mensajes que no son válidos de una forma u otra).

El componente de control es el módulo de `policy`. Cada `EmailMessage`, cada `generator`, y cada `parser` tiene un objeto de `policy` asociado que controla su comportamiento. Por lo general, una aplicación solo necesita especificar la política cuando se crea un `EmailMessage`, ya sea instanciando directamente un `EmailMessage` para crear un nuevo correo electrónico o analizando un flujo de entrada con un `parser`. Pero la política se puede cambiar cuando el mensaje se serializa mediante un `generator`. Esto permite, por ejemplo, analizar un mensaje de correo electrónico genérico desde el disco, pero serializarlo utilizando la configuración estándar de SMTP al enviarlo a un servidor de correo electrónico.

El paquete de correo electrónico hace todo lo posible para ocultar los detalles de las diversas RFC que rigen de la aplicación. Conceptualmente, la aplicación debería poder tratar el mensaje de correo electrónico como un árbol estructurado de texto Unicode y archivos adjuntos binarios, sin tener que preocuparse por cómo se representan estos cuando se serializan. En la práctica, sin embargo, a menudo es necesario conocer al menos algunas de las reglas que rigen los mensajes MIME y su estructura, específicamente los nombres y la naturaleza de los «tipos de contenido» MIME y cómo identifican los documentos de varias partes. En su mayor parte, este conocimiento solo debería ser necesario para aplicaciones más complejas, e incluso entonces debería ser solo la estructura de alto nivel en cuestión, y no los detalles de cómo se representan esas estructuras. Dado que los tipos de contenido MIME se utilizan ampliamente en el software moderno de Internet (no solo en el correo electrónico), este será un concepto familiar para muchos programadores.

Las siguientes secciones describen la funcionalidad del paquete `email`. Comenzamos con el modelo de objetos `message`, que es la interfaz principal que usará una aplicación, y lo seguimos con los componentes del `parser` y `generator`. Luego cubrimos los controles de la `policy`, lo que completa el tratamiento de los principales componentes de la biblioteca.

Las siguientes tres secciones cubren las excepciones que puede generar el paquete y los defectos (incumplimiento de las RFC) que el `parser` puede detectar. Luego cubrimos los subcomponentes `headerregistry` y `contentmanager`, que proporcionan herramientas para realizar una manipulación más detallada de los encabezados y cargas útiles, respectivamente. Ambos componentes contienen características relevantes para consumir y producir mensajes no triviales, pero también documentan sus API de extensibilidad, que serán de interés para aplicaciones avanzadas.

A continuación, se muestra un conjunto de ejemplos del uso de las partes fundamentales de las API cubiertas en las secciones anteriores.

Lo anterior representa la API moderna (compatible con Unicode) del paquete de correo electrónico. Las secciones restantes, comenzando con la clase `Message`, cubren la API `compat32` heredada que trata mucho más directamente con los detalles de cómo se representan los mensajes de correo electrónico. La API `compat32` no oculta los detalles de las RFC de la aplicación, pero para las aplicaciones que necesitan operar a ese nivel, pueden ser herramientas útiles. Esta documentación también es relevante para las aplicaciones que todavía usan la API `compat32` por razones de compatibilidad con versiones anteriores.

Distinto en la versión 3.6: Documentos reorganizados y reescritos para promover la nueva API `EmailMessage/EmailPolicy`.

Contenido de la documentación del paquete `email`:

19.1.1 `email.message`: Representando un mensaje de correo electrónico

Código fuente: `Lib/email/message.py`

Nuevo en la versión 3.6:¹

La clase central en el paquete de `email` es la clase `EmailMessage`, importada desde el módulo `email.message`. Esta es la clase base para el modelo de objeto `email`. `EmailMessage` provee la funcionalidad clave para configurar y consultar los `headers`, para acceder al cuerpo del mensaje, y para crear o modificar la estructura del mensaje.

Un mensaje de e-mail consiste en `headers` y un `payload` (al que también nos referimos como `content`). `Headers` como **RFC 5322** o **RFC 6532** son nombres de campos de estilo y valores, donde el nombre y valor están separados por un “:”. Los dos puntos no son parte ni del nombre ni del valor. El `payload` puede ser un simple mensaje, un objeto binario, o una secuencia estructurada de sub-mensajes, cada uno con su propio conjunto de `headers` y su propio `payload`. El último tipo de `payload` es indicado por el mensaje con un MIME como `multipart/*` o `message/rfc822`.

El modelo conceptual provisto por un objeto `EmailMessage` es el de un diccionario ordenado de `headers` emparejados con un `payload` que representa al cuerpo del mensaje **RFC 5322**, que podría ser una lista de objetos sub-`EmailMessage`. Además de los métodos normales de diccionario para acceder a los `headers` y valores, tiene

¹ Originally added in 3.4 as a *provisional module*. Docs for legacy message class moved to `email.message.Message`: Representar un mensaje de correo electrónico usando la API `compat32`.

métodos para acceder a información especializada desde los *headers* (por ejemplo, el tipo de contenido MIME), para operar en el *payload*, generar una versión serializada del mensaje, y recorrer recursivamente el árbol de objetos.

The *EmailMessage* dictionary-like interface is indexed by the header names, which must be ASCII values. The values of the dictionary are strings with some extra methods. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. Unlike a real dict, there is an ordering to the keys, and there can be duplicate keys. Additional methods are provided for working with headers that have duplicate keys.

El *payload* es un objeto de cadena de caracteres o bytes, en el caso de objetos de mensaje simples, o una lista de objetos *EmailMessage*, para documentos de contenedor MIME como *multipart/** y objetos de mensaje *message/rfc822*.

class email.message.**EmailMessage** (*policy=default*)

Si se especifica *policy*, use las reglas especificadas para actualizar y serializar la representación del mensaje. Si *policy* no es establecida, use *default*, que sigue las reglas RFC de email excepto para el fin de línea (en lugar del RFC `\r\n`, usa los finales estándar de Python `\n` como final de línea). Para más información ve a la documentación del *policy*.

as_string (*unixfrom=False, maxheaderlen=None, policy=None*)

Retorna el mensaje entero como cadena de caracteres. Cuando la opción *unixform* es verdadera, el *header* está incluido en la cadena de caracteres retornada. *unixform* está predeterminado con valor *False*. Por compatibilidad con versiones anteriores, la base *Message*, la case *maxheaderlen* es aceptada pero con valor *None* como predeterminado, por lo que la longitud de línea se controla mediante *max_line_length*. El argumento *policy* puede ser usado para anular el valor predeterminado obtenido de la instancia del mensaje. Esto puede ser usado para controlar parte del formato producido por el método, ya que el *policy* especificado pasará a *Generator*.

Aplanar el mensaje puede acarrear cambios en *EmailMessage* si es necesario rellenar los valores predeterminados para completar la transformación a una cadena de caracteres (por ejemplo, se pueden generar o modificar límites MIME).

Tenga en cuenta que este método se proporciona como una comodidad y quizás no sea la forma más eficiente de serializar mensajes en su aplicación, especialmente si estás tratando con múltiples mensajes. Consulte *Generator* por una API más flexible para serializar mensajes. No olvide también que este método está restringido a producir mensajes serializados como «7 bit clean» cuando *utf8* es *False*, que es el valor predeterminado.

Distinto en la versión 3.6: el comportamiento predeterminado cuando *maxheaderlen* no está especificado cambió de 0 al valor de *max_line_length*.

__str__ ()

Equivalente a `as_string(policy=self.policy.clone(utf8=True))`. Permite `str(msg)` para producir un *string* que contenga un mensaje serializado en un formato legible.

Distinto en la versión 3.4: el método se cambió para usar *utf8=True*, produciendo así un **RFC 6531** como representación del mensaje, en vez de ser un alias de *as_string()*.

as_bytes (*unixfrom=False, policy=None*)

Retorna el mensaje plano como un objeto de bytes. Cuando *unixform* es verdadero, el *header* es incluido en la cadena de caracteres retornada. El valor predeterminado de *unixform* es *False*. El argumento *policy* puede ser usado para sobrescribir el valor predeterminado obtenido de la instancia del mensaje. Esto puede ser usado para controlar parte del formato producido por el método, ya que el *policy* especificado pasará a *Generator*.

Aplanar el mensaje puede acarrear cambios en *EmailMessage* si es necesario rellenar los valores predeterminados para completar la transformación a una cadena de caracteres (por ejemplo, se pueden generar o modificar límites MIME).

Tenga en cuenta que este método se proporciona como una comodidad y quizás no sea la forma más eficiente de serializar mensajes en su aplicación, especialmente si estas tratando con múltiples mensajes. Consulte *Generator* por una API más flexible para serializar mensajes.

__bytes__ ()

Equivalente a `as_bytes()`. Permite `bytes(msg)` para producir un objeto byte que contenga el mensaje serializado.

is_multipart()

Retorna True si el *payload* del mensaje es una lista de objetos de sub-*EmailMessage*, de otra manera retorna False. Cuando *is_multipart()* retorna False, el *payload* deberá ser un objeto cadena de caracteres (que podría ser un *payload* binario codificado con CTE). Note que si *is_multipart()* retorna True no necesariamente significa que «*msg.get_content_maintype() == "multipart"*» retornará True. Por ejemplo, *is_multipart* retornará True cuando la *EmailMessage* sea del tipo *message/rfc822*.

set_unixfrom(unixfrom)

Configura la cabecera del mensaje a *unixfrom*, que debería ser una cadena de caracteres. (Consulte *mboxMessage* para una descripción de este *header*)

get_unixfrom()

Retorna la cabecera del mensaje. Predeterminado None si la cabecera no ha sido configurada.

Los siguientes métodos implementan el mapeo como una interfaz para acceder al *header* del mensaje. Tenga en cuenta que hay algunas diferencias semánticas entre esos métodos y una interfaz de mapeo normal (es decir, diccionario). Por ejemplo, en un diccionario no hay claves duplicadas, pero pueden haber *headers* duplicados. Además, en los diccionarios no hay un orden garantizado para las claves retornadas por *keys()*, pero en un objeto *EmailMessage*, los *headers* siempre regresan en orden de aparición en el mensaje original, o en el que fueron agregados luego. Cualquier *header* borrado y vuelto a añadir siempre se agrega al final de la lista.

Estas diferencias semánticas son intencionales y están sesgadas hacia la conveniencia en los casos de uso más comunes.

Note que en todos los casos, cualquier *header* presente en el mensaje no se incluye en la interfaz de mapeo.

__len__()

Retorna el número total de *headers*, incluidos los duplicados.

__contains__(name)

Retorna True si el objeto del mensaje tiene un campo llamado “nombre”. La comparación se realiza sin tener en cuenta mayúsculas o minúsculas y “nombre” no incluye “:”. Se utiliza para el operador *in*. Por ejemplo:

```
if 'message-id' in myMessage:
    print('Message-ID:', myMessage['message-id'])
```

__getitem__(name)

Retorna el valor del *header* nombrado. *name* no incluye el separador de dos puntos, “:”. Si el *header* se pierde, regresa None, un *KeyError* no se genera nunca.

Tenga en cuenta que si el campo nombrado aparece más de una vez en el *header* del mensaje, esa cantidad de veces el valor regresado será indefinido. Use el método *get_all()* para obtener los valores de todos los *headers* existentes llamados *name*.

Usando el *standard non-“compat32”*, el valor regresado es una instancia de una subclase de *email.headerregistry.BaseHeader*.

__setitem__(name, val)

Agrega un *header* al mensaje con un campo “nombre” y un valor “val”. El campo se agrega al final de los *headers* existentes en el mensaje.

Tenga en cuenta que esto no sobrescribe ni borra ningún *header* con el mismo nombre. Si quiere asegurarse de que el nuevo *header* es el único en el mensaje con el campo “nombre”, borre el campo primero, por ejemplo:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

Si el *policy* define ciertos *headers* para ser únicos (como lo hace el *standard*), este método puede generar un *ValueError* cuando se intenta asignar un valor a un *header* preexistente. Este comportamiento es intencional por consistencia, pero no dependa de ello, ya que podemos optar por hacer que tales asignaciones eliminen el *header* en el futuro.

__delitem__ (*name*)

Elimine todas las apariciones del campo “nombre” de los *headers* del mensaje. No se genera ninguna excepción si el campo nombrado no está presente en los encabezados.

keys ()

Retorna una lista de los nombres de todos los campos del *header* del mensaje.

values ()

Retorna una lista de todos los valores de los campos del mensaje.

items ()

Retorna una lista de 2 tuplas que contienen todos los campos *header* y *value* del mensaje.

get (*name*, *failobj=None*)

Retorna el valor del *header* nombrado. Esto es idéntico a **__getitem__** () excepto si el opcional *failobj* regresado en el *header* nombrado no se encuentra (*failobj* es por defecto *None*).

Aquí hay algunos métodos adicionales útiles relacionados con el *header*:

get_all (*name*, *failobj=None*)

Retorna una lista de todos los valores para el campo *nombre*. Si no se nombran tales *headers* en el mensaje, regresa *failobj* (por defecto *None*)

add_header (*_name*, *_value*, ***_params*)

Configuración extendida de *headers*. Este método es similar a **__setitem__** (), excepto que se pueden proporcionar parámetros de *header* adicionales como argumentos de palabras clave.

Por cada ítem en los parámetros del diccionario *_params*, la clave se toma como el nombre del parámetro, con barra baja (“_”) convertidos a guiones (“-”) (ya que en Python no se permiten “-” como identificadores). Normalmente, el parámetro debe ser añadido como *key='value'* a menos que el valor sea *None*, en ese caso solo la clave debe ser añadida.

Si el valor contiene caracteres no-ASCII, el *charset* y el lenguaje deben ser controlados especificando el valor como una triple tupla en formato (*CHARSET*, *LANGUAJE*, *VALUE*), donde *CHARSET* es una *string* llamando al *charset* usado para codificar el valor, *LANGUAJE* generalmente se establece en *None* o en una cadena de caracteres vacía (consulte [RFC 2231](#) para más opciones), y *VALUE* es el valor de la cadena de caracteres que contiene puntos de código no-ASCII. Si la triple tupla no pasa y el valor contiene caracteres no-ASCII, es automáticamente codificada en formato [RFC 2231](#), usando *CHARSET* de *utf-8* y *LANGUAJE* *None*.

Aquí hay un ejemplo:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

Esto agregará un *header* que se ve como:

```
Content-Disposition: attachment; filename="bud.gif"
```

Un ejemplo de la interfaz extendida con caracteres no-ASCII:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

replace_header (*_name*, *_value*)

Reemplaza un *header*. Reemplaza el primer *header* encontrado en el mensaje que coincida con *_name*, conservando el orden de *header* y uso de minúsculas (y mayúsculas) del nombre de campo del *header* original. Si no hay coincidencia, se lanzará un *KeyError*.

get_content_type ()

Retorna el tipo de contenido del mensaje, pasado a minúsculas de la forma *maintype/subtype*. Si no hay *header* llamado *Content-Type* en el mensaje, regresa el valor de *get_default_type* (). Si *Content-Type* no es válido, retorna *text/plain*.

(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 ap-

pears 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*.)

get_content_maintype()

Return the message's main content type. This is the *maintype* part of the string returned by *get_content_type()*.

get_content_subtype()

Return the message's sub-content type. This is the *subtype* part of the string returned by *get_content_type()*.

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*.

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, so it only affects the return value of the *get_content_type* methods when no *Content-Type* header is present in the message.

set_param(param, value, header='Content-Type', requote=True, charset=None, language="", replace=False)

Set a parameter in the *Content-Type* header. If the parameter already exists in the header, replace its value with *value*. When *header* is *Content-Type* (the default) and the header does not yet exist in the message, add it, set its value to *text/plain*, and append the new parameter value. Optional *header* specifies an alternative header to *Content-Type*.

If the value contains non-ASCII characters, the *charset* and *language* may be explicitly specified using the optional *charset* and *language* parameters. Optional *language* specifies the **RFC 2231** language, defaulting to the empty string. Both *charset* and *language* should be strings. The default is to use the *utf8 charset* and *None* for the *language*.

If *replace* is *False* (the default) the header is moved to the end of the list of headers. If *replace* is *True*, the header will be updated in place.

El uso del parámetro *requote* con objetos *EmailMessage* está obsoleto.

Note that existing parameter values of headers may be accessed through the *params* attribute of the header value (for example, `msg['Content-Type'].params['charset']`).

Distinto en la versión 3.4: Se agregó la palabra clave *replace*.

del_param(param, header='content-type', requote=True)

Remove the given parameter completely from the *Content-Type* header. The header will be re-written in place without the parameter or its value. Optional *header* specifies an alternative to *Content-Type*.

El uso del parámetro *requote* con objetos *EmailMessage* está obsoleto.

get_filename(failobj=None)

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=None)

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 from 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.

get_content_charset (*failobj=None*)

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.

get_charsets (*failobj=None*)

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. 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*.

is_attachment ()

Return True if there is a *Content-Disposition* header and its (case insensitive) value is attachment, False otherwise.

Distinto en la versión 3.4.2: `is_attachment` is now a method instead of a property, for consistency with `is_multipart()`.

get_content_disposition ()

Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or None. The possible values for this method are *inline*, *attachment* or None if the message follows **RFC 2183**.

Nuevo en la versión 3.5.

Los siguientes métodos se refieren a interrogar y manipular el contenido (*payload*) del mensaje.

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.

Aquí hay un ejemplo que imprime el tipo MIME de cada parte de una estructura de mensaje de varias partes:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

`walk` iterates over the subparts of any part where `is_multipart()` returns True, even though `msg.get_content_maintype() == 'multipart'` may return False. We can see this in our example by making use of the `_structure` debug helper function:

```
>>> from email.iterators import _structure
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
```

(continué en la próxima página)

(proviene de la página anterior)

```
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
  text/plain
  message/delivery-status
    text/plain
    text/plain
  message/rfc822
    text/plain
```

Here the message parts are not multipart, but they do contain subparts. `is_multipart()` returns True and `walk` descends into the subparts.

get_body (*preferencelist*=('related', 'html', 'plain'))

Retorna la parte MIME que es la mejor candidata para ser el «cuerpo» del mensaje.

preferencelist must be a sequence of strings from the set `related`, `html`, and `plain`, and indicates the order of preference for the content type of the part returned.

Empieza a buscar coincidencias candidatas con el objeto en el que se llama al método `get_body`”.

If `related` is not included in *preferencelist*, consider the root part (or subpart of the root part) of any related encountered as a candidate if the (sub-)part matches a preference.

When encountering a `multipart/related`, check the `start` parameter and if a part with a matching *Content-ID* is found, consider only it when looking for candidate matches. Otherwise consider only the first (default root) part of the `multipart/related`.

If a part has a *Content-Disposition* header, only consider the part a candidate match if the value of the header is `inline`.

Si ninguno de los candidatos coincide con ninguna de las preferencias en *preferencelist*, retorna `None`.

Notes: (1) For most applications the only *preferencelist* combinations that really make sense are ('plain',), ('html', 'plain'), and the default ('related', 'html', 'plain'). (2) Because matching starts with the object on which `get_body` is called, calling `get_body` on a `multipart/related` will return the object itself unless *preferencelist* has a non-default value. (3) Messages (or message parts) that do not specify a *Content-Type* or whose *Content-Type* header is invalid will be treated as if they are of type `text/plain`, which may occasionally cause `get_body` to return unexpected results.

iter_attachments ()

Return an iterator over all of the immediate sub-parts of the message that are not candidate «body» parts. That is, skip the first occurrence of each of `text/plain`, `text/html`, `multipart/related`, or `multipart/alternative` (unless they are explicitly marked as attachments via *Content-Disposition: attachment*), and return all remaining parts. When applied directly to a `multipart/related`, return an iterator over the all the related parts except the root part (ie: the part pointed to by the `start` parameter, or the first part if there is no `start` parameter or the `start` parameter doesn't match the *Content-ID* of any of the parts). When applied directly to a `multipart/alternative` or a non-multipart, return an empty iterator.

iter_parts ()

Return an iterator over all of the immediate sub-parts of the message, which will be empty for a non-multipart. (See also `walk()`.)

get_content (*args, *content_manager*=None, **kw)

Call the `get_content()` method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the *content_manager* specified by the current *policy*.

set_content (*args, content_manager=None, **kw)

Call the `set_content()` method of the `content_manager`, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If `content_manager` is not specified, use the `content_manager` specified by the current `policy`.

make_related (boundary=None)

Convert a non-multipart message into a multipart/related message, moving any existing `Content-` headers and payload into a (new) first part of the multipart. If `boundary` is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_alternative (boundary=None)

Convert a non-multipart or a multipart/related into a multipart/alternative, moving any existing `Content-` headers and payload into a (new) first part of the multipart. If `boundary` is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_mixed (boundary=None)

Convert a non-multipart, a multipart/related, or a multipart-alternative into a multipart/mixed, moving any existing `Content-` headers and payload into a (new) first part of the multipart. If `boundary` is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

add_related (*args, content_manager=None, **kw)

If the message is a multipart/related, create a new message object, pass all of the arguments to its `set_content()` method, and `attach()` it to the multipart. If the message is a non-multipart, call `make_related()` and then proceed as above. If the message is any other type of multipart, raise a `TypeError`. If `content_manager` is not specified, use the `content_manager` specified by the current `policy`. If the added part has no `Content-Disposition` header, add one with the value `inline`.

add_alternative (*args, content_manager=None, **kw)

If the message is a multipart/alternative, create a new message object, pass all of the arguments to its `set_content()` method, and `attach()` it to the multipart. If the message is a non-multipart or multipart/related, call `make_alternative()` and then proceed as above. If the message is any other type of multipart, raise a `TypeError`. If `content_manager` is not specified, use the `content_manager` specified by the current `policy`.

add_attachment (*args, content_manager=None, **kw)

If the message is a multipart/mixed, create a new message object, pass all of the arguments to its `set_content()` method, and `attach()` it to the multipart. If the message is a non-multipart, multipart/related, or multipart/alternative, call `make_mixed()` and then proceed as above. If `content_manager` is not specified, use the `content_manager` specified by the current `policy`. If the added part has no `Content-Disposition` header, add one with the value `attachment`. This method can be used both for explicit attachments (`Content-Disposition: attachment`) and inline attachments (`Content-Disposition: inline`), by passing appropriate options to the `content_manager`.

clear()

Elimina el `payload` y todos los `headers`.

clear_content()

Remove the payload and all of the `Content-` headers, leaving all other headers intact and in their original order.

`EmailMessage` objects have the following instance attributes:

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. As with the *preamble*, if there is no epilog text this attribute will be `None`.

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.

class `email.message.MIMEPart (policy=default)`

This class represents a subpart of a MIME message. It is identical to *EmailMessage*, except that no *MIME-Version* headers are added when *set_content()* is called, since sub-parts do not need their own *MIME-Version* headers.

Notas al pie

19.1.2 email.parser: Analizar mensajes de correo electrónico

Código fuente: [Lib/email/parser.py](#)

Se pueden construir estructuras de objetos de mensaje de dos formas: pueden ser creados de puro invento al crear un objeto *EmailMessage*, añadir encabezados usando la interfaz de diccionario, y añadir carga(s) usando el método *set_content()* y otros relacionados, o pueden ser creados al analizar una representación serializada de un mensaje de correo electrónico.

El paquete *email* proporciona un analizador estándar que entiende la mayoría de estructuras de documentos de correo electrónico, incluyendo documentos MIME. Le puedes pasar al analizador bytes, una cadena de caracteres o una archivo de objeto, y el analizador te retornará la instancia *EmailMessage* raíz de la estructura del objeto. Para mensajes simples que no sean MIME, la carga de su objeto raíz probablemente será una cadena de caracteres conteniendo el texto o el mensaje. Para mensajes MIME, el objeto raíz retornará `True` de su método *is_multipart()*, y las subpartes pueden ser accedidas a través de los métodos de manipulación de carga, tales como *get_body()*, *iter_parts()*, y *walk()*.

De hecho hay dos interfaces de analizadores disponibles para usar, la API *Parser* y la API progresiva *FeedParser*. La API *Parser* es más útil si tú tienes el texto del mensaje entero en memoria, o si el mensaje entero reside en un archivo en el sistema. *FeedParser* es más apropiado cuando estás leyendo el mensaje de un *stream* que puede ser bloqueado esperando más entrada (tal como leer un mensaje de correo electrónico de un socket). El *FeedParser* puede consumir y analizar el mensaje de forma progresiva, y sólo retorna el objeto raíz cuando cierras el analizador.

Tenga en cuenta que el analizador puede ser extendido en formas limitadas, y por supuesto puedes implementar tu propio analizador completamente desde cero. Toda la lógica que conecta el analizador empaquetado del paquete *email* y la clase *EmailMessage* está encarnada en la clase *policy*, por lo que un analizador personalizado puede crear árboles de objetos mensaje en cualquier forma que encuentre necesario al implementar versiones personalizadas de los métodos apropiados de *policy*.

API *FeedParser*

La clase *BytesFeedParser*, importado del módulo `email.feedparser`, proporciona una API que es propicia para el análisis progresivo de mensajes de correo electrónico, tal como sería necesario cuando se esté leyendo el texto de un mensaje de correo electrónico de una fuente que puede bloquear (tal como un socket). Desde luego se puede usar la clase *BytesFeedParser* para analizar un mensaje de correo electrónico completamente contenido en un *bytes-like object*, cadena de caracteres, o archivo, pero la API *BytesParser* puede ser más conveniente para tales casos de uso. Las semánticas y resultados de las dos API de los analizadores son idénticas.

La API de *BytesFeedParser* es simple; puedes crear una instancia, le proporcionas un montón de bytes hasta que no haya más necesidad de hacerlo, entonces cierras el analizador para recuperar el objeto del mensaje raíz. El *BytesFeedParser* es extremadamente preciso cuando está analizando mensajes conformes al estándar, y hace un buen trabajo al analizar mensajes no conformes, proporcionando información acerca de cómo un mensaje fue considerado inservible. Ingresará una lista de cualquier problema que encontró en el atributo *defects* del objeto mensaje. Véase el módulo `email.errors` para la lista de defectos que puede encontrar.

Aquí está el API para *BytesFeedParser*:

class `email.parser.BytesFeedParser` (*_factory=None*, *, *policy=policy.compat32*)

Crea una instancia de *BytesFeedParser*. El argumento opcional *_factory* es un invocable sin argumentos; si no se especifica, usa el *message_factory* de *policy*. Llama a *_factory* cuando sea necesario un nuevo objeto mensaje.

Si se especifica *policy*, usa las reglas que especifica para actualizar la representación del mensaje. Si *policy* no está puesta, usa la política (*policy*) *compat32*, que mantiene compatibilidad con la versión 3.2 de Python del paquete de correo electrónico y proporciona a *Message* como la fábrica por defecto. Todas las otras políticas proveen a *EmailMessage* como el *_factory* por defecto. Para más información en lo demás que *policy* controla, véase la documentación *policy*.

Nota: La palabra clave **policy** siempre debe estar especificada; El valor por defecto cambiará a `email.policy.default` en una versión futura de Python.

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Se añadió la palabra clave *policy*.

Distinto en la versión 3.6: *_factory* es por defecto la *policy message_factory*.

feed (*data*)

Le proporciona al analizador algunos datos más. *data* debe ser un *bytes-like object* conteniendo una o más líneas. Las líneas pueden ser parciales y el analizador va a juntar tales líneas parciales apropiadamente. las líneas pueden tener cualquiera de las tres terminaciones de línea comunes: retorno de cargo (*retorno de cargo*), nueva línea (*newline*), o retorno de cargo y nueva línea (pueden ser mezclados).

close ()

Completa el análisis de todos los datos previamente proporcionados y retorna la raíz del objeto mensaje. No está definido lo que pasa si se llama a *feed()* después de que este método haya sido llamado.

class `email.parser.FeedParser` (*_factory=None*, *, *policy=policy.compat32*)

Funciona como *BytesFeedParser* excepto que la entrada al método *feed()* no debe ser una cadena de caracteres. Esto es utilidad limitada, ya que la única manera de que tal mensaje sea válido es que sólo contenga texto ASCII o, si *utf8* es *True*, sin binarios adjuntos.

Distinto en la versión 3.3: Se añadió la palabra clave *policy*.

API Parser

La clase `BytesParser`, importado del módulo `email.parser`, proporciona una API que puede ser usada para analizar un mensaje cuando el contenido completo del mensaje esté disponible en un *bytes-like object* o archivo. El módulo `email.parser` también proporciona a `Parser` para analizar cadenas de caracteres, y analizadores de sólo cabeceras, `BytesHeaderParser` y `HeaderParser` que pueden ser usados si sólo estás interesado en las cabeceras del mensaje. `BytesHeaderParser` y `HeaderParser` puede ser más rápidos en estas situaciones, ya que no intentan analizar el cuerpo del mensaje, en vez de eso configuran la carga al cuerpo puro.

class `email.parser.BytesParser` (*_class=None*, *, *policy=policy.compat32*)

Crea una instancia de `BytesParser`. Los argumentos *_class* y *policy* tiene el mismo significado y semántica que los argumentos *_factory* y *policy* de `BytesFeedParser`.

Nota: La palabra clave ***policy*** siempre debe estar especificada; El valor por defecto cambiará a `email.policy.default` en una versión futura de Python.

Distinto en la versión 3.3: Se eliminó el argumento *strict* que fue deprecado en 2.4. Se añadió la palabra clave *policy*.

Distinto en la versión 3.6: *_class* es por defecto la política `message_factory`.

parse (*fp*, *headersonly=False*)

Lee todos los datos del objeto binario parecido a archivo *fp*, analiza los bytes resultantes, y retorna el objeto mensaje. *fp* debe soportar tanto el método `readline()` como el método `read()`.

Los bytes contenidos en *fp* deben ser formateados como un bloque de cabeceras de estilo y líneas de continuación de cabecera de **RFC 5322** (o, si `utf8` es `True`, **RFC 6532**). El bloque cabecera se termina o al final de los datos o por una línea blanca. Después del bloque de cabecera esta el cuerpo del mensaje (que puede contener subpartes codificadas como MIME, incluyendo subpartes con un *Content-Transfer-Encoding* de 8bit).

El argumento opcional *headersonly* es un flag que especifica si se debe analizar después de leer las cabeceras o no. El valor por defecto es `False`, significando que analiza el contenido entero del archivo.

parsebytes (*bytes*, *headersonly=False*)

Similar al método `parse()`, excepto que toma un *bytes-like object* en vez de un objeto similar a un archivo. Llamar a este método en un *bytes-like object* es equivalente a envolver a *bytes* en una instancia de `BytesIO` primero y llamar a `parse()`.

El argumento opcional *headersonly* es como el método `parse()`.

Nuevo en la versión 3.2.

class `email.parser.BytesHeaderParser` (*_class=None*, *, *policy=policy.compat32*)

Exactamente como `BytesParser`, excepto que *headersonly* es por defecto `True`.

Nuevo en la versión 3.3.

class `email.parser.Parser` (*_class=None*, *, *policy=policy.compat32*)

Esta clase es paralela a `BytesParser`, pero trata entradas de cadenas de caracteres.

Distinto en la versión 3.3: Se eliminó el argumento *strict*. Se añadió la palabra clave *policy*.

Distinto en la versión 3.6: *_class* es por defecto la política `message_factory`.

parse (*fp*, *headersonly=False*)

Lee todos los datos del modo texto del objeto parecido a archivo *fp*, analiza el texto resultante, y retorna el objeto mensaje raíz. *fp* debe soportar tanto el método `readline()` y el método `read()` en objetos parecidos a archivos.

Además de el requisito del modo texto, este método opera como `BytesParser.parse()`.

parsestr (*text*, *headersonly=False*)

Similar al método `parse()`, excepto que toma un objeto de cadena de caracteres de un objeto similar a un archivo. Llamar a este método en una cadena de caracteres es equivalente a envolver a *text* en una instancia de `StringIO` primero y llamar a `parse()`.

El argumento opcional *headersonly* es como el método *parse()*.

class email.parser.HeaderParser(_class=None, *, policy=policy.compat32)

Exactamente como *Parser*, excepto que *headersonly* es por defecto *True*.

Ya que crear una estructura de un objeto mensaje de una cadena de caracteres o un objeto archivo es una tarea tan común, Se proporcionaron 4 funciones como una conveniencia. Están disponibles en paquete de espacio de nombres de alto nivel *email*.

email.message_from_bytes(s, _class=None, *, policy=policy.compat32)

Retorna una estructura del objeto mensaje de un *bytes-like object*. Esto es equivalente a *BytesParser()*. *parsebytes(s)*. El argumento opcional *_class* y *policy* son interpretados como sucede con el constructor de clase *BytesParser*.

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Se eliminó el argumento *strict*. Se añadió la palabra clave *policy*.

email.message_from_binary_file(fp, _class=None, *, policy=policy.compat32)

Retorna una estructura árbol del objeto mensaje de un *file object* binario abierto. Esto es equivalente a *BytesParser().parse(fp)*. *_class* y *policy* son interpretados como sucede con el constructor de clase *BytesParser*.

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Se eliminó el argumento *strict*. Se añadió la palabra clave *policy*.

email.message_from_string(s, _class=None, *, policy=policy.compat32)

Retorna una estructura del objeto mensaje de una cadena de caracteres. Esto es equivalente a *Parser().parsestr(s)*. *_class* y *policy* son interpretados como sucede con el constructor de clase *Parser*.

Distinto en la versión 3.3: Se eliminó el argumento *strict*. Se añadió la palabra clave *policy*.

email.message_from_file(fp, _class=None, *, policy=policy.compat32)

Retorna una estructura árbol del objeto mensaje de un *file object* abierto. Esto es equivalente a *Parser().parse(fp)*. *_class* y *policy* son interpretados como sucede con el constructor de clase *Parser*.

Distinto en la versión 3.3: Se eliminó el argumento *strict*. Se añadió la palabra clave *policy*.

Distinto en la versión 3.6: *_class* es por defecto la política *message_factory*.

Aquí está un ejemplo de cómo puedes usar *message_from_bytes()* en una entrada interactiva de Python:

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

Notas adicionales

Aquí están algunas notas sobre la semántica del análisis:

- La mayoría de los mensajes de tipo que no son *multipart* son actualizados como un solo objeto mensaje con una carga de cadena de caracteres. Estos objetos retornarán *False* para *is_multipart()*, y *iter_parts()* cederá (*yield*) una lista vacía.
- Todos los mensajes de tipo *multipart* serán analizados como un objeto mensaje contenedor con una lista de objetos sub-mensajes para sus cargas. El mensaje del contenedor externo retornará *True* para *is_multipart()*, y *iter_parts()* cederá (*yield*) una lista de subpartes.
- La mayoría de mensajes con una tipo de contenido de *message/** (tal como *message/delivery-status* y *message/rfc822*) también serán analizados como objetos contenedores que contienen una lista de cargas de longitud 1. Su método *is_multipart()* retornará *True*. El único elemento cedido (*yielded*) por *iter_parts()* será un objeto sub-mensaje.
- Algunos mensajes de conformidad no estándar pueden no ser internamente consistentes acerca de su *multipart*-idad. Tales mensajes pueden tener una cabecera *Content-Type* de tipo *multipart*, pero su método *is_multipart()* puede retornar *False*. Si tales mensajes son analizados con *FeedParser*,

tendrán una instancia de la clase `MultipartInvariantViolationDefect` en su lista de atributos *defects*. Véase `email.errors` para más detalles.

19.1.3 `email.generator`: Generando documentos MIME

Código fuente: `Lib/email/generator.py`

Una de las tareas más comunes es generar la versión plana (serializada) del mensaje de correo electrónico representado por una estructura de objeto de mensaje. Se tendrá que hacer esto si se desea enviar un mensaje a través de `smtpplib.SMTP.sendmail()`, o el módulo `nntplib`, o imprimir el mensaje en la consola. Tomar una estructura de objeto de mensaje y producir una representación serializada es el trabajo de las clases generadoras.

Al igual que con el módulo `email.parser`, no se limita a la funcionalidad del generador incluido; se podría escribir uno desde cero. Sin embargo, el generador incluido sabe cómo generar la mayoría del correo electrónico de una manera compatible con los estándares, debería controlar los mensajes de correo electrónico MIME y no MIME bien. Está diseñado para que las operaciones de análisis y generación orientadas a bytes sean inversas, asumiendo que se utilice la misma *policy* de no transformación para ambos. Es decir, analizar la secuencia de bytes serializada a través de la clase `BytesParser` y, a continuación, regenerar la secuencia de bytes serializada mediante `BytesGenerator` debe producir una salida idéntica a la entrada¹. (Por otro lado, el uso del generador en un `EmailMessage` construido por el programa puede dar lugar a cambios en el objeto `EmailMessage` a medida que se rellenan los valores predeterminados.)

La clase `Generator` se puede utilizar para acoplar un mensaje en una representación serializada de texto (a diferencia de la binaria). Sin embargo, como Unicode no puede representar datos binarios directamente, el mensaje es necesariamente transformado en algo que contiene sólo caracteres ASCII. Se utiliza las técnicas de codificación de transferencia de contenido RFC de correo electrónico estándar para codificar mensajes de correo electrónico para el transporte a través de canales que no son «8 bits limpios».

Para adaptar procesamiento reproducible de mensajes firmados por SMIME, `Generator` deshabilita el encabezamiento para las partes del mensaje de tipo `multipart/signed` y todas sus subpartes.

class `email.generator.BytesGenerator` (*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *, *policy=None*)

Retorna un objeto `BytesGenerator` que escribirá cualquier mensaje provisto por el método `flatten()`, o cualquier texto de escape sustituto cifrado con el método `write()`, al *file-like object* *outfp*. *outfp* debe soportar un método “write” que acepte datos binarios.

Si *mangle_from_* opcional es `True`, se coloca un carácter “>” delante de cualquier línea del cuerpo que comience con la cadena exacta “«From «”, es decir, “From” seguido de un espacio al principio de una línea. *mangle_from_* vuelve de forma predeterminada al valor de la configuración *mangle_from_* de la *norma* (que es “True” para la norma `compat32` y “False” para todas las demás). *mangle_from_* está diseñado para su uso cuando los mensajes se almacenan en formato unix mbox (consulte `mailbox` y `WHY THE CONTENT-LENGTH FORMAT IS BAD`).

Si *maxheaderlen* no es `None`, se repliega las líneas de encabezado que son mayores que *maxheaderlen*, o si es 0, no se reenvuelve ningún encabezado. Si *manheaderlen* es `None` (predeterminado), se envuelven los encabezados y otras líneas de mensajes de acuerdo a los ajustes de *policy*.

Si la *norma* es especificada, se usa esa norma para controlar la generación de mensajes. Si la *norma* es `None` (predeterminado), se usa la norma asociada con el `Message` o el objeto `EmailMessage` pasado para `flatten` para controlar la generación del mensaje. Se puede ver `email.policy` para detalles de que controla la *norma*.

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Agregada la palabra clave *norma*.

¹ Esta instrucción supone que se utiliza la configuración adecuada para `unixfrom`, y que no hay ninguna configuración *policy* que llame a ajustes automáticos (por ejemplo, `refold_source` debe ser `none`, que es *no* es el valor predeterminado). Esto tampoco es 100% verdadero, ya que si el mensaje no se ajusta a los estándares RFC ocasionalmente la información sobre el texto original exacto se pierde durante la el análisis de recuperación de errores. Es un objetivo fijar estos últimos casos extremos cuando sea posible.

Distinto en la versión 3.6: El comportamiento predeterminado de los parámetros *mangle_from_* y *maxheaderlen* es para seguir la norma.

flatten (*msg*, *unixfrom=False*, *linesep=None*)

Imprime la representación textual de la estructura del objeto de mensaje originada en *msg* al archivo de salida especificado cuando se creó la instancia *BytesGenerator*.

Si el tipo *cte_type* de la opción *policy*:attr:"email.policy.Policy.cte_type" es "8bit" (valor predeterminado), se copia los encabezados en el mensaje analizado original que no se hayan modificado con ningún bytes a la salida con el conjunto de bits altos, reproducido como en el original, y se conserva el *Content-Transfer-Encoding* no ASCII de cualquier parte del cuerpo que los tenga. Si *cte_type* es 7bit, se convierte los bytes con el conjunto de bits altos según sea necesario utilizando un *Content-Transfer-Encoding* compatible con ASCII. Es decir, se transforma partes con *Content-Transfer-Encoding* no ASCII (*Content-Transfer-Encoding: 8bit*) en un conjunto de caracteres compatible con ASCII *Content-Transfer-Encoding*, y se codifica bytes inválidos RFC no ASCII en encabezados mediante el conjunto de caracteres MIME unknown-8bit, lo que los convierte en compatibles con RFC.

Si *unixfrom* es *True*, se imprime el delimitador de encabezado de sobre utilizado por el formato de buzón de correo Unix (consulta *mailbox*) antes del primero de los encabezados [RFC 5322](#) del objeto de mensaje raíz. Si el objeto raíz no tiene encabezado de sobre, se crea uno estándar. El valor predeterminado es *False*. Tener en cuenta que para las subpartes, nunca se imprime ningún encabezado de sobre.

Si *linesep* no es *None*, se usa como caracter separador entre todas las líneas del mensaje acoplado. Si *linesep* es *None* (predeterminado), se usa el valor especificado en la *norma*.

clone (*fp*)

Retorna un clon independiente de esta instancia de *BytesGenerator* con las mismas configuraciones exactas, y *fp* como el nuevo *outfp*.

write (*s*)

Codifica *s* usando el códec ASCII u el manipulador de error *escape* sustituto, y lo pasa al método *write* del *outfp* pasado al constructor de la clase *BytesGenerator*.

Como conveniencia, *EmailMessage* provee los métodos *as_bytes()* y *bytes(aMessage)* (también conocido como *__bytes__()*) que simplifican la generación de la representación serializada binaria de un objeto mensaje. Para más detalle, ver *email.message*.

Dado que las cadenas de caracteres no pueden representar datos binarios, la clase *Generator* debe convertir los datos binarios en cualquier mensaje que aplane a un formato compatible con ASCII, convirtiéndolos en un *Content-Transfer-Encoding* compatible con ASCII. Usando la terminología de RFC de correo electrónico, se puede pensar en esto como *Generator* serializando a una secuencia de E/S que no es «8 bit clean». En otras palabras, la mayoría de las aplicaciones querrán usar *BytesGenerator*, y no *Generator*.

class email.generator.**Generator** (*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *, *policy=None*)

Retorna un objeto *Generator* que escribirá cualquier mensaje provisto al método *flatten()*, o cualquier texto provisto al método *write()*, al *file-like object* *outfp*. *outfp* debe soportar un método *write* que acepte datos de cadena de caracteres.

Si *mangle_from_* opcional es *True*, se coloca un carácter ">" delante de cualquier línea del cuerpo que comience con la cadena exacta "<<From <", es decir, "<<From" seguido de un espacio al principio de una línea. *mangle_from_* vuelve de forma predeterminada al valor de la configuración *mangle_from_* de la *norma* (que es "True" para la norma *compat32* y "False" para todas las demás). *mangle_from_* está diseñado para su uso cuando los mensajes se almacenan en formato unix mbox (consulte *mailbox* y [WHY THE CONTENT-LENGTH FORMAT IS BAD](#)).

Si *maxheaderlen* no es *None*, se repliega las líneas de encabezado que son mayores que *maxheaderlen*, o si es 0, no se reenvuelve ningún encabezado. Si *manheaderlen* es *None* (predeterminado), se envuelven los encabezados y otras líneas de mensajes de acuerdo a los ajustes de *policy*.

Si la *norma* es especificada, se usa esa norma para controlar la generación de mensajes. Si la *norma* es *None* (predeterminado), se usa la norma asociada con el *Message* o el objeto *EmailMessage* pasado para

`flatten` para controlar la generación del mensaje. Se puede ver `email.policy` para detalles de que controla la *norma*.

Distinto en la versión 3.3: Agregada la palabra clave *norma*.

Distinto en la versión 3.6: El comportamiento predeterminado de los parámetros *mangle_from_* y *maxheader-len* es para seguir la norma.

flatten (*msg*, *unixfrom=False*, *linesep=None*)

Imprime la representación textual de la estructura del objeto de mensaje originado en el *msg* al archivo especificado de salida cuando la instancia de `Generator` fue creada.

Si la opción `policy.cte_type` es 8bit, se genera el mensaje como si la opción estuviera establecida en 7bit. (Esto es necesario porque las cadenas de caracteres no pueden representar bytes que no sean ASCII). Convierte cualesquiera bytes con el conjunto de bits alto según sea necesario utilizando un *Content-Transfer-Encoding* compatible con ASCII. Es decir, transforma partes con *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) no ASCII en un conjunto de caracteres *Content-Transfer-Encoding* compatible con ASCII. Y codifica bytes no ASCII RFC inválidos ASCII en encabezados mediante el conjunto de caracteres MIME unknown-8bit, lo que los convierte en compatibles con RFC.

Si *unixfrom* es `True`, se imprime el delimitador de encabezado de sobre utilizado por el formato de buzón de correo Unix (consulta `mailbox`) antes del primero de los encabezados **RFC 5322** del objeto de mensaje raíz. Si el objeto raíz no tiene encabezado de sobre, se crea uno estándar. El valor predeterminado es `False`. Tener en cuenta que para las subpartes, nunca se imprime ningún encabezado de sobre.

Si *linesep* no es `None`, se usa como caracter separador entre todas las líneas del mensaje acoplado. Si *linesep* es `None` (predeterminado), se usa el valor especificado en la *norma*.

Distinto en la versión 3.2: Agrega soporte para el recodificado de cuerpos de mensajes 8bit, y el argumento *linesep*.

clone (*fp*)

Retorna un clon independiente de esta instancia de `Generator` con las mismas opciones exactas y *fp* como la nueva *outfp*.

write (*s*)

Escribe *s* al método `write` del *outfp* pasado al constructor de la `Generator`. Esto provee justo la suficiente API de tipo archivo para instancias de `Generator` para ser usadas en la función `print()`.

Para conveniencia, `EmailMessage` provee los métodos `as_string()` y `str(aMessage)` (también conocido como `__str__()`), que simplifican la generación de una representación de una cadena de caracteres formateada de un objeto mensaje. Para más detalles, ver `email.message`.

El módulo `email.generator` también provee una clase derivada, `DecodedGenerator`, la cual es como la clase base `Generator`, excepto que las partes no *text* no están serializadas, sino que en su lugar, están representadas en el flujo de salida por una cadena de caracteres derivada de una plantilla llenada con información sobre la parte.

class `email.generator.DecodedGenerator` (*outfp*, *mangle_from_=None*, *maxheader-len=None*, *fmt=None*, *, *policy=None*)

Se actúa como `Generator`, excepto para cualquier subparte del mensaje pasado a `Generator.flatten()`, si la subparte es de tipo principal *text*, se imprime la carga útil decodificada de la subparte, y si el tipo principal no es *text*, en lugar de imprimirlo se rellena la cadena de caracteres *fmt* utilizando la información de la parte y se imprime la cadena de caracteres rellena resultante.

Para llenar *fmt*, se ejecuta `fmt % part_info`, donde `part_info` es un diccionario compuesto por las siguientes claves y valores:

- *type* – Todo el tipo MIME de la parte no *text*
- *maintype* – Principal tipo MIME de la parte no *text*
- *subtype* – Tipo sub-MIME de la parte no-*text*
- *filename* – Nombre de archivo de la parte no *text*

- `description` – Descripción asociada con la parte `notext`
- `encoding` – Codificado de la transferencia del contenido de la parte `no-text`

Si `fmt` es `None`, se usa el siguiente `fmt` predeterminado:

«[Non-text `%(type)s`] part of message omitted, filename `%(filename)s`]»

Los opcionales `_mangle_from_` y `maxheaderlen` son como en la clase base `Generator`.

Notas al pie

19.1.4 `email.policy`: Objetos *Policy*

Nuevo en la versión 3.3.

Código fuente: `Lib/email/policy.py`

El enfoque principal del paquete `email` es la gestión de mensajes de correo electrónico como se describe por los varios RFC de correos electrónicos y MIME. Sin embargo, el formato general de los mensajes de correo electrónico (un bloque de campos de cabecera cada uno consistiendo en un nombre seguido de dos puntos seguido de un valor, el bloque entero seguido por una línea blanca y un “cuerpo” arbitrario), es un formato que ha encontrado utilidad fuera del campo de correos electrónicos. Algunos de estos usos cumplen bastante cerca los RFC de correos electrónicos principales, alguno no. Incluso cuando se trabaja con correos electrónicos, hay veces cuando es deseable romper la estricta conformidad con los RFC, tal como generar correos electrónicos que se integran con servidores de correos electrónicos que no siguen los estándares, o que implementan extensiones que quieres usar en formas que violan los estándares.

Los objetos *policy* dan al paquete de correos electrónicos la flexibilidad de manejar todos estos casos de uso diversos.

Un objeto *Policy* encapsula un conjunto de atributos y métodos que controlan el comportamiento de varios componentes del paquete de correos electrónicos durante el uso. Las instancias *Policy* pueden ser pasadas a varias clases y métodos en el paquete de correos electrónicos para alterar el comportamiento por defecto. Los valores que se pueden configurar y sus valores por defecto se describen abajo.

Hay una *policy* por defecto usada por todas las clases en el paquete de correos electrónicos. Para todas las clases `Parser` y las funciones de conveniencia relacionadas, y para la clase `Message`, esta es una *policy* `Compat32`, a través de sus correspondientes instancias `compat32` pre-definidas. Esta política mantiene una compatibilidad (en algunos casos, compatibilidad con los bugs) con el paquete de correos electrónicos de las versiones anteriores de Python3.3.

El valor por defecto para la palabra clave *policy* de `EmailMessage` es la *policy* `EmailPolicy`, a través de su instancia pre-definida `default`.

Cuando un objeto `Message` o `EmailMessage` es creado, adquiere un *policy*. Si el mensaje es creado por un `parser`, un *policy* pasado al analizador será el *policy* usado por el mensaje que cree. Si el mensaje es creado por el programa, entonces el *policy* puede ser especificado cuando sea creado. Cuando un mensaje es pasado a un `generator`, el generador usa el *policy* del mensaje por defecto, pero también puedes pasar un *policy* específico al generador que anulará el que está guardado en el objeto mensaje.

El valor por defecto del argumento por palabra clave *policy* para las clases `email.parser` y las funciones de conveniencia del analizador **se cambiarán** en una futura versión de Python. Por consiguiente, **siempre debes especificar explícitamente qué *policy* quieres usar** cuando se llama a cualquiera de las clases y funciones descritas en el módulo `parser`.

La primera parte de esta documentación cubre las características de *Policy*, un *abstract base class* que define las características que son comunes a todos los objetos *policy*, incluyendo `compat32`. Esto incluye ciertos métodos gancho (*hook*) que son llamados internamente por el paquete de correos electrónicos, que un *policy* personalizado puede invalidar para obtener un comportamiento diferente. La segunda parte describe las clases concretas `EmailPolicy` y `Compat32`, que implementan los ganchos (*hooks*) que proporcionan el comportamiento estándar y el comportamiento y características compatibles hacia atrás, respectivamente.

Las instancias `Policy` son inmutables, pero pueden ser clonadas, aceptando los mismos argumentos de palabra clave que el constructor de clase y retorna una nueva instancia `Policy` que es una copia del original pero con los valores de atributos específicos cambiados.

Como un ejemplo, el siguiente código puede ser usado para leer un mensaje de correo electrónico de un archivo en disco y pasarlo al programa de sistema `sendmail` en un sistema Unix:

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f, policy=policy.default)
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r\n'))
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

Aquí le estamos diciendo a `BytesGenerator` que use los caracteres de separación de línea correctos de RFC cuando crea la cadena binaria para alimentar a `stding` de `sendmail` (`sendmail`'s `stdin`), donde el `policy` por defecto usaría separadores de línea `\n`.

Algunos métodos del paquete de correos electrónicos aceptan un argumento de palabra clave `policy`, permitiendo que el `policy` pueda ser anulado para ese método. Por ejemplo, el siguiente código usa el método `as_bytes()` del objeto `msg` del ejemplo anterior y escribe el mensaje en un archivo usando los separadores de línea nativos para la plataforma en el que esté corriendo:

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep)))
17
```

Los objetos `policy` puede ser combinados usando el operador de adición, produciendo un objeto `policy` cuya configuración es una combinación de los valores que de los objetos sumados:

```
>>> compat SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict SMTP = compat SMTP + compat_strict
```

Esta operación no es conmutativa; es decir, el orden en el que los objetos son añadidos importa. Para ilustrar:

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class `email.policy.Policy` (****kw**)

Este es el *abstract base class* para todas las clases `policy`. Proporciona las implementaciones por defecto para un par de métodos triviales, también como la implementación de las propiedades de inmutabilidad, el método `clone()`, y las semánticas del constructor.

Se pueden pasar varios argumentos de palabra clave al constructor de una clase `policy`. Los argumentos que pueden ser especificados son cualquier propiedad que no sea método en esta clase, además de cualquier otra propiedad adicional que no sea método en la clase concreta. Un valor especificado en el constructor anulará el valor por defecto para los atributos correspondientes.

Esta clase define las siguientes propiedades, y por consiguiente los valores a continuación pueden ser pasados al constructor de cualquier clase `policy`:

max_line_length

El tamaño máximo de cualquier línea en la salida serializada, sin contar el fin de la línea de carácter(res). Por defecto es 78, por el [RFC 5322](#). Un valor de 0 o *None* indica que ningún envolvimiento de líneas puede ser hecha en lo más mínimo.

linesep

La cadena de caracteres a ser usada para terminar las líneas en una salida serializada. El valor por defecto es `\n` porque esa es la disciplina del fin de línea interna usada por Python, aunque `\r\n` es requerida por los RFCs.

cte_type

Controla el tipo de Codificaciones de Transferencia de Contenido que pueden ser o son necesarios para ser usados. Los valores posibles son:

7bit	todos los datos deben ser «compatibles con 7 bit (<i>7 bit clean</i>)» (ASCII-only). Esto significa que donde sea necesario, los datos serán codificados usando o codificación imprimible entre comillas o base64.
8bit	los datos no son limitados a ser compatibles con 7 bits (<i>7 bit clean</i>). Se requiere que los datos en las cabeceras todavía sean sólo ASCII y por lo tanto estarán codificadas (véase <code>fold_binary()</code> y <code>utf8</code> abajo por las excepciones), pero las partes del cuerpo pueden usar 8bit CTE.

Un valor `cte_type` de `8bit` sólo trabaja con `BytesGenerator`, no `Generator`, porque las cadenas no pueden contener datos binarios. Si un `Generator` está operando bajo un *policy* que especifica `cte_type=8bit`, actuará como si `cte_type` fuese `7bit`.

raise_on_defect

Si es *True*, cualquier defecto encontrado será lanzado como error. Si es *False* (el valor por defecto), los defectos serán pasados al método `register_defect()`.

mangle_from_

Si es *True*, las líneas empezando con «*From*» en el cuerpo son saltados al poner un `>` en frente de ellos. Este parámetro es usado cuando el mensaje está siendo serializado por un generador. El valor por defecto es: *False*.

Nuevo en la versión 3.5: El parámetro `mangle_from_`.

message_factory

Una función de fábrica para construir un nuevo objeto mensaje vacío. Usado por el analizador cuando construye mensajes. Por defecto es *None*, en cuyo caso `Message` es usado.

Nuevo en la versión 3.6.

verify_generated_headers

If *True* (the default), the generator will raise `HeaderWriteError` instead of writing a header that is improperly folded or delimited, such that it would be parsed as multiple headers or joined with adjacent data. Such headers can be generated by custom header classes or bugs in the `email` module.

As it's a security feature, this defaults to *True* even in the `Compat32` policy. For backwards compatible, but unsafe, behavior, it must be set to *False* explicitly.

Nuevo en la versión 3.8.20.

El siguiente método de *Policy* está destinado a ser llamado por código usando la librería de correos electrónicos para crear instancias de *policy* con configuraciones personalizadas:

clone (kw)**

Retorna una nueva instancia de *Policy* cuyos atributos tienen los mismos valores que la instancia actual, excepto donde se le dan, a esos atributos, nuevos valores por los argumentos de palabra clave.

Los métodos de *Policy* restantes son llamados por el código de paquete de correos electrónicos, y no están destinados a ser llamados por una aplicación usando el paquete de correos electrónicos. Un *policy* personalizado debe implementar todos estos métodos.

handle_defect (*obj*, *defect*)

Trata un *defect* encontrado en *obj*. Cuando el paquete de correos electrónicos llama a este método, *defect* siempre será una subclase de `Defect`.

La implementación por defecto verifica la bandera `raise_on_defect`. Si es `True`, *defect* es lanzado como una excepción. Si es `False` (el valor por defecto), *obj* y *defect* son pasados a `register_defect()`.

register_defect (*obj*, *defect*)

Registra un *defect* en *obj*. En el paquete de correos electrónicos, *defect* será siempre una subclase de `Defect`.

La implementación por defecto llama al método `append` del atributo `defects` de *obj*. Cuando un paquete de correos electrónicos llama a `handle_defect`, *obj* normalmente tendrá un atributo `defects` que tiene un método `append`. Los tipos de objetos personalizados usados con el paquete de correos electrónicos (por ejemplo, objetos `Message` personalizados) también deben proporcionar tal atributo, de otro modo, los defectos en los mensajes analizados levantarán errores no esperados.

header_max_count (*name*)

Retorna el máximo número permitido de cabeceras llamadas *name*.

Llamado cuando una cabecera es añadida a un objeto `EmailMessage` o `Message`. Si el valor retornado no es `0` o `None`, y ya hay un número de cabeceras con el nombre *name* mayor o igual que el valor retornado, un `ValueError` es lanzado.

Porque el comportamiento por defecto de `Message.__setitem__` es agregar el valor a la lista de cabeceras, es fácil crear cabeceras duplicadas sin darse cuenta. Este método permite que ciertas cabeceras sean limitadas en el número de instancias de esa cabecera que puede ser agregada a `Message` programáticamente. (El límite no es observado por el analizador, que fielmente producirá tantas cabeceras como existen en el mensaje siendo analizado.)

La implementación por defecto retorna `None` para todos los nombres de cabeceras.

header_source_parse (*sourcelines*)

El paquete de correos electrónicos llama a este método con una lista de cadenas de caracteres, cada terminación de cadena con los caracteres de separación de línea encontrada en la fuente siendo analizada. La primera línea incluye el campo del nombre de cabecera y el separador. Todos los espacios en blanco en la fuente son preservados. El método debe retornar la tupla (*name*, *value*) que va a ser guardada en el `Message` para representar la cabecera analizada.

Si una implementación desea mantener compatibilidad con los *policy* del paquete de correos electrónicos existentes, *name* debe ser el nombre con las letras preservadas (todos los caracteres hasta el separador “:”), mientras que *value* debe ser el valor desdoblado (todos los caracteres de separación de líneas eliminados, pero los espacios en blanco intactos), privado de espacios en blanco al comienzo.

sourcelines puede contener datos binarios *surrogateescaped*.

No hay implementación por defecto

header_store_parse (*name*, *value*)

El paquete de correos electrónicos llama a este método con el nombre y valor proporcionados por el programa de aplicación cuando la aplicación está modificando un `Message` programáticamente (en lugar de un `Message` creado por un analizador). El método debe retornar la tupla (*name*, *value*) que va a ser almacenada en el `Message` para representar la cabecera.

Si una implementación desea retener compatibilidad con los *policy* del paquete de correos electrónicos existentes, el *name* y *value* deben ser cadenas de caracteres o subclases de cadenas que no cambien el contenido de los pasados en los argumentos.

No hay implementación por defecto

header_fetch_parse (*name*, *value*)

El paquete de correos electrónicos llama a este método con el *name* y *valor* actualmente guardados en `Message` (el mensaje) cuando la cabecera es solicitada por el programa de aplicación, y lo que sea que el método retorne es lo que es pasado de vuelta a la aplicación como el valor de la cabecera siendo

recuperado. Note que puede haber más de una cabecera con el mismo nombre guardado en `Message`; el método es pasado al nombre específico y valor de la cabecera destinado a ser retornado a la aplicación.

value puede contener datos binarios *surrogateescaped*. No debe haber datos binarios *surrogateescaped* en el valor retornado por el método.

No hay implementación por defecto

fold (*name*, *value*)

El paquete de correos electrónicos llama a este método con los *name* y *value* actualmente guardados en `Message` para una cabecera dada. El método debe retornar una cadena de caracteres que represente esa cabecera «doblada» correctamente (de acuerdo a los ajustes del *policy*) al componer *name* con *value* e insertar caracteres *lineseq* en los lugares apropiados. Véase [RFC 5322](#) para una discusión de las reglas para doblar cabeceras de correos electrónicos.

value puede contener datos binarios *surrogateescaped*. No debe haber datos binarios *surrogateescaped* en la cadena de caracteres retornada por el método.

fold_binary (*name*, *value*)

Igual que `fold()`, excepto que el valor retornado debe ser un objeto bytes en vez de una cadena de caracteres.

value puede contener datos binarios *surrogateescaped*. Estos pueden ser convertidos de vuelta a datos binarios en el objeto de bytes retornado.

class email.policy.**EmailPolicy** (**kw)

Este *Policy* concreto proporciona el comportamiento que sirve para cumplir con los RFCs actuales para correos electrónicos. Estos incluyen (pero no están limitados a) [RFC 5322](#), [RFC 2047](#), y los actuales RFCs MIME.

Esta *policy* incorpora nuevos analizadores de cabeceras y algoritmos de doblado. En vez de cadenas de caracteres simples, las cabeceras son subclases de `str` con atributos que dependen del tipo del campo. El analizado y algoritmo de doblado implementan los [RFC 2047](#) y [RFC 5322](#) por completo.

El valor por defecto para el atributo *message_factory* es `EmailMessage`.

Además de los atributos que se pueden configurar listados arriba que aplican a todas las *policies*, este *policy* añade los siguientes atributos adicionales:

Nuevo en la versión 3.6:¹

utf8

Si es `False`, se sigue el [RFC 5322](#), siendo compatible con caracteres non-ASCII en las cabeceras al codificarlas como «palabras codificadas». Si es `True`, sigue el [RFC 6532](#) y usa el formato de codificación `utf-8` para las cabeceras. Los mensajes formateados de esta manera puede ser pasados a servidores SMTP que admitan la extensión SMTPUTF8 ([RFC 6531](#)).

refold_source

Si el valor para una cabecera en el objeto `Message` se originó de un *parser* (en lugar de ser establecido por el programa), este atributo indica tanto si un generador debe redoblar ese valor cuando se transforma al mensaje de vuelta a la forma serializada o no. Los valores posibles son:

<code>none</code>	todos los valores de la fuente usan el doblamiento original
<code>long</code>	los valores de la fuente que tengan una línea que sea más grande que <code>max_line_length</code> serán redoblados
<code>all</code>	todos los valores son redoblados.

El valor por defecto es `long`.

header_factory

Un invocable que toma dos argumentos, *name* y *value*, donde *name* es un nombre de campo de cabecera y *value* es un valor del campo de la cabecera no doblado, y retorna una subclase de cadenas de caracteres que representan la cabecera. Un valor por defecto *header_factory* (véase

¹ Se añadió originalmente en 3.3 como un *característica provisional*.

`headerregistry`) es proporcionado que admite el análisis personalizado para los varios tipos de cabecera [RFC 5322](#) de direcciones y fechas, y los tipos del cabeceras MIME principales. La compatibilidad de analizadores personalizados adicionales será agregada en el futuro.

content_manager

Un objeto con al menos dos métodos: `get_content` and `set_content`. Cuando los métodos `get_content()` o `set_content()` del objeto `message.EmailMessage` son llamados, llama al método correspondiente de este objeto, pasándole el mensaje objeto como su primer argumento, y cualquier argumento o palabra clave que fuese pasado como un argumento adicional. Por defecto `content_manager` se pone a `raw_data_manager`.

Nuevo en la versión 3.4.

La clase proporciona las siguientes implementaciones concretas de los métodos abstractos de `Policy`:

header_max_count (*name*)

Retorna el valor del atributo `max_count` de la clase especializada usada para representar la cabecera con el nombre dado.

header_source_parse (*sourcelines*)

El nombre es analizado como todo hasta el ":" y retornado inalterado. El valor es determinado al remover los espacios en blanco al principio del resto de la primera línea, juntando todas las subsecuentes líneas, y removiendo cualquier carácter CR o LF.

header_store_parse (*name, value*)

El nombre es retornado inalterado. Si el valor de la entrada tiene un atributo `name` y coincide con `name` ignorando mayúsculas y minúsculas, el valor es retornado inalterado. Si no, el `name` y `value` son pasados a `header_factory`, y el objeto cabecera resultante es retornado como el valor. En este caso un `ValueError` es lanzado si el valor de la entrada contiene caracteres CR o LF.

header_fetch_parse (*name, value*)

Si el valor tiene un atributo `name`, es retornado inalterado. De otra manera el `name`, y el `value` con cualquier carácter CR o LF eliminado, son pasados a la `header_factory`, y el objeto cabecera resultante es retornado. Cualquier byte `surrogateescaped` es convertido al glifo de carácter desconocido de unicode.

fold (*name, value*)

El doblamiento de la cabecera es controlado por la configuración de `policy.refold_source`. Se considera que un valor es `source value` (valor fuente) si y sólo si no tiene un atributo `name` ``` (tener un atributo ```name` significa que es un objeto cabecera de algún tipo). Si un valor fuente necesita ser redoblado de acuerdo a la política, es convertido en un objeto cabecera al pasarle el `name` y `value` con cualquier carácter CR y LF eliminado al `header_factory`. El doblamiento de un objeto cabecera es hecho al llamar a su método `fold` con el `policy` actual.

Los valores fuente son separadas en líneas usando el método `splitlines()`. Si el valor no va a ser redoblado, las líneas son unidas de nuevo usando el `linesep` del `policy` y retornadas. La excepción son las líneas que contienen datos binarios non-ascii. En ese caso el valor es redoblado a pesar de la configuración de `refold_source`, que causa que los datos binarios sean codificados CTE usando el juego de caracteres (charset) `unknown-8bit`.

fold_binary (*name, value*)

Igual que `fold()` si `cte_type` fuese 7bit, excepto que el valor retornado son bytes.

Si `cte_type` es 8bit, los datos binarios non-ASCII son convertidos de vuelta a bytes. Las cabeceras con datos binarios son redoblados, sin considerar la configuración de `refold_header`, ya que no hay forma de saber si los datos binarios consisten en caracteres de un sólo byte o caracteres con múltiples bytes.

Las siguientes instancias de `EmailPolicy` proporcionan valores por defecto apropiados para dominios de aplicaciones específicas. Note que en el futuro el comportamiento de estas instancias (en particular la instancia `HTTP`) puede ser ajustado para cumplir incluso más de cerca a los RFC relevantes a sus dominios.

email.policy.default

Una instancia de `EmailPolicy` con todos los valores por defecto sin cambiar. Este `policy` usa la terminación de línea `\n` estándar de Python en vez de correcto por el RFC `\r\n`.

email.policy.SMTP

Apropiado para la serialización de mensajes en cumplimiento con los RFC de correos electrónicos. Como `default`, pero con `linesep` puesto en `\r\n`, que es conforme con el RFC.

email.policy.SMTPUTF8

Igual que SMTP excepto que `utf8` es `True`. Útil para serializar mensajes a un almacén de mensajes sin usar palabras codificadas en las cabeceras. Sólo debe ser utilizada para transmisiones por SMTP si las direcciones del remitente o recipiente tienen caracteres non-ASCII (el método `smtplib.SMTP.send_message()` gestiona esto automáticamente).

email.policy.HTTP

Apropiado para serializar cabeceras con uso en tráfico de HTTP. Como SMTP excepto que `max_line_length` es puesto en `None` (ilimitado).

email.policy.strict

Instancias de conveniencia. Igual que `default` excepto que `raise_of_defect` es puesto en `True`. Esto permite que cualquier *policy* sea hecho estricto al escribir:

```
somepolicy + policy.strict
```

Con todos estos *EmailPolicies*, el API efectivo del paquete de correos electrónicos es cambiado del API de Python 3.2 en las siguientes maneras:

- Estableciendo una cabecera en una *Message* resulta en que la cabecera sea analizada y un objeto cabecera sea creado.
- Buscar una valor de cabecera de un *Message* resulta en que la cabecera sea analizada y un objeto cabecera sea creado y retornado.
- Cualquier objeto cabecera, o cualquier cabecera que sea redoblada debido a las configuraciones del *policy*, es doblado usando un algoritmo que implementa el algoritmo de doblado del RFC por completo, incluyendo saber dónde las palabras codificadas son requeridas y permitidas.

Desde la vista de la aplicación, significa que cualquier cabecera obtenida a través de *EmailMessage* es un objeto cabecera con atributos extra, cuyo valor de cadena es el valor Unicode de la cabecera completamente decodificada. Asimismo, se le puede asignar un nuevo valor a una cabecera, o a una nueva cabecera creada, usando una cadena de caracteres Unicode, y el *policy* se ocupará de convertir la cadena Unicode en la forma decodificada correcta según el RFC.

Los objetos cabecera y sus atributos son descritos en `headerregistry`.

class email.policy.Compat32 (kw)**

Este *Policy* concreto es el *policy* compatible hacia atrás. Replica el comportamiento del paquete de correos electrónicos en Python 3.2. El módulo *policy* también define una instancia de esta clase, *compat32*, que es usado como el *policy* por defecto. Por consiguiente, el comportamiento por defecto del paquete de correos electrónicos es mantener compatibilidad con Python 3.2.

Los siguientes atributos tienen valores que son diferentes del *Policy* por defecto:

mangle_from_

El valor por defecto es `True`.

La clase proporciona las siguientes implementaciones concretas de los métodos abstractos de *Policy*:

header_source_parse (sourcelines)

El nombre es analizado como todo hasta el “:” y retornado inalterado. El valor es determinado al remover los espacios en blanco al principio del resto de la primera línea, juntando todas las subsecuentes líneas, y removiendo cualquier carácter CR o LF.

header_store_parse (name, value)

El nombre y valor son retornados inalterados.

header_fetch_parse (name, value)

Si el valor contiene datos binarios, es convertido a un objeto *Header* usando el juego de caracteres (*charset*) `unknown-8bit`. De otro modo, es retornado sin modificar.

fold (*name*, *value*)

Las cabeceras son dobladas usando el algoritmo de doblado de *Header*, lo que preserva los saltos de líneas existentes en el valor, y envuelve cada línea resultante hasta el `max_line_length`. Los datos binarios Non-ASCII son codificados por *CTE* usando el juego de caracteres (*charset*) `unknown-8bit`.

fold_binary (*name*, *value*)

Las cabeceras son dobladas usando el algoritmo de doblado *Header*, lo que preserva los saltos de línea existentes en el valor, y envuelve cada línea resultante hasta `max_line_length`. Si `cte_type` es `7bit`, los datos binarios non-ascii son codificados por *CTE* usando el juego de caracteres `unknown-8bit`. De otro modo, la cabecera fuente original es usada, con sus saltos de línea existentes y cualquier (inválido según el RFC) dato binario que puede contener.

`email.policy.compat32`

Una instancia de *Compat32*, proporcionando compatibilidad hacia atrás con el comportamiento del paquete de correos electrónicos en Python 3.2.

Notas al pie de página

19.1.5 email.errors: Clases de excepción y defecto

Código fuente: [Lib/email/errors.py](#)

Las siguientes clases de excepción se definen en el módulo `email.errors`:

exception `email.errors.MessageError`

Esta es la clase base para todas las excepciones que el paquete `email` puede lanzar. Se deriva de la clase estándar *Exception* y no define métodos adicionales.

exception `email.errors.MessageParseError`

Esta es la clase base para las excepciones generadas por la clase *Parser*. Se deriva de *MessageError*. Esta clase también es utilizada internamente por el analizador sintáctico utilizado por *headerregistry*.

exception `email.errors.HeaderParseError`

Lanzada en algunas condiciones de error al analizar los encabezados **RFC 5322** de un mensaje, esta clase se deriva de *MessageParseError*. El método `set_boundary()` lanzará este error si el tipo de contenido es desconocido cuando se llama al método. *Header* puede lanzar este error para ciertos errores de decodificación de base64, y cuando se intenta crear un encabezado que parece contener un encabezado incrustado (es decir, hay lo que se supone que es un línea de continuación que no tiene espacios en blanco iniciales y parece un encabezado).

exception `email.errors.BoundaryError`

Deprecada y no utilizada actualmente.

exception `email.errors.MultipartConversionError`

Se lanza cuando se agrega una carga útil (*payload*) a un objeto *Message* usando `add_payload()`, pero la carga útil ya es un número escalar y el tipo principal del mensaje *Content-Type* no es *multipart* ni perdido (*missing*). *MultipartConversionError* hereda al mismo tiempo de *MessageError* y el incorporado *TypeError*.

Dado que `Message.add_payload()` está en desuso, esta excepción rara vez se presenta en la práctica. Sin embargo, la excepción también se puede lanzar si se llama al método `attach()` en una instancia de una clase derivada de *MIMENonMultipart* (por ejemplo *MIMEImage*).

exception `email.errors.HeaderWriteError`

Raised when an error occurs when the *generator* outputs headers.

Aquí está la lista de defectos que *FeedParser* puede encontrar mientras analiza mensajes. Tenga en cuenta que los defectos se agregan al mensaje donde se encontró el problema, por ejemplo, si un mensaje anidado dentro de un *multipart/alternative* tenía un encabezado mal formado, ese objeto de mensaje anidado tendría un defecto, pero el contenido los mensajes no lo harían.

Todas las clases de defectos se derivan de `email.errors.MessageDefect`.

- `NoBoundaryInMultipartDefect` – Un mensaje que se dice que es multiparte, pero que no tiene el parámetro *boundary*.
- `StartBoundaryNotFoundDefect` – El límite de inicio reclamado en el encabezado *Content-Type* nunca se encontró.
- `CloseBoundaryNotFoundDefect` – Se encontró un límite de inicio, pero nunca se encontró un límite cercano correspondiente.

Nuevo en la versión 3.3.

- `FirstHeaderLineIsContinuationDefect` – El mensaje tenía una línea de continuación como primera línea de encabezado.
- `MisplacedEnvelopeHeaderDefect` – Se encontró un encabezado «*Unix From*» en medio de un bloque de encabezado.
- `MissingHeaderBodySeparatorDefect` – Se encontró una línea al analizar sintácticamente los encabezados que no tenían espacios en blanco iniciales pero que no contenían “:”. El análisis continúa asumiendo que la línea representa la primera línea del cuerpo.

Nuevo en la versión 3.3.

- `MalformedHeaderDefect` – Se encontró un encabezado al que le faltaban dos puntos o tenía un formato incorrecto.

Obsoleto desde la versión 3.3: Este defecto no se ha utilizado por varias versiones de Python.

- `MultipartInvariantViolationDefect` – Un mensaje que se afirma ser *multipart*, pero no se encontraron subpartes. Tenga en cuenta que cuando un mensaje tiene este defecto, su método `is_multipart()` puede retornar `False` aunque su tipo de contenido afirma ser *multipart*.
- `InvalidBase64PaddingDefect` – Al decodificar un bloque de bytes codificados en *base64*, el relleno no era correcto. Se agrega suficiente relleno (*padding*) para realizar la decodificación, pero los bytes decodificados resultantes pueden no ser válidos.
- `InvalidBase64CharactersDefect` – Al decodificar un bloque de bytes codificados en *base64*, se encontraron caracteres fuera del alfabeto *base64*. Los caracteres se ignoran, pero los bytes decodificados resultantes pueden no ser válidos.
- `InvalidBase64LengthDefect` – Al decodificar un bloque de bytes codificados en *base64*, el número de caracteres *base64* sin relleno no era válido (1 más que un múltiplo de 4). El bloque codificado se mantuvo tal cual.

19.1.6 `email.headerregistry`: Objetos de encabezado personalizados

Código fuente: `Lib/email/headerregistry.py`

Nuevo en la versión 3.6:¹

Los encabezados están representados por subclases personalizadas de `str`. La clase particular utilizada para representar un encabezado dado está determinada por `header_factory` del `policy` vigente cuando se crean los encabezados. Esta sección documenta el `header_factory` particular implementado por el paquete de correo electrónico para el manejo mensajes de correo electrónico compatibles con **RFC 5322**, que no solo proporciona objetos de encabezado personalizados para varios tipos de encabezados, sino que también proporciona un mecanismo de extensión para que las aplicaciones agreguen sus propios tipos de encabezados personalizados.

Cuando se utiliza cualquiera de los objetos de política derivados de `EmailPolicy`, todos los encabezados son producidos por `HeaderRegistry` y tienen `BaseHeader` como su última clase base. Cada clase de encabezado tiene una clase base adicional que está determinada por el tipo de encabezado. Por ejemplo, muchos encabezados tienen la clase `UnstructuredHeader` como su otra clase base. La segunda clase especializada para un encabezado está determinada por el nombre del encabezado, utilizando una tabla de búsqueda almacenada en `HeaderRegistry`.

¹ Originalmente añadido en 3.3 como módulo provisional *provisional module*

Todo esto se gestiona de forma transparente para el programa de aplicación típico, pero se proporcionan interfaces para modificar el comportamiento predeterminado para su uso por aplicaciones más complejas.

Las secciones a continuación primero documentan las clases base de encabezados y sus atributos, seguidas por la API para modificar el comportamiento de *HeaderRegistry*, y finalmente las clases de soporte utilizadas para representar los datos analizados a partir de encabezados estructurados.

class email.headerregistry.**BaseHeader** (*name, value*)

name y *value* se pasan a *BaseHeader* desde la llamada *header_factory*. El valor de cadena de caracteres de cualquier objeto de encabezado es el *value* completamente descodificado en unicode.

Esta clase base define las siguientes propiedades de solo lectura:

name

El nombre del encabezado (la parte del campo antes del “:”). Este es exactamente el valor pasado en *header_factory* llamada para *name*; es decir, se conserva el caso.

defects

Una tupla de instancias *HeaderDefect* que informan sobre cualquier problema de cumplimiento de RFC que se encuentre durante el análisis. El paquete de correo electrónico intenta estar completo para detectar problemas de cumplimiento. Vea el módulo *errors* para una discusión de los tipos de defectos que pueden ser reportados.

max_count

El número máximo de encabezados de este tipo que pueden tener el mismo *name*. Un valor de *None* significa ilimitado. El valor de *BaseHeader* para este atributo es *None*; se espera que las clases de encabezado especializadas anulen este valor según sea necesario.

BaseHeader también proporciona el siguiente método, que es llamado por el código de la biblioteca de correo electrónico y, en general, no debe ser llamado por programas de aplicación:

fold (*, *policy*)

Retorna una cadena que contenga *linesep* caracteres según sea necesario para doblar correctamente el encabezado de acuerdo con *policy*. Un atributo *cte_type* de ‘8bit’ se tratará como si fuera “7bit”, ya que los encabezados no pueden contener datos binarios arbitrarios. Si *utf8* es *False*, los datos no ASCII estarán codificados **RFC 2047**.

BaseHeader por sí solo no se puede utilizar para crear un objeto de encabezado. Define un protocolo con el que coopera cada encabezado especializado para producir el objeto de encabezado. Específicamente, *BaseHeader* requiere que la clase especializada proporcione un *classmethod()* llamado *parse*. Este método se llama de la siguiente manera:

```
parse(string, kwds)
```

kwds es un diccionario que contiene una clave preinicializada, *defects*. *defects* es una lista vacía. El método de análisis debe agregar cualquier defecto detectado a esta lista. A la devolución, el diccionario *kwds* debe (*must*) contener valores para al menos las claves *decoded`* y *defects.decoded* debe ser el valor de cadena para el encabezado (es decir, el valor del encabezado completamente decodificado a Unicode). El método de análisis debe asumir que *string* puede contener partes codificadas por transferencia de contenido, pero también debe manejar correctamente todos los caracteres Unicode válidos para que pueda analizar valores de encabezado no codificados.

Entonces, el *__new__* de *BaseHeader* crea la instancia del encabezado y llama a su método *init*. La clase especializada solo necesita proporcionar un método *init* si desea establecer atributos adicionales más allá de los proporcionados por *BaseHeader*. Tal método *init* debería verse así:

```
def init(self, /, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

Es decir, cualquier cosa adicional que la clase especializada ponga en el diccionario *kwds* debe eliminarse y manejarse, y el contenido restante de *kw* (y *args*) debe pasar *BaseHeader* al método *init*.

class email.headerregistry.UnstructuredHeader

Un encabezado «sin estructura» es el tipo predeterminado de encabezado en [RFC 5322](#). Cualquier encabezado que no tenga una sintaxis especificada se trata como no estructurado. El ejemplo clásico de un encabezado no estructurado es el encabezado *Subject*.

En [RFC 5322](#), un encabezado no estructurado es una ejecución de texto arbitrario en el conjunto de caracteres ASCII. [RFC 2047](#), sin embargo, tiene un mecanismo compatible [RFC 5322](#) para codificar texto no ASCII como caracteres ASCII dentro de un valor de encabezado. Cuando un *value* que contiene palabras codificadas se pasa al constructor, el analizador `UnstructuredHeader` convierte dichas palabras codificadas en unicode, siguiendo las reglas [RFC 2047](#) para texto no estructurado. El analizador utiliza heurística para intentar decodificar ciertas palabras codificadas no compatibles. Los defectos se registran en tales casos, así como defectos por problemas como caracteres no válidos dentro de las palabras codificadas o el texto no codificado.

Este tipo de encabezado no proporciona atributos adicionales.

class email.headerregistry.DateHeader

[RFC 5322](#) especifica un formato muy específico para las fechas dentro de los encabezados de correo electrónico. El analizador `DateHeader` reconoce ese formato de fecha, además de reconocer una serie de formas variantes que a veces se encuentran «en la naturaleza» (*«in the wild»*).

Este tipo de encabezado proporciona los siguientes atributos adicionales:

datetime

Si el valor del encabezado puede reconocerse como una fecha válida de una forma u otra, este atributo contendrá una instancia `datetime` que representa esa fecha. Si la zona horaria de la fecha de entrada se especifica como `-0000` (lo que indica que está en UTC pero no contiene información sobre la zona horaria de origen), entonces `datetime` será un ingenuo `datetime`. Si se encuentra un desplazamiento de zona horaria específico (incluido `+0000`), entonces `datetime` contendrá un `datetime` consciente que usa `datetime.timezone` para registrar el desplazamiento de la zona horaria.

El valor decodificado del encabezado se determina formateando la `datetime` de acuerdo con las reglas [RFC 5322](#); es decir, esto se establece en:

```
email.utils.format_datetime(self.datetime)
```

Al crear un `DateHeader`, *value* puede ser `datetime` instancia. Esto significa, por ejemplo, que el siguiente código es válido y hace lo que cabría esperar:

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

Debido a que se trata de una `datetime` naif, se interpretará como una marca de tiempo UTC, y el valor resultante tendrá una zona horaria de `-0000`. Mucho más útil es usar la función `localtime()` del módulo `utils`:

```
msg['Date'] = utils.localtime()
```

Este ejemplo establece el encabezado de la fecha en la hora y fecha actuales utilizando el desplazamiento de zona horaria actual.

class email.headerregistry.AddressHeader

Los encabezados de dirección son uno de los tipos de encabezados estructurados más complejos. La clase `AddressHeader` proporciona una interfaz genérica para cualquier encabezado de dirección.

Este tipo de encabezado proporciona los siguientes atributos adicionales:

groups

Una tupla de objetos `Group` que codifican las direcciones y los grupos que se encuentran en el valor del encabezado. Las direcciones que no forman parte de un grupo se representan en esta lista como `Groups` de una sola dirección cuyo `display_name` es `None`.

addresses

Una tupla de objetos `Address` que codifican todas las direcciones individuales del valor del encabezado. Si el valor del encabezado contiene algún grupo, las direcciones individuales del grupo se incluyen en

la lista en el punto donde el grupo aparece en el valor (es decir, la lista de direcciones se «aplana» («*flattened*») en una lista unidimensional).

El valor `decoded` del encabezado tendrá todas las palabras codificadas decodificadas a Unicode. Los nombres de dominio codificados *idna* también se decodifican en Unicode. El valor `decoded` se establece mediante *join* del valor *str* de los elementos del atributo `groups` con `' , '`.

Se puede utilizar una lista de objetos *Address* y *Group* en cualquier combinación para establecer el valor de un encabezado de dirección. Los objetos *Group* cuyo `display_name` sea `None` se interpretarán como direcciones únicas, lo que permite copiar una lista de direcciones con los grupos intactos utilizando la lista obtenida del atributo `groups` de el encabezado de origen.

class `email.headerregistry.SingleAddressHeader`

Una subclase de *AddressHeader* que agrega un atributo adicional:

address

La única dirección codificada por el valor del encabezado. Si el valor del encabezado en realidad contiene más de una dirección (lo que sería una violación del RFC bajo el valor predeterminado *policy*), acceder a este atributo resultará en un *ValueError*.

Muchas de las clases anteriores también tienen una variante *Unique* (por ejemplo, “*UniqueUnstructuredHeader*”). La única diferencia es que en la variante *Unique*, *max_count* se establece en 1.

class `email.headerregistry.MIMEVersionHeader`

En realidad, solo hay un valor válido para el encabezado *MIME-Version*, y ese es `1.0`. Para pruebas futuras, esta clase de encabezado admite otros números de versión válidos. Si un número de versión tiene un valor válido por **RFC 2045**, entonces el objeto de encabezado tendrá valores distintos de `None` para los siguientes atributos:

version

El número de versión como una cadena de caracteres, con cualquier espacio en blanco y/o comentarios eliminados.

major

El número de versión mayor como un entero

minor

El número de versión menor como un entero

class `email.headerregistry.ParameterizedMIMEHeader`

Todos los encabezados *MIME* comienzan con el prefijo “*Content-*”. Cada encabezado específico tiene un valor determinado, que se describe en la clase de ese encabezado. Algunos también pueden tomar una lista de parámetros complementarios, que tienen un formato común. Esta clase sirve como base para todos los encabezados *MIME* que toman parámetros.

params

Un diccionario que asigna nombres de parámetros a valores de parámetros.

class `email.headerregistry.ContentTypeHeader`

Una clase *ParameterizedMIMEHeader* que maneja el encabezado *Content-Type*.

content_type

La cadena de caracteres de tipo de contenido, en la forma *maintype/subtype*.

maintype

subtype

class `email.headerregistry.ContentDispositionHeader`

Una clase *ParameterizedMIMEHeader* que maneja el encabezado *Content-Disposition*.

content_disposition

`inline` y `attachment` son los únicos valores válidos de uso común.

class `email.headerregistry.ContentTransferEncoding`

Maneja el encabezado de *Content-Transfer-Encoding*.

cte

Los valores válidos son 7bit, 8bit, base64, y quoted-printable. Consulte [RFC 2045](#) para obtener más información.

```
class email.headerregistry.HeaderRegistry (base_class=BaseHeader,          de-
                                         fault_class=UnstructuredHeader,
                                         use_default_map=True)
```

Esta es la mecánica utilizada por *EmailPolicy* por defecto. HeaderRegistry construye la clase utilizada para crear una instancia de encabezado dinámicamente, usando *base_class* y una clase especializada recuperada de un registro que contiene. Cuando un nombre de encabezado determinado no aparece en el registro, la clase especificada por *default_class* se utiliza como clase especializada. Cuando *use_default_map* es True (el valor predeterminado), la asignación estándar de nombres de encabezado a clases se copia en el registro durante la inicialización. *base_class* es siempre la última clase en la lista `__bases__` de la clase generada.

Las asignaciones predeterminadas son:

```
subject UniqueUnstructuredHeader
date UniqueDateHeader
resent-date DateHeader
orig-date UniqueDateHeader
sender UniqueSingleAddressHeader
resent-sender SingleAddressHeader
to UniqueAddressHeader
resent-to AddressHeader
cc UniqueAddressHeader
resent-cc AddressHeader
bcc UniqueAddressHeader
resent-bcc AddressHeader
from UniqueAddressHeader
resent-from AddressHeader
reply-to UniqueAddressHeader
mime-version MIMEVersionHeader
content-type ContentTypeHeader
content-disposition ContentDispositionHeader
content-transfer-encoding ContentTransferEncodingHeader
message-id MessageIDHeader
```

HeaderRegistry tiene los siguientes métodos:

```
map_to_type (self, name, cls)
```

name es el nombre del encabezado que se asignará. Se convertirá a minúsculas en el registro. *cls* es la clase especializada que se utilizará, junto con *base_class*, para crear la clase utilizada para instanciar encabezados que coincidan con *name*.

```
__getitem__ (name)
```

Construye y retorna una clase para manejar la creación de un encabezado *nombre*.

```
__call__ (name, value)
```

Recupera el encabezado especializado asociado con *name* del registro (usando *default_class* si *name* no aparece en el registro) y lo compone con *base_class* para producir una clase, llama al constructor de

la clase construida, pasándole el mismo lista de argumentos y, finalmente, retorna la instancia de clase creada de ese modo.

Las siguientes clases son las clases que se utilizan para representar datos analizados a partir de encabezados estructurados y, en general, pueden ser utilizadas por un programa de aplicación para construir valores estructurados para asignar a encabezados específicos.

```
class email.headerregistry.Address (display_name=", username", domain",  
                                     addr_spec=None)
```

La clase utilizada para representar una dirección de correo electrónico. La forma general de una dirección es:

`[display_name] <username@domain>`

or:

`username@domain`

donde cada parte debe ajustarse a reglas de sintaxis específicas explicadas en [RFC 5322](#).

Para su comodidad, se puede especificar *addr_spec* en lugar de *username* y *domain*, en cuyo caso *username* y *domain* se analizarán a partir de *addr_spec*. Un *addr_spec* debe ser una cadena de caracteres entre comillas RFC adecuada; si no es *Address*, se generará un error. Se permiten caracteres Unicode y se codificarán como propiedad cuando se serialicen. Sin embargo, según las RFC, *no* se permite unicode en la parte del nombre de usuario de la dirección.

display_name

La parte del nombre para mostrar de la dirección, si la hubiera, con todas las citas eliminadas. Si la dirección no tiene un nombre para mostrar, este atributo será una cadena vacía.

username

La parte del *username* de la dirección, con todas las citas eliminadas.

domain

La parte de *domain* de la dirección.

addr_spec

La parte de la dirección *username@domain*, citada correctamente para usarla como dirección simple (el segundo formulario que se muestra arriba). Este atributo no es mutable.

__str__()

El valor *str* del objeto es la dirección citada de acuerdo con las reglas [RFC 5322](#), pero sin codificación de transferencia de contenido de ningún carácter que no sea ASCII.

Para admitir SMTP ([RFC 5321](#)), *Address* maneja un caso especial: si *username* y *domain* son ambos la cadena de caracteres vacía (o *None*), entonces el valor de cadena de caracteres *Address* es *<>*.

```
class email.headerregistry.Group (display_name=None, addresses=None)
```

La clase utilizada para representar un grupo de direcciones. La forma general de un grupo de direcciones es:

`display_name: [address-list];`

Para facilitar el procesamiento de listas de direcciones que constan de una mezcla de grupos y direcciones únicas, también se puede utilizar un *Group* para representar direcciones únicas que no forman parte de un grupo al establecer *display_name* en *None* y proporcionando una lista de direcciones únicas como *addresses*.

display_name

El *display_name* del grupo. Si es *None* y hay exactamente una *Address* en *addresses*, entonces el *Group* representa una única dirección que no está en un grupo.

addresses

Posiblemente una tupla vacía de *Address* que representan las direcciones en el grupo.

__str__()

El valor *str* de un *Group* se formatea de acuerdo con [RFC 5322](#), pero sin codificación de transferencia de contenido de ningún carácter que no sea ASCII. Si *display_name* no es ninguno y hay una sola *Address* en la lista de *addresses*, el valor de *str* será el mismo que el *str* de ese single *Address*.

Pie de notas

19.1.7 `email.contentmanager`: Gestión de contenido MIME

Código fuente: `Lib/email/contentmanager.py`

Nuevo en la versión 3.6:¹

class `email.contentmanager.ContentManager`

Clase base para gestores de contenido. Proporciona los mecanismos de registro estándar para registrar convertidores entre contenido MIME y otras representaciones, así como los métodos de envío `get_content` y `set_content`.

get_content (*msg*, **args*, ***kw*)

Busca una función de controlador basada en el `mimetype` de *msg* (ver el siguiente párrafo), la llama, le pasa todos los argumentos y retorna el resultado de la llamada. La expectativa es que el controlador extraiga la carga útil de *msg* y retorne un objeto que codifica información sobre los datos extraídos.

Para encontrar el controlador, busca las siguientes llaves en el registro, deteniéndose con la primera que encuentre:

- la cadena que representa el tipo MIME completo (`maintype/subtype`)
- la cadena de caracteres que representa el `maintype`
- la cadena de caracteres vacía

Si ninguna de estas llaves produce un controlador, se lanza una excepción `KeyError` para el tipo MIME completo.

set_content (*msg*, *obj*, **args*, ***kw*)

Si el `maintype` es `multipart`, se lanza un `TypeError`; de lo contrario, busca una función de controlador basada en el tipo de *obj* (ver el siguiente párrafo), llama a `clear_content()` en el *msg* y llama a la función de controlador, pasando todos los argumentos. La expectativa es que el controlador transforme y almacene *obj* en *msg*, posiblemente realizando otros cambios a *msg* también, como agregar varios encabezados MIME para codificar la información necesaria para interpretar los datos almacenados.

Para encontrar el controlador, obtiene el tipo de *obj* (`typ = type(obj)`), y busca las siguientes llaves en el registro, deteniéndose con la primera encontrada:

- el tipo en sí (`typ`)
- el nombre completo de calificación del tipo (`typ.__module__ + '.' + typ.__qualname__`).
- el nombre de calificación del tipo (`typ.__qualname__`)
- el nombre del tipo (`typ.__name__`).

Si ninguno de los anteriores coincide, repite todas las comprobaciones anteriores para cada uno de los tipos en el `MRO` (`typ.__mro__`). Finalmente, si ninguna otra llave produce un controlador, busca un controlador para la llave `None`. Si no hay un controlador para `None`, lanza un `KeyError` para el nombre completo de calificación del tipo.

También agrega un encabezado `MIME-Version` si no hay uno presente (vea también `MIMEPart`).

add_get_handler (*key*, *handler*)

Registra el *handler* de funciones como el manejador de *key*. Para los posibles valores de *key*, consulte `get_content()`.

add_set_handler (*typekey*, *handler*)

Registra el *handler* como la función a llamar cuando un objeto de un tipo coincidente *typekey* se pasa a `set_content()`. Para los posibles valores de *typekey*, consulte `set_content()`.

¹ Originalmente añadido en la versión 3.4 como un *módulo provisional*

Instancias gestoras de contenido

Actualmente, el paquete de correo electrónico solo proporciona un administrador de contenido concreto, `raw_data_manager`, aunque en el futuro se pueden agregar más. `raw_data_manager` es el `content_manager` proporcionado por `EmailPolicy` y sus derivados.

`email.contentmanager.raw_data_manager`

Este administrador de contenido proporciona sólo una interfaz mínima más allá de la proporcionada por `Message` en sí: trata solo con texto, cadenas de bytes sin procesar, y objetos `Message`. Sin embargo, proporciona ventajas significativas en comparación con la API base: `get_content` en una parte de texto retornará una cadena de caracteres unicode sin que la aplicación tenga que decodificarla manualmente, `set_content` proporciona un amplio conjunto de opciones para controlar los encabezados añadidos a una parte y controlar la codificación de transferencia de contenido, y permite el uso de los diversos métodos `add_`, simplificando así la creación de mensajes multiparte.

`email.contentmanager.get_content(msg, errors='replace')`

Retorna la carga útil de la parte como una cadena de caracteres (para partes de `text`), un objeto `EmailMessage` (para partes de `message/rfc822`), o un objeto de bytes (para todos los demás tipos que no son multiparte). Lanza un `KeyError` si se llama en un multiparte. Si la parte es una parte de `text` y se especifica `errors`, se usa como el controlador de errores al decodificar la carga útil a unicode. El controlador de errores predeterminado es `replace`.

`email.contentmanager.set_content(msg, <'str'>, subtype="plain", charset='utf-8', cte=None, disposition=None, filename=None, cid=None, params=None, headers=None)`

`email.contentmanager.set_content(msg, <'bytes'>, maintype, subtype, cte="base64", disposition=None, filename=None, cid=None, params=None, headers=None)`

`email.contentmanager.set_content(msg, <'EmailMessage'>, cte=None, disposition=None, filename=None, cid=None, params=None, headers=None)`

Añade cabeceras y carga útil al `msg`:

Añade un encabezado `Content-Type` con un valor `maintype/subtype`.

- Para `str`, establece el `maintype` de MIME en `text`, y establece el subtipo en `subtype` si se especifica, o `plain` si no está presente.
- Para `bytes`, usa el `maintype` y `subtype` especificados, o lanza un `TypeError` si no se especifican.
- Para objetos `EmailMessage`, establece el `maintype` en `message`, y establece el `subtype` en `subtype` si se especifica o `rfc822` si no se especifica. Si `subtype` es `partial`, se lanza un error (los objetos de `bytes` deben usarse para construir partes `message/partial`).

Si se proporciona `charset` (lo cual solo es válido para `str`), codifica la cadena de caracteres en bytes utilizando el conjunto de caracteres especificado. El valor por defecto es `utf-8`. Si el `charset` especificado es un alias conocido del nombre de un conjunto de caracteres del estándar MIME, utiliza el conjunto de caracteres estándar en su lugar.

Si se establece `cte`, codifica la carga útil mediante la codificación de transferencia de contenido especificada y establece el encabezado `Content-Transfer-Encoding` en ese valor. Los valores posibles para `cte` son `quoted-printable`, `base64`, `7bit`, `8bit`, y `binary`. Si la entrada no se puede codificar en la codificación especificada (por ejemplo, especificando un `cte` de `7bit` para una entrada que contiene valores no ASCII), se lanza un `ValueError`.

- Para objetos `str`, si `cte` no está configurado, se usa la heurística para determinar la codificación más compacta.
- Para `EmailMessage`, según [RFC 2046](#), se lanza un error si se solicita un `cte` de `quoted-printable` o `base64` para el `subtype` `rfc822`, y para cualquier `cte` que no sea `7bit` para el `subtype` `external-body`. Para `message/rfc822`, se usa `8bit` si no se especifica `cte`. Para todos los demás valores de `subtype`, se usa `7bit`.

Nota: Un *cte* de binary todavía no funciona correctamente. El objeto `EmailMessage` modificado por `set_content` es correcto, pero `BytesGenerator` no lo serializa correctamente.

Si se establece *disposición*, se usa como valor del encabezado `Content-Disposition`. Si no se especifica y se especifica *filename*, agrega el encabezado con el valor `attachment`. Si no se especifica *disposition* y tampoco se especifica *filename*, no agrega el encabezado. Los únicos valores válidos para *disposition* son `attachment` e `inline`.

Si se especifica el *filename*, se usa como el valor del parámetro `filename` del encabezado `Content-Disposition`.

Si se especifica *cid*, agrega un encabezado `Content-ID` con valor *cid*.

Si se especifica *params*, itera su método `items` y use los pares resultantes (`key`, `value`) para establecer parámetros adicionales en el encabezado `Content-Type`.

Si se especifica *headers* y es una lista de cadenas de caracteres de la forma `headername: headervalue` o una lista de objetos `header` (que se distinguen de las cadenas de caracteres por tener un atributo `name`), agrega los encabezados a *msg*.

Notas al pie de página

19.1.8 email: Ejemplos

Aquí hay algunos ejemplos de cómo usar el paquete `email` para leer, escribir y enviar mensajes de correo electrónico simples, así como mensajes MIME más complejos.

Primero, veamos cómo crear y enviar un mensaje de texto simple (tanto el contenido de texto como las direcciones pueden contener caracteres unicode):

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.message import EmailMessage

# Open the plain text file whose name is in textfile for reading.
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = f'The contents of {textfile}'
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server.
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

El análisis de los encabezados **RFC 822** se pueden hacer fácilmente usando las clases del módulo `parser`:

```
# Import the email modules we'll need
from email.parser import BytesParser, Parser
from email.policy import default

# If the e-mail headers are in a file, uncomment these two lines:
```

(continué en la próxima página)

(proviene de la página anterior)

```
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

# Or for parsing headers in a string (this is an uncommon operation), use:
headers = Parser(policy=default).parsestr(
    'From: Foo Bar <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: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# You can also access the parts of the addresses:
print('Recipient username: {}'.format(headers['to'].addresses[0].username))
print('Sender name: {}'.format(headers['from'].addresses[0].display_name))
```

Aquí hay un ejemplo de cómo enviar un mensaje MIME que contiene un grupo de fotos familiares que pueden residir en un directorio:

```
# Import smtplib for the actual sending function
import smtplib

# And imghdr to find the types of our images
import imghdr

# Here are the email package modules we'll need
from email.message import EmailMessage

# Create the container email message.
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

# Open the files in binary mode. Use imghdr to figure out the
# MIME subtype for each specific image.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
        msg.add_attachment(img_data, maintype='image',
                           subtype=imghdr.what(None, img_data))

# Send the email via our own SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

Aquí hay un ejemplo de cómo enviar todo el contenido de un directorio como un mensaje de correo electrónico:¹

```
#!/usr/bin/env python3

"""Send the contents of a directory as a MIME message."""
```

(continué en la próxima página)

¹ Gracias a Matthew Dixon Cowles por la inspiración y ejemplos originales.

(proviene de la página anterior)

```

import os
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from argparse import ArgumentParser

from email.message import EmailMessage
from email.policy import SMTP

def main():
    parser = ArgumentParser(description="""\
Send the contents of a directory as a MIME message.
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_argument('-d', '--directory',
                        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_argument('-o', '--output',
                        metavar='FILE',
                        help="""Print the composed message to FILE instead of
sending the message to the SMTP server.""")
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header (required)')
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one required)')

    args = parser.parse_args()
    directory = args.directory
    if not directory:
        directory = '.'
    # Create the message
    msg = EmailMessage()
    msg['Subject'] = f'Contents of directory {os.path.abspath(directory)}'
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
    msg.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)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
                              maintype=maintype,
                              subtype=subtype,

```

(continué en la próxima página)

(proviene de la página anterior)

```
                                filename=filename)
# Now send or store the message
if args.output:
    with open(args.output, 'wb') as fp:
        fp.write(msg.as_bytes(policy=SMTP))
else:
    with smtplib.SMTP('localhost') as s:
        s.send_message(msg)

if __name__ == '__main__':
    main()
```

Aquí hay un ejemplo de cómo descomprimir un mensaje MIME como el anterior, en un directorio de archivos:

```
#!/usr/bin/env python3

"""Unpack a MIME message into a directory of files."""

import os
import email
import mimetypes

from email.policy import default
from argparse import ArgumentParser

def main():
    parser = ArgumentParser(description="""\
Unpack a MIME message into a directory of files.
""")
    parser.add_argument('-d', '--directory', required=True,
                        help="""Unpack the MIME message into the named
                        directory, which will be created if it doesn't already
                        exist.""")
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
        msg = email.message_from_binary_file(fp, policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass

    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:
                # Use a generic bag-of-bits extension
                ext = '.bin'
            filename = f'part-{counter:03d}{ext}'
```

(continué en la próxima página)

(proviene de la página anterior)

```

        counter += 1
        with open(os.path.join(args.directory, filename), 'wb') as fp:
            fp.write(part.get_payload(decode=True))

if __name__ == '__main__':
    main()

```

Aquí hay un ejemplo de cómo crear un mensaje HTML con una versión alternativa de texto sin formato. Para hacer las cosas un poco más interesantes, incluimos una imagen relacionada en la parte html, y guardamos una copia de lo que vamos a enviar en el disco, además de enviarlo.

```

#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# Create the base text message.
msg = EmailMessage()
msg['Subject'] = "Ayons asperges pour le déjeuner"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com")
msg['To'] = (Address("Penelope Pussycat", "penelope", "example.com"),
            Address("Fabrette Pussycat", "fabrette", "example.com"))
msg.set_content("""\
Salut!

Cela ressemble à un excellent recipie[1] déjeuner.

[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718

--Pepé
""")

# Add the html version. This converts the message into a multipart/alternative
# container, with the original text message as the first part and the new html
# message as the second part.
asparagus_cid = make_msgid()
msg.add_alternative("""\
<html>
  <head></head>
  <body>
    <p>Salut!</p>
    <p>Cela ressemble à un excellent
      <a href="http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718">
        recipie
      </a> déjeuner.
    </p>
    
  </body>
</html>
""".format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')
# note that we needed to peel the <> off the msgid for use in the html.

# Now add the related image to the html part.
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image', 'jpeg',
                                     cid=asparagus_cid)

```

(continué en la próxima página)

(proviene de la página anterior)

```
# Make a local copy of what we are going to send.
with open('outgoing.msg', 'wb') as f:
    f.write(bytes(msg))

# Send the message via local SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

Si nos enviaron el mensaje del último ejemplo, aquí hay una forma en que podríamos procesarlo:

```
import os
import sys
import tempfile
import mimetypes
import webbrowser

# Import the email modules we'll need
from email import policy
from email.parser import BytesParser

# An imaginary module that would make this work and be safe.
from imaginary import magic_html_parser

# In a real program you'd get the filename from the arguments.
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# Now the header items can be accessed as a dictionary, and any non-ASCII will
# be converted to unicode:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# If we want to print a preview of the message content, we can extract whatever
# the least formatted payload is and print the first three lines. Of course,
# if the message has no plain text part printing the first three lines of html
# is probably useless, but this is just a conceptual example.
simplest = msg.get_body(preferencelist=('plain', 'html'))
print()
print(''.join(simplest.get_content().splitlines(keepends=True)[:3]))

ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# We can extract the richest alternative in order to display it:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':
        for line in richest.get_content().splitlines():
            print(line)
        sys.exit()
    elif richest['content-type'].subtype == 'html':
        body = richest
    else:
        print("Don't know how to display {}".format(richest.get_content_type()))
        sys.exit()
elif richest['content-type'].content_type == 'multipart/related':
    body = richest.get_body(preferencelist=('html'))
    for part in richest.iter_attachments():
```

(continué en la próxima página)

(proviene de la página anterior)

```

fn = part.get_filename()
if fn:
    extension = os.path.splitext(part.get_filename())[1]
else:
    extension = mimetypes.guess_extension(part.get_content_type())
with tempfile.NamedTemporaryFile(suffix=extension, delete=False) as f:
    f.write(part.get_content())
    # again strip the <> to go from email form of cid to html form.
    partfiles[part['content-id'][1:-1]] = f.name
else:
    print("Don't know how to display {}".format(richest.get_content_type()))
    sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    # The magic_html_parser has to rewrite the href="cid:..." attributes to
    # point to the filenames in partfiles. It also has to do a safety-sanitize
    # of the html. It could be written using html.parser.
    f.write(magic_html_parser(body.get_content(), partfiles))
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
    os.remove(fn)

# Of course, there are lots of email messages that could break this simple
# minded program, but it will handle the most common ones.

```

Hasta el aviso, el resultado de lo anterior es:

```

To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.
↪com>
From: Pepé Le Pew <pepe@example.com>
Subject: Ayons asperges pour le déjeuner

Salut!

Cela ressemble à un excellent recipie[1] déjeuner.

```

Notas al pie

API heredada:

19.1.9 `email.message.Message`: Representar un mensaje de correo electrónico usando la API `compat32`

La clase `Message` es muy similar a la clase `EmailMessage`, sin los métodos añadidos por esa clase y con el comportamiento predeterminado de algunos otros métodos siendo ligeramente diferente. También documentamos aquí algunos métodos que, aun siendo soportados por `EmailMessage`, no están recomendados a no ser que estés lidiando con código heredado.

Por lo demás, la filosofía y estructura de las dos clases es la misma.

Este documento describe el comportamiento bajo la política por defecto (para `Message`) `Compat32`. Si vas a usar otra política, deberías estar usando la clase `EmailMessage` en su lugar.

An email message consists of *headers* and a *payload*. Headers must be [RFC 5322](#) style 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. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as `multipart/*` or `message/rfc822`.

El modelo conceptual proporcionado por un objeto `Message` es el de un diccionario ordenado de encabezados con métodos adicionales para acceder a información especializada de los encabezados, para acceder a la carga, para generar una versión serializada del mensaje y para recorrer recursivamente el árbol del objeto. Ten en cuenta que son soportados encabezados duplicados pero deben ser usados métodos especiales para acceder a ellos.

El pseudodiccionario `Message` es indexado por los nombres de encabezados, los cuales deben ser valores ASCII. Los valores del diccionario son cadenas que se supone que contienen sólo caracteres ASCII; hay algún manejo especial para la entrada no ASCII, pero esta no siempre produce los resultados correctos. Los encabezados son almacenados y retornados preservando mayúsculas y minúsculas, pero los nombres de campos son emparejados sin distinción entre mayúsculas y minúsculas. También puede haber sólo un encabezado de envoltura, también conocido como el encabezado `Unix-From` o el encabezado `From_`. La carga (*payload*) es una cadena o bytes, en el caso de objetos de mensajes simples, o una lista de objetos `Message`, para contenedores de documentos MIME (ej. `multipart/*` y `message/rfc822`).

Aquí están los métodos de la clase `Message`:

class `email.message.Message` (*policy=compat32*)

Si se especifica *policy* (debe ser una instancia de una clase *policy*) utiliza las reglas que especifica para actualizar y serializar la representación del mensaje. Si no se define *policy*, utiliza la política `compat32`, la cual mantiene compatibilidad con la versión de Python 3.2 del paquete `email`. Para más información consulta la documentación de *policy*.

Distinto en la versión 3.3: El argumento de palabra clave *policy* fue añadido.

as_string (*unixfrom=False, maxheaderlen=0, policy=None*)

Retorna el mensaje completo aplanado como una cadena. Cuando el parámetro opcional *unixfrom* es verdadero, el encabezado de envoltura se incluye en la cadena retornada. *unixfrom* es por defecto `False`. Por razones de compatibilidad con versiones anteriores, *maxheaderlen* es 0 por defecto, por lo que si quieres un valor diferente debes sobrescribirlo explícitamente (el valor especificado por *max_line_length* en la política será ignorado por este método). El argumento *policy* puede ser usado para sobrescribir la política por defecto obtenida de la instancia del mensaje. Esto puede ser usado para controlar algo del formato producido por el método, ya que la *policy* especificada puede ser pasada al `Generator`.

Aplanar el mensaje puede desencadenar cambios en `Message` si por defecto necesita ser rellenado para completar la transformación a una cadena (por ejemplo, límites MIME pueden ser generados o modificados).

Ten en cuenta que este método es proporcionado como conveniencia y puede no siempre formatear el mensaje de la forma que quieres. Por ejemplo, de forma predeterminada no realiza la mutilación de líneas que comienzan con `From` que es requerida por el formato unix mbox. Para mayor flexibilidad, instancia un `Generator` y utiliza su método `flatten()` directamente. Por ejemplo:

```
from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

Si el objeto de mensaje contiene datos binarios que no están codificados de acuerdo a los estándares RFC, los datos no compatibles serán reemplazados por puntos de código Unicode de «carácter desconocido». (Consulta también `as_bytes()` y `BytesGenerator`.)

Distinto en la versión 3.4: el argumento de palabra clave *policy* fue añadido.

__str__ ()

Equivalente a `as_string()`. Permite a `str(msg)` producir una cadena conteniendo el mensaje formateado.

as_bytes (*unixfrom=False, policy=None*)

Retorna el mensaje completo aplanado como un objeto de bytes. Cuando el argumento opcional *unixfrom* es verdadero, el encabezado de envoltura se incluye en la cadena retornada. *unixfrom* es por defecto `False`. El argumento *policy* puede ser usado para sobrescribir la política por defecto obtenida desde la

instancia del mensaje. Esto puede ser usado para controlar algo del formato producido por el método, ya que el *policy* especificado será pasado al `BytesGenerator`.

Aplanar el mensaje puede desencadenar cambios en `Message` si por defecto necesita ser rellenado para completar la transformación a una cadena (por ejemplo, límites MIME pueden ser generados o modificados).

Nota que este método es proporcionado como conveniencia y puede no siempre formatear el mensaje de la forma que quieres. Por ejemplo, por defecto no realiza la mutilación de línea que comienzan con `From` que es requerida por el formato unix mbox. Para mayor flexibilidad, instancia un `BytesGenerator` y utiliza su método `flatten()` directamente. Por ejemplo:

```
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

Nuevo en la versión 3.4.

`__bytes__()`

Equivalente a `as_bytes()`. Permite a `bytes(msg)` producir un objeto de bytes conteniendo el mensaje formateado.

Nuevo en la versión 3.4.

`is_multipart()`

Retorna `True` si la carga del mensaje es una lista de objetos heredados de `Message`, si no retorna `False`. Cuando `is_multipart()` retorna `False`, la carga debe ser un objeto de cadena (el cual puede ser una carga CTE codificada en binario). (Ten en cuenta que `is_multipart()` retornando `True` no significa necesariamente que `msg.get_content_maintype() == "multipart"` retornará `True`. Por ejemplo, `is_multipart` retornará `True` cuando el `Message` es de tipo `message/rfc822`.)

`set_unixfrom(unixfrom)`

Establece el mensaje del encabezado de envoltura a `unixfrom`, el cual debe ser una cadena.

`get_unixfrom()`

Retorna el mensaje del encabezado de envoltura. Por defecto a `None` si el encabezado de envoltura nunca fue definido.

`attach(payload)`

Añade el `payload` dado a la carga actual, la cual debe ser `None` o una lista de objetos `Message` antes de la invocación. Después de la invocación, la carga siempre será una lista de objetos `Message`. Si quieres definir la carga a un objeto escalar (ej. una cadena), usa `set_payload()` en su lugar.

Este es un método heredado. En la clase `EmailMessage` su funcionalidad es remplazada por `set_content()` y los métodos relacionados `make` y `add`.

`get_payload(i=None, decode=False)`

Retorna la carga (`payload`) actual, la cual será una lista de objetos `Message` cuando `is_multipart()` es `True`, o una cadena cuando `is_multipart()` es `False`. Si la carga es una lista y mutas el objeto de lista, modificarás la carga del mensaje.

Con el argumento opcional `i`, `get_payload()` retornará el elemento número `i` de la carga (`payload`), contando desde cero, si `is_multipart()` es `True`. Un `IndexError` será generado si `i` es menor que 0 ó mayor o igual que el número de elementos en la carga. Si la carga es una cadena (ej. `is_multipart()` es `False`) y se define `i`, se genera un `TypeError`.

El argumento opcional `decode` es un indicador que determina si una carga debería ser decodificada o no, de acuerdo al encabezado `Content-Transfer-Encoding`. Cuando es `True` y el mensaje no es multiparte, la carga será decodificada si el valor de su encabezado es `quoted-printable` o `base64`. Si se usa alguna otra codificación o falta el encabezado `Content-Transfer-Encoding`, la carga es retornada tal cual (sin decodificar). En todos los casos el valor retornado son datos binarios. Si el mensaje es multiparte y el indicador `decode` es `True`, entonces se retorna `None`. Si la carga es `base64` y no

fue perfectamente formada (falta relleno, tiene caracteres fuera del alfabeto base64), entonces un defecto apropiado será añadido a la propiedad `defect` del mensaje (`InvalidBase64PaddingDefect` o `InvalidBase64CharactersDefect`, respectivamente).

Cuando `decode` es `False` (por defecto) el cuerpo es retornado como una cadena sin decodificar el *Content-Transfer-Encoding*. Sin embargo, para un *Content-Transfer-Encoding* de 8bit, se realiza un intento para decodificar los bytes originales usando el `charset` especificado por el encabezado *Content-Type*, usando el manejador de error `replace`. Si ningún `charset` es especificado o si el `charset` dado no es reconocido por el paquete `email`, el cuerpo es decodificado usando el conjunto de caracteres ASCII por defecto.

Este es un método heredado. En la clase `EmailMessage` su funcionalidad es remplazada por `get_content()` y `iter_parts()`.

set_payload (*payload*, *charset=None*)

Define la carga completa del objeto mensaje a *payload*. Es responsabilidad del cliente asegurar invariantes de carga. El argumento opcional *charset* define el conjunto de caracteres por defecto del mensaje; consulta `set_charset()` para más detalles.

Este es un método heredado. En la clase `EmailMessage` su funcionalidad es remplazada por `set_content()`.

set_charset (*charset*)

Define el conjunto de caracteres de la carga a *charset*, el cual puede ser tanto una instancia `Charset` (ver `email.charset`), una cadena denominando un conjunto de caracteres, o `None`. Si es una cadena, será convertida a una instancia `Charset`. Si *charset* es `None`, el parámetro `charset` será eliminado del encabezado *Content-Type* (el mensaje no será modificado de otra manera). Cualquier otro valor generará un `TypeError`.

Si no hay un encabezado existente *MIME-Version*, será añadido uno. Si no hay un encabezado existente *Content-Type*, será añadido uno con valor `text/plain`. Tanto como si el encabezado *Content-Type* existe actualmente como si no, su parámetro `charset` será establecido a `charset.output_charset`. Si `charset.input_charset` y `charset.output_charset` difieren, la carga será recodificada al `output_charset`. Si no hay un encabezado existente *Content-Transfer-Encoding*, entonces la carga será codificada por transferencia, si es necesario, usando el `Charset` especificado y un encabezado con el valor apropiado será añadido. Si ya existe un encabezado *Content-Transfer-Encoding*, la carga se asume que ya está correctamente codificada usando ese *Content-Transfer-Encoding* y no es modificada.

Este es un método heredado. En la clase `EmailMessage` su funcionalidad es remplazada por el parámetro *charset* del método `email.message.EmailMessage.set_content()`.

get_charset ()

Retorna la instancia `Charset` asociada con la carga del mensaje.

Este es un método heredado. En la clase `EmailMessage` siempre retorna `None`.

Los siguientes métodos implementan una interfaz parecida a un mapeo para acceder a los encabezados **RFC 2822** del mensaje. Ten en cuenta que hay algunas diferencias semánticas entre esos métodos y una interfaz de mapeo normal (ej. diccionario). Por ejemplo, en un diccionario no hay claves duplicadas, pero aquí pueden haber encabezados de mensaje duplicados. También, en diccionarios no hay un orden garantizado de las claves retornadas por `keys()`, pero en un objeto `Message`, los encabezados siempre son retornados en el orden que aparecieron en el mensaje original, o en el que fueron añadidos al mensaje más tarde. Cualquier encabezado eliminado y vuelto a adicionar siempre es añadido al final de la lista de encabezados.

Esas diferencias semánticas son intencionales y están sesgadas hacia la máxima comodidad.

Ten en cuenta que en todos los casos, cualquier encabezado de envoltura presente en el mensaje no está incluido en la interfaz de mapeo.

En un modelo generado desde bytes, cualesquiera valores de encabezado que (en contravención de los RFCs) contienen bytes ASCII serán representados, cuando sean obtenidos mediante esta interfaz, como objetos `Header` con un conjunto de caracteres *unknown-8bit*.

__len__()

Retorna el número total de encabezados, incluyendo duplicados.

__contains__(name)

Retorna `True` si el objeto mensaje tiene un campo llamado *name*. La concordancia se realiza sin distinguir mayúsculas de minúsculas y *name* no debería incluir el carácter de doble punto final. Usado para el operador `in`, ej:

```
if 'message-id' in myMessage:
    print('Message-ID:', myMessage['message-id'])
```

__getitem__(name)

Retorna el valor del campo del encabezado nombrado. *name* no debe incluir el separador de campo de doble punto. Si falta un encabezado, se retorna `None`; nunca se genera un error `KeyError`.

Ten en cuenta que si el campo nombrado aparece más de una vez en los encabezados del mensaje, no se define cuales serán exactamente aquellos valores de campos retornados. Usa el método `get_all()` para obtener los valores de todos los encabezados nombrados existentes.

__setitem__(name, val)

Añade un encabezado al mensaje con el nombre de campo *name* y el valor *val*. El campo es añadido al final de los campos existentes del mensaje.

Ten en cuenta que esto no sobrescribe ni elimina ningún encabezado existente con el mismo nombre. Si quieres asegurar que el nuevo encabezado es el único presente en el mensaje con el nombre de campo *name*, elimina el campo primero, ej:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(name)

Elimina todas las ocurrencias de un campo con el nombre *name* de los encabezados del mensaje. No se genera ninguna excepción si el encabezado nombrado no está presente en los encabezados.

keys()

Retorna una lista de todos los nombres de campos de encabezados del mensaje.

values()

Retorna una lista de todos los valores de campos del mensaje.

items()

Retorna una lista de tuplas de dos elementos conteniendo todos los campos y valores de encabezados del mensaje.

get(name, failobj=None)

Retorna el valor del campo de encabezado nombrado. Esto es idéntico a `__getitem__()` excepto que el argumento *failobj* es retornado si falta el encabezado nombrado (por defecto a `None`).

Aquí hay algunos métodos útiles adicionales:

get_all(name, failobj=None)

Retorna una lista de todos los valores para el campo denominado *name*. Si no hay tales encabezados nombrados en el mensaje, retorna *failobj* (por defecto `None`).

add_header(_name, _value, **_params)

Configuración de encabezado extendida. Este método es similar a `__setitem__()` excepto que pueden ser provistos parámetros adicionales de encabezado como argumentos de palabra clave. *_name* es el campo de encabezado a añadir y *_value* es el valor *primario* para el encabezado.

Para cada elemento en el diccionario de argumentos de palabra clave *_params*, la clave se toma como el nombre del parámetro con guiones bajos convertidos a guiones medios (ya que los guiones medios son ilegales como identificadores en Python). Normalmente, el parámetro será añadido como `key="value"` a no ser que el valor sea `None`, en cuyo caso sólo la clave será añadida. Si el valor contiene caracteres no ASCII, puede ser especificado como una tupla de tres elementos en el formato `(CHARSET, LANGUAGE, VALUE)`, donde `CHARSET` es una cadena que nombra el conjunto de caracteres a ser

usado al codificar el valor, `LANGUAGE` puede normalmente ser definido a `None` o una cadena vacía (ver [RFC 2231](#) para otras posibilidades) y `VALUE` es la cadena del valor conteniendo puntos de caracteres no ASCII. Si no se pasa una tupla de tres elementos y el valor contiene caracteres no ASCII, se codifica automáticamente en formato [RFC 2231](#) usando `CHARSET` como `utf-8` y `LANGUAGE` como `None`.

Aquí hay un ejemplo:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

Esto añadirá un encabezado que se verá como

```
Content-Disposition: attachment; filename="bud.gif"
```

Un ejemplo con caracteres no ASCII:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

Lo que produce

```
Content-Disposition: attachment; filename*="iso-8859-1"Fu%DFballer.ppt"
```

replace_header (*_name*, *_value*)

Reemplaza un encabezado. Reemplaza el primer encabezado encontrado en el mensaje que concuerda con *_name*, conservando el orden del encabezado y el nombre del campo. Si no se encuentra ningún encabezado que concuerde, se genera un error *KeyError*.

get_content_type ()

Retorna el tipo de contenido del mensaje. La cadena retornada se fuerza a letras minúsculas de la forma *maintype/subtype*. Si no hay ningún encabezado *Content-Type* en el mensaje será retornado el tipo por defecto como es dado por *get_default_type* (). Dado que según [RFC 2045](#), los mensajes tienen siempre un tipo predeterminado, *get_content_type* () siempre retornará un valor.

[RFC 2045](#) define el tipo predeterminado del mensaje a *text/plain* a no ser que aparezca dentro de un contenedor *multipart/digest*, en cuyo caso sería *message/rfc822*. Si el encabezado *Content-Type* tiene una especificación de tipo inválido, [RFC 2045](#) ordena que el tipo por defecto sea *text/plain*.

get_content_maintype ()

Retorna el tipo de contenido principal del mensaje. Esta es la parte *maintype* de la cadena retornada por *get_content_type* ().

get_content_subtype ()

Retorna el tipo del subcontenido del mensaje. Esta es la parte *subtype* de la cadena retornada por *get_content_type* ().

get_default_type ()

Retorna el tipo del contenido por defecto. La mayoría de mensajes tienen un tipo de contenido por defecto de *text/plain*, excepto para mensajes que son subpartes de contenedores *multipart/digest*. Tales subpartes tienen como tipo de contenido predeterminado *message/rfc822*.

set_default_type (*ctype*)

Establece el tipo de contenido por defecto. *ctype* debería ser *text/plain* o *message/rfc822*, aunque esto no es obligatorio. El tipo de contenido predeterminado no se almacena en el encabezado *Content-Type*.

get_params (*failobj=None*, *header='content-type'*, *unquote=True*)

Retorna los parámetros del *Content-Type* del mensaje como una lista. Los elementos de la lista retornada son tuplas de dos elementos de pares clave/valor, tal y como son partidas por el signo '='. El lado izquierdo del '=' es la clave, mientras el lado derecho es el valor. Si no hay signo '=' en el parámetro, el valor es la cadena vacía, en caso contrario el valor es como se describe en *get_param* () y no está citado si el parámetro opcional *unquote* es `True` (por defecto).

El parámetro opcional *failobj* es el objeto a retornar si no hay encabezado *Content-Type*. El parámetro opcional *header* es el encabezado a buscar en lugar de *Content-Type*.

Este es un método heredado. En la clase `EmailMessage` su funcionalidad es remplazada por la propiedad *params* de los objetos individuales de encabezado retornados por los métodos de acceso del encabezado.

get_param (*param*, *failobj*=None, *header*='content-type', *unquote*=True)

Retorna el valor del parámetro *param* del encabezado *Content-Type* como una cadena. Si el mensaje no tiene encabezado *Content-Type* o si no existe tal parámetro, entonces se retorna *failobj* (por defecto es None).

El parámetro opcional *header*, si es definido, especifica el encabezado del mensaje a usar en lugar de *Content-Type*.

Las claves de parámetros siempre son comparadas distinguiendo entre mayúsculas y minúsculas. El valor de retorno puede ser una cadena, una tupla de 3 elementos si el parámetro fue codificado según [RFC 2231](#). Cuando es una tupla de 3 elementos, los elementos del valor tienen la forma (CHARSET, LANGUAGE, VALUE). Ten en cuenta que CHARSET y LANGUAGE pueden ser None, en cuyo caso debes considerar VALUE como codificado en el conjunto de caracteres *us-ascii*. Generalmente puedes ignorar LANGUAGE.

Si a tu aplicación no le importa si el parámetro fue codificado según [RFC 2231](#), puedes contraer el valor del parámetro invocando `email.utils.collapse_rfc2231_value()`, pasando el valor de retorno desde `get_param()`. Esto retornará una cadena Unicode convenientemente decodificada cuando el valor es una tupla o la cadena original sin entrecomillar si no lo es. Por ejemplo:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

En cualquier caso, el valor del parámetro (tanto la cadena retornada o el elemento VALUE en la tupla de 3 elementos) siempre está sin entrecomillar, a no ser que *unquote* está establecido a False.

Este es un método heredado. En la clase `EmailMessage` su funcionalidad es remplazada por la propiedad *params* de los objetos individuales de encabezado retornados por los métodos de acceso del encabezado.

set_param (*param*, *value*, *header*='Content-Type', *requote*=True, *charset*=None, *language*=", *replace*=False)

Establece un parámetro en el encabezado *Content-Type*. Si el parámetro ya existe en el encabezado, su valor será remplazado con *value*. Si el encabezado *Content-Type* no ha sido definido todavía para este mensaje, será establecido a *text/plain* y el nuevo valor del parámetro será añadido según [RFC 2045](#).

El parámetro opcional *header* especifica una alternativa a *Content-Type* y todos los parámetros serán entrecomillados si es necesario a no ser que el parámetro opcional *requote* sea False (por defecto es True).

Si se especifica el parámetro opcional *charset*, el parámetro será codificado de acuerdo a [RFC 2231](#). El parámetro opcional *language* especifica el lenguaje RFC 2231, por defecto una cadena vacía. Tanto *charset* como *language* deberían ser cadenas.

Si *replace* es False (por defecto) el encabezado será movido al final de la lista de encabezados. Si *replace* es True, el encabezado será actualizado.

Distinto en la versión 3.4: el parámetro de palabra clave *replace* fue añadido.

del_param (*param*, *header*='content-type', *requote*=True)

Elimina el parámetro dado completamente del encabezado *Content-Type*. El encabezado será reescrito en sí mismo sin el parámetro o su valor. Todos los valores serán entrecomillados si es necesario a no ser que *requote* sea False (por defecto es True). El parámetro opcional *header* especifica una alternativa a *Content-Type*.

set_type (*type*, *header*='Content-Type', *requote*=True)

Establece el tipo y subtipo principal para el encabezado *Content-Type*. *type* debe ser una cadena de

la forma *maintype/subtype*, si no será generado un *ValueError*.

Este método reemplaza el encabezado *Content-Type*, manteniendo todos los parámetros en su lugar. Si *quote* es *False*, este dejará el encabezado existente tal como está, en caso contrario los parámetros serán entrecomillados (por defecto).

Un encabezado alternativo puede ser especificado en el argumento *header*. Cuando el encabezado *Content-Type* es definido, un encabezado *MIME-Version* también es añadido.

Este es un método heredado. En la clase *EmailMessage* su funcionalidad es reemplazada por los métodos *make_* y *add_*.

get_filename (*failobj=None*)

Retorna el valor del parámetro *filename* del encabezado *Content-Disposition* del mensaje. Si el encabezado no tiene un parámetro *filename*, este método recurre a buscar el parámetro *name* en el encabezado *Content-Type*. Si tampoco se encuentra o falta el encabezado, entonces retorna *failobj*. La cadena retornada siempre será sin entrecomillar según *email.utils.unquote()*.

get_boundary (*failobj=None*)

Retorna el valor del parámetro *boundary* del encabezado *Content-Type* del mensaje o *failobj* tanto si falta el encabezado como si no tiene parámetro *boundary*. La cadena retornada siempre será sin entrecomillar según *email.utils.unquote()*.

set_boundary (*boundary*)

Establece el parámetro *boundary* del encabezado *Content-Type* a *boundary*. *set_boundary()* siempre entrecomillará *boundary* si es necesario. Se genera *HeaderParseError* si el objeto de mensaje no tiene encabezado *Content-Type*.

Ten en cuenta que usar este método es sutilmente diferente a borrar el antiguo encabezado *Content-Type* y añadir uno nuevo con el nuevo límite mediante *add_header()* porque *set_boundary()* preserva el orden del encabezado *Content-Type* en la lista de encabezados. Sin embargo, no conserva ninguna línea de continuación que pueden haber estado presentes en el encabezado original *Content-Type*.

get_content_charset (*failobj=None*)

Retorna el parámetro *charset* del encabezado *Content-Type*, forzado a letras minúsculas. Si no hay un encabezado *Content-Type* o si ese encabezado no tiene parámetro *charset*, se retorna *failobj*.

Ten en cuenta que este método difiere de *get_charset()*, el cual retorna la instancia *Charset* para la codificación por defecto del cuerpo del mensaje.

get_charsets (*failobj=None*)

Retorna una lista conteniendo los nombres de los conjuntos de caracteres en el mensaje. Si el mensaje es *multipart*, entonces la lista contendrá un elemento para cada subparte en la carga (*payload*), en caso contrario será una lista de un elemento.

Cada elemento en la lista será una cadena la cual es el valor del parámetro *charset* en el encabezado *Content-Type* para la subparte representada. Sin embargo, si la subparte no tiene encabezado *Content-Type*, no tiene parámetro *charset* o no es del tipo MIME *text* principal, entonces ese elemento en la lista retornada será *failobj*.

get_content_disposition ()

Retorna el valor en minúsculas (sin parámetros) del encabezado del mensaje *Content-Disposition* si tiene uno o *None*. Los valores posibles para este método son *inline*, *attachment* o *None* si el mensaje sigue el **RFC 2183**.

Nuevo en la versión 3.5.

walk ()

El método *walk()* es un generador de todo propósito el cual puede ser usado para iterar sobre todas las partes y subpartes de árbol de objeto de mensaje, en orden de recorrido de profundidad primero. Siempre usarás típicamente *walk()* como iterador en un bucle *for*; cada iteración retorna la siguiente subparte.

Aquí hay un ejemplo que imprime el tipo MIME de cada parte de una estructura de mensaje multiparte:


```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

`walk` itera sobre las subpartes de cualquier parte donde `is_multipart()` retorna `True`, aunque `msg.get_content_maintype() == 'multipart'` puede retornar `False`. Vemos esto en nuestro ejemplo haciendo uso de la función de ayuda de depuración `_structure`:

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
  text/plain
    message/delivery-status
      text/plain
      text/plain
    message/rfc822
      text/plain
```

Aquí las partes de `message` no son `multipart`s, pero contienen subpartes. `is_multipart()` retorna `True` y `walk` desciende a las subpartes.

Los objetos *Message* pueden contener opcionalmente dos atributos de instancia, los cuales pueden ser usados al generar el texto plano de un mensaje MIME.

preamble

El formato de un documento MIME permite algo de texto entre la línea en blanco que sigue a los encabezados y la primera cadena límite multiparte. Normalmente, este texto nunca es visible en un lector de correo compatible con MIME porque queda fuera de la armadura MIME estándar. Sin embargo, viendo el texto del mensaje en crudo o viendo el mensaje en un lector no compatible con MIME, este texto puede volverse visible.

El atributo *preamble* contiene este texto de refuerzo adicional para documentos MIME. Cuando el *Parser* descubre algo de texto después de los encabezados pero antes de la primera cadena límite, asigna este texto al atributo *preamble* del mensaje. Cuando el *Generator* está escribiendo la representación de texto sin formato de un mensaje MIME y puede encontrar el mensaje como un atributo *preamble*, escribirá este texto en el área entre los encabezados y el primer límite. Consulta *email.parser* y *email.generator* para más detalles.

Ten en cuenta que si el objeto de mensaje no tiene preámbulo, el atributo *preamble* será `None`.

epilogue

El atributo *epilogue* actúa de la misma manera que el atributo *preamble*, excepto que contiene texto que aparece entre el último límite y el fin del mensaje.

No necesitas establecer el epílogo de la cadena vacía en orden para el *Generator* para imprimir una nueva línea al final del archivo.

defects

El atributo *defects* contiene una lista de todos los problemas encontrados al analizar este mensaje. Consulta [email.errors](#) para una descripción detallada de los posibles defectos de análisis.

19.1.10 email.mime: Creación de correo electrónico y objetos MIME desde cero

Código fuente: [Lib/email/mime/](#)

Este módulo forma parte de la API heredada de correo electrónico (Compat32). Su funcionalidad se sustituye parcialmente por el [contentmanager](#) en la nueva API, pero en ciertas aplicaciones estas clases pueden seguir siendo útiles, incluso en código no heredado.

Normalmente, se obtiene una estructura de objeto de mensaje pasando un archivo o algún texto a un analizador, que analiza el texto y retorna el objeto de mensaje principal. Sin embargo, también puede crear una estructura de mensajes completa desde cero, o incluso objetos individuales [Message](#) a mano. De hecho, también puede tomar una estructura existente y agregar nuevos objetos [Message](#), moverlos, etc. Esto compone una interfaz muy conveniente para cortar y gestionar mensajes MIME.

Puede crear una nueva estructura de objeto mediante la creación de una instancia de [Message](#), añadiendo accesorios y todos los encabezados necesarios manualmente. Para mensajes MIME sin embargo, el paquete [email](#) proporciona subclases convenientes para facilitar las cosas.

Descripción de las clases:

class email.mime.base.**MIMEBase** (*_maintype*, *_subtype*, *, *policy=compat32*, ***_params*)
Módulo: email.mime.base

Esta es la clase básica para todas las subclases específicas [Message](#) para MIME. Normalmente no creará instancias específicas de la [MIMEBase](#), aunque es posible. La [MIMEBase](#) se proporciona principalmente como una clase básica conveniente para subclases más específicas, apropiadas para MIME.

_maintype es el tipo principal *Content-Type* (por ejemplo *text* o *image*), y *_subtype* es el tipo menor *Content-Type* (por ejemplo *plain* o *gif*). *_params* es un diccionario de parámetro clave/valor y se pasa directamente a [Message.add_header](#).

Si se especifica *policy*, (por defecto, la directiva [compat32](#)) se pasará a [Message](#).

La clase [MIMEBase](#) siempre agrega un encabezado *Content-Type* (basado en *_maintype*, *_subtype* y *_params*), y un encabezado *MIME-Version* (siempre establecido en 1.0).

Distinto en la versión 3.6: Se ha añadido el parámetro *policy* de solo palabra clave.

class email.mime.nonmultipart.**MIMENonMultipart**
Módulo: email.mime.nonmultipart

Una subclase de [MIMEBase](#), es una clase base intermedia para los mensajes MIME que no son *multipart*. El propósito principal de esta clase es evitar el uso del método [attach\(\)](#), que solo tiene sentido para los mensajes *multipart*. Si se llama a [attach\(\)](#), se lanza una excepción [MultipartConversionError](#).

class email.mime.multipart.**MIMEMultipart** (*_subtype='mixed'*, *boundary=None*, *_subparts=None*, *, *policy=compat32*, ***_params*)
Módulo: email.mime.multipart

Una subclase de [MIMEBase](#), se trata de una clase base intermedia para los mensajes MIME que son *multipart*. El valor predeterminado opcional de *_subtype* es *mixed*, pero se puede utilizar para especificar el subtipo del mensaje. Se agregará un encabezado *Content-Type* de *multipart/_subtype* al objeto del mensaje. También se agregará un encabezado *MIME-Version*.

El *boundary* opcional es la cadena de límite multiparte. Cuando *None* (valor predeterminado), el límite se calcula cuando es necesario (por ejemplo, cuando se serializa el mensaje).

_subparts es una secuencia de subpartes iniciales para la carga útil. Debe ser posible convertir esta secuencia en una lista. Siempre puede adjuntar nuevas subpartes al mensaje mediante el método [Message.attach](#).

El valor predeterminado del argumento *policy* opcional es [compat32](#).

Los parámetros adicionales para el encabezado *Content-Type* se toman de los argumentos de palabra clave, o se pasan al argumento *_params*, que es un diccionario de palabras clave.

Distinto en la versión 3.6: Se ha añadido el parámetro *policy* de solo palabra clave.

```
class email.mime.application.MIMEApplication(_data, _subtype='octet-stream', _encoder=email.encoders.encode_base64, *,
                                             policy=compat32, **_params)
```

Módulo: `email.mime.application`

Una subclase de *MIMENonMultipart*, la clase *MIMEApplication* se utiliza para representar objetos de mensaje MIME de tipo principal *application*. *_data* es una cadena de caracteres que contiene los datos de bytes sin procesar. *_subtype* opcional especifica el subtipo MIME y el valor predeterminado es *octet-stream*.

_encoder opcional es un recurso invocable (es decir, una función) que realizará la codificación real de los datos para el transporte. Este invocable toma un argumento, que es la instancia *MIMEApplication*. Debe utilizar *get_payload()* y *set_payload()* para cambiar la carga útil a la forma codificada. También debe agregar cualquier *Content-Transfer-Encoding* u otros encabezados al objeto del mensaje según sea necesario. La codificación predeterminada es base64. Consulte el módulo *email.encoders* para obtener una lista de los codificadores integrados.

El valor predeterminado del argumento *policy* opcional es *compat32*.

_params se pasan directamente al constructor de la clase base.

Distinto en la versión 3.6: Se ha añadido el parámetro *policy* de solo palabra clave.

```
class email.mime.audio.MIMEAudio(_audiodata, _subtype=None, _encoder=email.encoders.encode_base64, *,
                                  policy=compat32, **_params)
```

Módulo: `email.mime.audio`

Una subclase de *MIMENonMultipart*, la clase *MIMEAudio* se utiliza para crear objetos de mensaje MIME de tipo principal *audio*. *_audiodata* es una cadena de caracteres que contiene los datos de audio sin procesar. Si estos datos pueden ser decodificados por el módulo estándar de Python *sndhdr*, entonces el subtipo se incluirá automáticamente en el encabezado *Content-Type*. De lo contrario, puede especificar explícitamente el subtipo de audio mediante el argumento *_subtype*. Si no se pudo adivinar el tipo secundario y no se ha proporcionado *_subtype*, se lanza *TypeError*.

_encoder opcional es un recurso invocable (es decir, una función) que realizará la codificación real de los datos de audio para el transporte. Este invocable toma un argumento, que es la instancia *MIMEAudio*. Debe utilizar *get_payload()* y *set_payload()* para cambiar la carga útil a la forma codificada. También debe agregar cualquier *Content-Transfer-Encoding* u otros encabezados al objeto del mensaje según sea necesario. La codificación predeterminada es base64. Consulte el módulo *email.encoders* para obtener una lista de los codificadores integrados.

El valor predeterminado del argumento *policy* opcional es *compat32*.

_params se pasan directamente al constructor de la clase base.

Distinto en la versión 3.6: Se ha añadido el parámetro *policy* de solo palabra clave.

```
class email.mime.image.MIMEImage(_imagedata, _subtype=None, _encoder=email.encoders.encode_base64, *,
                                  policy=compat32, **_params)
```

Módulo: `email.mime.image`

Una subclase de *MIMENonMultipart*, la clase *MIMEImage* se utiliza para crear objetos de mensaje MIME de tipo principal *image*. *_imagedata* es una cadena de caracteres que contiene los datos de imagen sin procesar. Si estos datos pueden ser decodificados por el módulo estándar de Python *imghdr*, entonces el subtipo se incluirá automáticamente en el encabezado *Content-Type*. De lo contrario, puede especificar explícitamente el subtipo de imagen mediante el argumento *_subtype*. Si no se pudo adivinar el tipo secundario y no se ha proporcionado *_subtype*, se lanza *TypeError*.

_encoder opcional es un recurso invocable (es decir, una función) que realizará la codificación real de los datos de imagen para el transporte. Este invocable toma un argumento, que es la instancia *MIMEImage*. Debe utilizar

`get_payload()` y `set_payload()` para cambiar la carga útil a la forma codificada. También debe agregar cualquier *Content-Transfer-Encoding* u otros encabezados al objeto del mensaje según sea necesario. La codificación predeterminada es base64. Consulte el módulo `email.encoders` para obtener una lista de los codificadores integrados.

El valor predeterminado del argumento *policy* opcional es `compat32`.

`_params` se pasan directamente al constructor `MIMEBase`.

Distinto en la versión 3.6: Se ha añadido el parámetro *policy* de solo palabra clave.

```
class email.mime.message.MIMEMessage (_msg, _subtype='rfc822', *, policy=compat32)
Módulo: email.mime.message
```

Una subclase de `MIMENonMultipart`, la clase `MIMEMessage` se utiliza para crear objetos MIME de tipo principal *message*. `_msg` se utiliza como la carga y debe ser una instancia de la clase `Message` (o una subclase de la misma), de lo contrario se lanza un `TypeError`.

`_subtype` opcional establece el subtipo del mensaje; por defecto es `rfc822`.

El valor predeterminado del argumento *policy* opcional es `compat32`.

Distinto en la versión 3.6: Se ha añadido el parámetro *policy* de solo palabra clave.

```
class email.mime.text.MIMEText (_text, _subtype='plain', _charset=None, *, policy=compat32)
Módulo: email.mime.text
```

Una subclase de `MIMENonMultipart`, la clase `MIMEText` se utiliza para crear objetos MIME de tipo principal *text*. `_text` es la cadena de caracteres de la carga útil. `_subtype` es el tipo menor y el valor predeterminado es `plain`. `_charset` es el paquete de caracteres del texto y se pasa como argumento al constructor `MIMENonMultipart`; el valor predeterminado es `us-ascii` si la cadena contiene sólo puntos de código `ascii`, y `utf-8` en caso contrario. El parámetro `_charset` acepta una cadena o una instancia `Charset`.

A menos que el argumento `_charset` se establezca explícitamente como `None`, el objeto `MIMEText` creado tendrá a la vez un encabezado *Content-Type* con un parámetro `charset`, y un encabezado *Content-Transfer-Encoding*. Esto significa que una llamada posterior a `set_payload` no dará lugar a una carga codificada, incluso si se pasa un conjunto de caracteres en el comando `set_payload`. Puede «restablecer» este comportamiento eliminando el encabezado *Content-Transfer-Encoding*, después de lo cual una llamada a `set_payload` codificará automáticamente la nueva carga útil (y agregará un nuevo encabezado *Content-Transfer-Encoding*).

El valor predeterminado del argumento *policy* opcional es `compat32`.

Distinto en la versión 3.5: `_charset` también acepta instancias `Charset`.

Distinto en la versión 3.6: Se ha añadido el parámetro *policy* de solo palabra clave.

19.1.11 email.header: Cabeceras internacionalizadas

Código fuente: `Lib/email/header.py`

Este módulo es parte de la API de email heredada (Compat 32). En la API actual, la codificación y decodificación de las cabeceras se gestiona de forma transparente por la API de tipo diccionario de la clase `EmailMessage`. Además de los usos del código heredado, este módulo puede ser útil en aplicaciones que necesiten controlar completamente el conjunto de caracteres usado cuando se codifican las cabeceras.

El resto del texto de esta sección es la documentación original del módulo.

RFC 2822 es el estándar base que describe el formato de los mensajes de correo electrónico. Deriva del estándar anterior **RFC 822**, cuyo uso se generalizó durante una época en la que la mayoría del correo electrónico se componía únicamente de caracteres ASCII. **RFC 2822** es una especificación que se escribió asumiendo que el correo electrónico contiene solo caracteres ASCII 7-bit.

Por supuesto, al haberse extendido el correo electrónico por todo el mundo, se ha internacionalizado, de forma que ahora pueden usarse los conjuntos de caracteres específicos de un idioma en los mensajes de correo electrónico.

El estándar base todavía requiere que los mensajes de correo electrónico sean transferidos usando solo caracteres ASCII 7-bit, así que se han escrito multitud de RFCs describiendo cómo codificar correos electrónicos que contengan caracteres no ASCII en formatos conforme a la [RFC 2822](#). Entre estas RFCs se incluyen [RFC 2045](#), [RFC 2046](#), [RFC 2047](#) y [RFC 2231](#). El paquete `email` soporta estos estándares en sus módulos `email.header` y `email.charset`.

Si quieres incluir caracteres no ASCII en tus cabeceras de correo electrónico, por ejemplo en los campos `Subject` o `To`, deberías usar la clase `Header` y asignar el campo del objeto `Message` a una instancia de `Header` en vez de usar una cadena de caracteres para el valor de la cabecera. Importa la clase `Header` del módulo `email.header`. Por ejemplo:

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
```

¿Has visto cómo hemos hecho que el campo `Subject` contuviera un carácter no ASCII? Lo hemos hecho creando una instancia de `Header` y pasándole el conjunto de caracteres en los que estaba codificado la cadena de bytes. Cuando la instancia de `Message` subsecuente se ha aplanado, el campo `Subject` se ha codificado en [RFC 2047](#) adecuadamente. Los lectores de correo que soportan MIME deberían mostrar esta cabecera usando el carácter ISO-8859-1 incrustado.

Aquí está la descripción de la clase `Header`:

class `email.header.Header` (*s=None*, *charset=None*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*, *errors='strict'*)

Crea una cabecera conforme a las especificaciones MIME que contiene cadenas de caracteres en diferentes conjuntos de caracteres.

El argumento opcional *s* es el valor inicial de la cabecera. Si es `None` (el valor por defecto), el valor inicial de la cabecera quedará sin asignar. Puedes añadirlo luego a la cabecera con llamadas al método `append()`. *s* debe ser una instancia de `bytes` o `str`, pero lee la documentación de `append()` para conocer los detalles semánticos.

El argumento opcional *charset* sirve para dos propósitos: tiene el mismo significado que el argumento *charset* en el método `append()`. También asigna el conjunto de caracteres por defecto para todas las llamadas subsecuentes a `append()` que omitan el argumento *charset*. Si no se proporciona *charset* en el constructor (por defecto), el conjunto de caracteres `us-ascii` se usa tanto para el conjunto de caracteres inicial de *s* como por defecto para las llamadas subsecuentes a `append()`.

La longitud de línea máxima puede especificarse explícitamente con *maxlinelen*. Para dividir la primera línea en un valor más corto (teniendo en cuenta los campos de la cabecera que no están incluidos en *s*, por ejemplo `Subject`), pasar el nombre del campo en *header_name*. El valor por defecto de *maxlinelen* es 76, y el valor por defecto de *header_name* es `None`, lo que quiere decir que no se tiene en cuenta para una cabecera larga dividida.

El argumento opcional *continuation_ws* debe ser conforme a las normas de espacios en blanco plegables de la [RFC 2822](#), y normalmente es o un espacio o un carácter tabulador. Este carácter se antepone delante de las líneas continuadas. El valor por defecto de *continuation_ws* es un solo espacio.

El argumento opcional *errors* se pasa directamente a través del método `append()`.

append (*s*, *charset=None*, *errors='strict'*)

Añade la cadena de caracteres *s* a la cabecera MIME.

El argumento opcional *charset*, si se proporciona, debe ser una instancia de `Charset` (ver `email.charset`) o el nombre de un conjunto de datos, que deberá ser convertido a una instancia de `Charset`. Un valor de `None` (el valor por defecto) significa que se usará el *charset* dado en el constructor.

s puede ser una instancia de `bytes` o de `str`. Si es una instancia de `bytes`, entonces *charset* es la codificación de esa cadena de bytes, y se lanzará un `UnicodeError` si la cadena de caracteres no puede decodificarse con ese conjunto de caracteres.

Si *s* es una instancia de `str`, entonces *charset* es una sugerencia que especifica el conjunto de caracteres usando en la cadena de caracteres.

En cualquier caso, cuando se produce una cabecera conforme a la [RFC 2822](#) usando las reglas de la [RFC 2047](#), la cadena de caracteres se codificará usando el códec de salida del conjunto de caracteres. Si la cadena de caracteres no puede codificarse usando el códec de salida, se lanzará un `UnicodeError`.

El argumento opcional *errors* se pasa como el argumento de errores para la llamada de decodificación si *s* es una cadena de bits.

encode (*splitchars*=', \t', *maxlinelen*=None, *linesep*='\n')

Codifica un mensaje de la cabecera en un formato conforme a RFC, posiblemente envolviendo las líneas largas y encapsulando las partes no ASCII en base64 o en codificaciones imprimibles entrecomilladas.

El argumento opcional *splitchars* es una cadena de caracteres que contiene caracteres a los que el algoritmo de separación debería asignar espacio extra durante la encapsulación de la cabecera normal. Esto da un basto soporte a los saltos sintácticos de alto nivel de la [RFC 2822](#): los puntos de separación precedidos por un caracter separador tienen preferencia durante la separación de la línea, con preferencia de caracteres en el orden en el que aparecen en la cadena de caracteres. El espacio y el tabulador pueden incluirse en la cadena de caracteres para indicar si se debería dar preferencia a uno sobre el otro como punto de separación cuando no aparezcan otros caracteres separadores en la línea que se está dividiendo. Los caracteres separadores no afectan a las líneas codificadas de la [RFC 2047](#).

maxlinelen, si se proporciona, sobrescribe el valor de longitud de línea máxima para la instancia.

linesep especifica los caracteres usados para separar las líneas de la cabecera plegada. Su valor por defecto es el valor más útil para el código de una aplicación Python (`\n`), pero se puede especificar `\r\n` para producir cabeceras con separadores de línea conforme a RFCs.

Distinto en la versión 3.2: Argumento *linesep* añadido.

La clase `Header` también proporciona una serie de métodos para soportar operaciones estándar y funciones incorporadas.

__str__ ()

Retorna una aproximación de `Header` como una cadena de caracteres, usando una longitud de línea ilimitada. Todas las piezas se convierten a unicode utilizando la codificación especificada y unidas adecuadamente. Todas las piezas con un conjunto de caracteres 'unknown-8bit' se decodifican como ASCII usando el gestor de errores de 'replace'.

Distinto en la versión 3.2: Añadida gestión del conjunto de caracteres 'unknown-8bit'.

__eq__ (other)

Este método permite comparar si dos instancias de `Header` son iguales.

__ne__ (other)

Este método permite comparar si dos instancias de `Header` no son iguales.

El módulo `email.header` También proporciona las prácticas funciones que se indican a continuación.

`email.header.decode_header` (*header*)

Decodifica el valor de un mensaje de la cabecera sin convertir el conjunto de caracteres. El valor de la cabecera está en *header*.

Esta función retorna una lista de duplas (`decoded_string`, `charset`) que contiene cada una de las partes decodificadas de la cabecera. *charset* será `None` para las partes no codificadas de la cabecera, de lo contrario será una cadena de caracteres en minúscula con el nombre del conjunto de caracteres especificado en la cadena de caracteres codificada.

Aquí va un ejemplo:

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xf6stal', 'iso-8859-1')]
```


`email.header.make_header(decoded_seq, maxlinelen=None, header_name=None, continuation_ws='')`

Crea una instancia de `Header` a partir de una secuencia de duplas como las retornadas por `decode_header()`.

`decode_header()` toma el valor de una cadena de caracteres de la cabecera y retorna una secuencia de duplas con el formato `(decoded_string, charset)`, donde `charset` es el nombre del conjunto de caracteres.

Esta función toma una de esas secuencias de duplas y retorna una instancia de `Header`. Los argumentos opcionales `maxlinelen`, `header_name` y `continuation_ws` son como los del constructor `Header`.

19.1.12 email.charset: Representa conjunto de caracteres

Código fuente: [Lib/email/charset.py](#)

Este módulo es parte de la API de email heredada (Compat 32). En la nueva API solo se usa la tabla de alias.

El texto restante de esta sección corresponde a la documentación original del módulo.

Este módulo proporciona una clase `Charset` para representar conjuntos de caracteres y conversiones de conjuntos de caracteres en mensajes de correo electrónico, así como un registro de conjuntos de caracteres y varios métodos de conveniencia para manipular este registro. Las instancias de `Charset` se utilizan en varios otros módulos dentro del paquete `email`.

Importe esta clase desde el módulo `email.charset`.

class `email.charset.Charset(input_charset=DEFAULT_CHARSET)`

Asigna conjuntos de caracteres a sus propiedades de correo electrónico.

Esta clase proporciona información sobre los requisitos impuestos al correo electrónico para un conjunto de caracteres específico. También proporciona rutinas de conveniencia para convertir entre juegos de caracteres, dada la disponibilidad de los códecs aplicables. Dado un conjunto de caracteres, hará todo lo posible para proporcionar información sobre cómo utilizar ese conjunto de caracteres en un mensaje de correo electrónico de forma compatible con RFC.

Ciertos conjuntos de caracteres deben codificarse con quoted-printable o base64 cuando se usan en encabezados o cuerpos de correo electrónico. Ciertos conjuntos de caracteres deben convertirse directamente y no están permitidos en el correo electrónico.

Opcional `input_charset` es como se describe a continuación; siempre se convierte a minúsculas. Después de que el alias sea normalizado también se utiliza como una búsqueda en el registro de conjuntos de caracteres para averiguar la codificación del encabezado, codificación de cuerpo, y códec de conversión de salida que se usarán para el conjunto de caracteres. Por ejemplo, si `input_charset` es `iso-8859-1`, los encabezados y cuerpos se codificarán mediante quoted-printable y no es necesario ningún códec de conversión de salida. Si `input_charset` es `us-ascii`, los encabezados se codificarán con base64, los cuerpos no se codificarán, pero el texto de salida se convertirá del conjunto de caracteres `us-ascii` al conjunto de caracteres `iso-8859-1`.

Las instancias `Charset` tienen los siguientes atributos de datos:

input_charset

El conjunto de caracteres inicial especificado. Los alias comunes se convierten a sus nombres de correo electrónico *Official* (por ejemplo, `latin_1` se convierte a `"iso-8859-1"`). El valor predeterminado es `"us-ascii"` de 7 bits.

header_encoding

Si el conjunto de caracteres debe codificarse antes de que pueda usarse en un encabezado de correo electrónico, este atributo se establecerá a `Charset.QP` (para quoted-printable), `Charset.BASE64` (para codificación base64), o `Charset.SHORTEST` para la más codificación más corta QP o BASE64. De lo contrario será `None`.

body_encoding

Igual que *header_encoding*, pero describe la codificación del cuerpo del mensaje de correo, que de hecho puede ser diferente a la codificación del encabezado. *Charset.SHORTEST* no está permitido para *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

El nombre del códec de Python usado para convertir el *input_charset* a Unicode. Si no es necesario un códec de conversión, este atributo será `None`.

output_codec

El nombre del códec de Python usado para convertir Unicode a *output_charset*. Si no es necesario un códec de conversión, este atributo tendrá el mismo valor que *input_codec*.

Las instancias *Charset* además tienen los siguientes métodos:

get_body_encoding()

Retorna la codificación de transferencia de contenido usada para la codificación del cuerpo.

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.

Retorna la cadena `quoted-printable` si *body_encoding* es `QP`, retorna la cadena `base64` si *body_encoding* es `BASE64`, y retorna la cadena `7bit` en caso contrario.

get_output_charset()

Return the output character set.

Este es el atributo *output_charset* si no es `None`, en caso contrario es *input_charset*.

header_encode(string)

Header-encode the string *string*.

El tipo de codificación (`base64` o `quoted-printable`) se basará en el atributo *header_encoding*.

header_encode_lines(string, maxlengths)

Codifica como encabezado *string* convirtiéndolo primero a bytes.

Es similar a *header_encode()* excepto que la cadena se ajusta a las longitudes máximas indicadas en el argumento *maxlengths*, el cual debe ser un iterador: cada elemento retornado por este iterador proporcionará la siguiente longitud máxima de línea.

body_encode(string)

Codifica como Cuerpo la cadena *string*.

El tipo de codificación (`base64` o `quoted-printable`) se basará en el atributo *body_encoding*.

La clase *Charset* también proporciona una serie de métodos para soportar operaciones estándar y funciones integradas.

__str__()

Returns *input_charset* as a string coerced to lower case. *__repr__()* is an alias for *__str__()*.

__eq__(other)

Este método le permite comparar dos instancias *Charset* por igualdad.

__ne__(other)

Este método le permite comparar dos instancias *Charset* por desigualdad.

El módulo *email.charset* provee además las siguientes funciones para agregar nuevas entradas al conjunto global de caracteres, alias y registros de códec:

`email.charset.add_charset(charset, header_enc=None, body_enc=None, output_charset=None)`

Añade propiedades de carácter al registro global.

charset es el conjunto de caracteres de entrada, y debe ser el nombre canónico del conjunto de caracteres.

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.

Opcional *output_charset* es el conjunto de caracteres en el que debe estar la salida. Las conversiones proceden del conjunto de caracteres de entrada, a Unicode, al conjunto de caracteres de salida cuando se llama al método `Charset.convert()`. El valor predeterminado es la salida en el mismo conjunto de caracteres que la entrada.

Tanto *input_charset* y *output_charset* deben tener entradas de códec Unicode en el conjunto de caracteres del módulo para la asignación del códec; use `add_codec()` para agregar códec que el módulo no conozca. Consulte la documentación del módulo `codecs` para más información.

The global character set registry is kept in the module global dictionary `CHARSETS`.

`email.charset.add_alias(alias, canonical)`

Añade un alias al conjunto de caracteres. *alias* es el nombre del alias, p. ej. *latin-1*. *canonical* es el nombre canónico del conjunto de caracteres, p. ej. *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 es el nombre canónico de un conjunto de caracteres. *codecname* es el nombre de un códec de Python, según corresponda para el segundo argumento del método `str` de `encode()`.

19.1.13 email.encoders: Codificadores

Código fuente: [Lib/email/encoders.py](#)

Este módulo forma parte de la anterior API de correo electrónico (Compat32). En la nueva API, la funcionalidad la proporciona el parámetro *cte* del método `set_content()`.

Este módulo está obsoleto (deprecated) en Python 3. Las funciones que aparecen aquí no deberían ser llamadas explícitamente ya que la clase `MIMEText` establece el tipo de contenido y el encabezado CTE utilizando los valores del `_subtype` y del `_charset` que se pasan cuando se instancia esa clase.

El texto que viene a continuación corresponde a la documentación original del módulo.

Cuando se crean objetos `Message` desde 0, a menudo se necesita codificar el contenido del mensaje para transportarlo a través de servidores de correo electrónico adecuados. Esto es así especialmente para el tipo de mensajes *image/** y *text/** que contienen datos binarios.

El paquete `email` proporciona algunos codificadores adecuados en su módulo `encoders`. Estos codificadores son en realidad utilizados por los constructores de las clases `MIMEAudio` y `MIMEImage` para proporcionar codificadores por defecto. Todas las funciones de codificación tienen exactamente un argumento, el mensaje a codificar. Normalmente extraen el contenido, lo codifican y borran el contenido para introducir el nuevo contenido codificado. También deberían marcar el encabezado *Content-Transfer-Encoding* como apropiado.

Ten en cuenta que estas funciones no sirven para un mensaje con múltiples partes. En lugar de aplicarlo al mensaje completo, las funciones deben aplicarse a cada subparte individual. Si se pasa un mensaje de múltiples partes como argumento se activará un mensaje de error `TypeError`.

A continuación, una lista de las funciones de codificación facilitadas:

`email.encoders.encode_quopri(msg)`

Codifica el contenido en formularios entrecomillados e imprimibles y marca el encabezado

Content-Transfer-Encoding como `quoted-printable`¹. Es un buen codificador para usar cuando la mayoría del contenido son datos imprimibles normales pero hay algún dato que no es imprimible.

`email.encoders.encode_base64(msg)`

Codifica el contenido en un formulario base64 y marca el encabezado *Content-Transfer-Encoding* como base64. Esta codificación es buena cuando la mayoría del contenido son datos no imprimibles ya que es un formulario más compacto que formularios entrecomillados e imprimibles. La desventaja es que incluye el texto que no es leíble por los humanos.

`email.encoders.encode_7or8bit(msg)`

Esto, en realidad, no modifica el contenido del mensaje, pero fija el encabezado *Content-Transfer-Encoding* a 7bit u 8bit, lo que considere más adecuado en función del contenido del mensaje.

`email.encoders.encode_noop(msg)`

Esto no hace nada; ni siquiera fija el encabezado *Content-Transfer-Encoding*.

Notas

19.1.14 `email.utils`: Utilidades misceláneas

Código fuente: [Lib/email/utils.py](#)

Existen varias funciones útiles proporcionadas en el módulo `email.utils`:

`email.utils.localtime(dt=None)`

Retorna el tiempo local como un objeto `datetime` consciente. Si se llama sin argumentos, retorna el tiempo actual. De lo contrario, el argumento `dt` debe ser una instancia `datetime`, y es convertida a la zona horaria local de acuerdo a la base de datos de zonas horarias del sistema. Si `dt` es naíf (*naif*, es decir, `dt.tzinfo` es `None`), se asume que está en tiempo local. En este caso, un valor positivo o cero para `isdst` hace que `localtime` asuma inicialmente que el horario de verano está o no (respectivamente) en efecto para el tiempo especificado. Un valor negativo para `isdst` hace que el `localtime` intente determinar si el horario de verano está en efecto para el tiempo especificado.

Nuevo en la versión 3.3.

`email.utils.make_msgid(idstring=None, domain=None)`

Retorna una cadena de caracteres apropiada para una cabecera *Message-ID* que cumpla con [RFC 2822](#). Si se especifica un `idstring` opcional, es una cadena de caracteres utilizada para reforzar la unicidad del identificador del mensaje. Si se especifica un `domain` opcional provee la porción del `msgid` después de “@”. El predeterminado es el hostname local. Normalmente no es necesario especificar este valor, pero puede ser útil en algunos casos, como cuando se está construyendo un sistema distribuido que utiliza un nombre de dominio consistente a lo largo de varios ordenadores.

Distinto en la versión 3.2: Se añadió la palabra clave `domain`.

Las funciones que quedan son parte de la API de correo electrónico heredada (`Compat32`). No hay necesidad de utilizarlas directamente con el nuevo API, dado que el análisis sintáctico y formateo que proporcionan es realizado automáticamente por el mecanismo de análisis sintáctico de la nueva API.

`email.utils.quote(str)`

Retorna una nueva cadena de caracteres con barras invertidas (\) en `str` reemplazado por dos barras invertidas, y comillas dobles reemplazadas por barra invertida seguido de comillas dobles.

`email.utils.unquote(str)`

Retorna una nueva cadena de caracteres que es una versión *unquoted* de `str`. Si `str` comienza y termina con comillas dobles, éstas son eliminadas. De igual manera si `str` empieza y termina con comillas angulares (< y >), éstas son eliminadas.

¹ El codificado con `encode_quopri()` también codifica todas las tabulaciones y caracteres de espacios en los datos.

`email.utils.parseaddr(address, *, strict=True)`

Interpreta *address* – la cual debe ser el valor de un campo que contenga un campo tal como *To* o *Cc* – para separarlo en sus componentes *realname* y *email address*. Retorna una tupla de información, a menos que la interpretación falle, en cuyo caso una 2-tupla de ('', '') es retornada.

If *strict* is true, use a strict parser which rejects malformed inputs.

Distinto en la versión 3.8.20: Add *strict* optional parameter and reject malformed inputs by default.

`email.utils.formataddr(pair, charset='utf-8')`

El inverso de `parseaddr()`, este toma una 2-tupla de la forma (*realname*, *real_email_address*) y retorna una cadena de caracteres válido para una cabecera *To* o *Cc*. Si el primer elemento de *pair* es falso, entonces el segundo elemento es retornado sin modificación.

El *charset* opcional es el conjunto de caracteres que será usado en la codificación [RFC 2047](#) del *real_name* si el *real_name* contiene caracteres que no sean ASCII. Puede ser una instancia de *str* o *Charset*. El valor predeterminado es *utf-8*.

Distinto en la versión 3.3: Se añadió la opción *charset*.

`email.utils.getaddresses(fieldvalues, *, strict=True)`

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()`.

If *strict* is true, use a strict parser which rejects malformed inputs.

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)
```

Distinto en la versión 3.8.20: Add *strict* optional parameter and reject malformed inputs by default.

`email.utils.parsedate(date)`

Intenta interpretar una fecha de acuerdo a las reglas en [RFC 2822](#). Sin embargo, algunos clientes de correo no siguen ese formato como está especificado, por lo cual `parsedate()` intenta adivinar correctamente en esos casos. *date* es una secuencia de caracteres que contiene una fecha [RFC 2822](#) como "Mon, 20 Nov 1995 19:12:08 -0500". Si tiene éxito en interpretar la fecha, `parsedate()` retorna una 9-tupla que puede ser pasada directamente a `time.mktime()`; de lo contrario *None* es retornado. Observar que los índices 6, 7 y 8 de la tupla resultante no son utilizables.

`email.utils.parsedate_tz(date)`

Realiza la misma función que `parsedate()`, pero retorna *None* o una 10-tupla; los primeros 9 elementos forman una tupla que puede ser pasada directamente a `time.mktime()`, y el décimo es el desfase de la zona horaria de la fecha con respecto a UTC (que es el término oficial para Greenwich Mean Time)¹. Si la cadena de caracteres de entrada no tiene zona horaria, el último elemento de la tupla retornada es 0, el cual representa UTC. Nótese que los índices 6, 7 y 8 de la tupla resultante no son utilizables.

`email.utils.parsedate_to_datetime(date)`

El inverso de `format_datetime()`. Realiza la misma función que `parsedate()`, pero al tener éxito retorna un *datetime*. Si la fecha de entrada tiene una zona horaria de -0000, el *datetime* será un *datetime* naíf, y si la fecha cumple con los RFCs representará un tiempo en UTC pero sin indicación de la zona horaria originaria actual del mensaje de donde proviene la fecha. Si la fecha de entrada tiene cualquier otro desfase de zona horaria válida, el *datetime* será un *datetime* consiente con el correspondiente `timezone.tzinfo`.

Nuevo en la versión 3.3.

¹ Nótese que el signo del desfase de la zona horaria es opuesto al signo de la variable `time.timezone` para la misma zona horaria; este último sigue el estándar POSIX mientras que este módulo sigue [RFC 2822](#).

`email.utils.mktime_tz (tuple)`

Cambia una 10-tupla, como la retornada por `parsedate_tz()` a una marca de tiempo UTC (segundos desde la Época). Si la zona horaria en la tupla es `None`, asume el tiempo local.

`email.utils.formatdate (timeval=None, localtime=False, usegmt=False)`

Retorna una fecha como una cadena de caracteres de acuerdo a [RFC 2822](#), por ejemplo:

`Fri, 09 Nov 2001 01:08:47 -0000`

`timeval` opcional, es un valor de tiempo de punto flotante como el aceptado por `time.gmtime()` y `time.localtime()`. Si no es dado, el tiempo actual es usado.

`localtime` opcional, es una bandera que cuando es `True`, interpreta `timeval` y retorna una fecha relativa a la zona horaria local en lugar de UTC, tomando apropiadamente en cuenta el horario de verano. El valor predeterminado es `False` con lo cual UTC es utilizado.

`usegmt` opcional, es una bandera que cuando es `True` retorna una fecha como cadena de caracteres con la zona horaria como una cadena de caracteres ascii GMT, en lugar de un numérico `-0000`. Esto es necesario para algunos protocolos (como HTTP). Sólo aplica cuando `localtime` es `False`. El valor predeterminado es `False`.

`email.utils.format_datetime (dt, usegmt=False)`

Similar a `formatdate`, pero la entrada es una instancia `datetime`. Si es un `datetime` naïf, se asume ser «UTC sin información sobre la zona horaria de origen», y el `-0000` convencional es usado para la zona horaria. Si es un `datetime` consciente entonces el desfase numérico de la zona horaria es utilizado. Si es una zona horaria consciente con un desfase cero, entonces `usegmt` puede ser `True`, en cuyo caso la cadena de caracteres GMT es utilizada en lugar del desfase numérico de zona horaria. Esto provee una manera de generar cabeceras de fecha HTTP conforme estándares.

Nuevo en la versión 3.3.

`email.utils.decode_rfc2231 (s)`

Decodifica la cadena de caracteres `s` de acuerdo a [RFC 2231](#).

`email.utils.encode_rfc2231 (s, charset=None, language=None)`

Codifica la cadena de caracteres `s` de acuerdo a [RFC 2231](#). `charset` y `language` opcionales, si son dados es el conjunto de caracteres y nombre del lenguaje a utilizar. Si ninguno es dado, `s` es retornado sin modificar. Si `charset` es dado pero `language` no, la cadena de caracteres es codificada usando la cadena de caracteres vacía para `language`.

`email.utils.collapse_rfc2231_value (value, errors='replace', fallback_charset='us-ascii')`

Cuando un parámetro de cabecera está codificado en formato [RFC 2231](#), `Message.get_param` puede retornar una 3-tupla conteniendo el conjunto de caracteres, lenguaje y valor. `collapse_rfc2231_value()` convierte esto en una cadena de caracteres unicode. El parámetro opcional `errors` es pasado al argumento `errors` del método `encode()` de `str`; de manera predeterminada recae en `'replace'`. `fallback_charset` opcional, especifica el conjunto de caracteres a utilizar si el especificado en la cabecera [RFC 2231](#) no es conocido por Python; su valor predeterminado es `'us-ascii'`.

Por conveniencia, si el `value` pasado a `collapse_rfc2231_value()` no es una tupla, debería ser una cadena de caracteres y se retorna sin citar.

`email.utils.decode_params (params)`

Decodifica la lista de parámetros de acuerdo a [RFC 2231](#). `params` es una secuencia de 2-tuplas conteniendo elementos de la forma `(content-type, string-value)`.

19.1.15 `email.iterators`: Iteradores

Código fuente: [Lib/email/iterators.py](#)

Iterar sobre un árbol de objetos mensaje es bastante fácil con el método `Message.walk`. El módulo `email.iterators` proporciona algunos iteradores útiles de más alto nivel sobre árboles de objetos mensaje.

`email.iterators.body_line_iterator(msg, decode=False)`

Itera sobre todas las cargas útiles de todas las subpartes de `msg`, retornando las cargas útiles en cadenas de caracteres línea por línea. Descarta todas las cabeceras de las subpartes, y descarta cualquier subparte con una carga útil que no sea una cadena de caracteres de Python. Esto de alguna forma es equivalente a leer la representación en texto plano del mensaje desde un fichero usando `readline()`, descartando todas las cabeceras intermedias.

El argumento opcional `decode` se pasa a través de `Message.get_payload`.

`email.iterators.typed_subpart_iterator(msg, maintype='text', subtype=None)`

Itera sobre todas las subpartes de `msg`, retornando solo las subpartes que coincidan el tipo MIME especificado por `maintype` y `subtype`.

Note que `subtype` es opcional; si se omite, entonces se comprobará si el tipo MIME de las subpartes coincide con el tipo principal solamente. `maintype` es opcional también; su valor por defecto es `text`.

Por tanto, por defecto `typed_subpart_iterator()` retorna cada parte que tenga un tipo MIME `text/*`.

La siguiente función se ha añadido como una útil herramienta de depuración. *No* debe ser considerada parte de la interfaz pública soportada para este paquete.

`email.iterators._structure(msg, fp=None, level=0, include_default=False)`

Imprime una representación con sangrías de los tipos del contenido de la estructura de objetos mensaje. Por ejemplo:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
  text/plain
  text/plain
  multipart/digest
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
  text/plain
```

El argumento opcional `fp` es un objeto similar a un fichero en el que imprimir la salida. Debe ser adecuado para la función de Python `print()`. Se usa `level` internamente. `include_default`, si tiene su valor a verdadero, imprime el tipo por defecto también.

Ver también:

Módulo `smtplib` Cliente SMTP (Protocolo simple de transporte de correo)

Módulo `poplib` Cliente POP (Protocolo de oficina postal)

Módulo `imaplib` Cliente IMAP (Protocolo de acceso a mensajes de Internet)

Módulo `nntplib` Cliente NNTP (Protocolo de transporte de noticias de red)

Módulo *mailbox* Herramientas para crear, leer y administrar colecciones de mensajes en disco utilizando una variedad de formatos estándar.

Módulo *smtpd* Marco del servidor SMTP (principalmente útil para pruebas)

19.2 json — Codificador y decodificador JSON

Código fuente: `Lib/json/__init__.py`

JSON (JavaScript Object Notation), especificado por [rfc:"7159"](#) (que obsoleta [RFC 4627](#)) y por [ECMA-404](#), es un formato ligero de intercambio de datos inspirado por la sintaxis literal de objetos en JavaScript (aunque no es un subconjunto estricto de JavaScript¹).

Advertencia: Be cautious when parsing JSON data from untrusted sources. A malicious JSON string may cause the decoder to consume considerable CPU and memory resources. Limiting the size of data to be parsed is recommended.

json expone una API familiar a los usuarios de los módulos de la biblioteca estándar *marshal* y *pickle*.

Codificación de jerarquías básicas de objetos de Python:

```
>>> 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('\u1234'))
'"\\u1234"'
>>> print(json.dumps('\\\\'))
'"\\\\"'
>>> print(json.dumps({'c': 0, 'b': 0, 'a': 0}, sort_keys=True))
'{"a": 0, "b": 0, "c": 0}'
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

Codificación compacta:

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

Impresión linda:

```
>>> import json
>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

Decodificación JSON:

¹ Como se indica en la errata para RFC 7159, JSON permite caracteres literales U+2028 (SEPARADOR DE LINEA) y U+2029 (SEPARADOR DE PÁRRAFO) en cadenas, mientras que JavaScript (a partir de ECMAScript Edición 5.1) no lo hace.

```
>>> import json
>>> json.loads('{"foo", {"bar":["baz", null, 1.0, 2]}}')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\\"foo\\\"bar\"')
'foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']
```

Decodificación personalizada de objetos JSON:

```
>>> 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')
```

Extendiendo *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', ', ', ']']
```

Usando *json.tool* desde el shell para validación e impresión con sangría:

```
$ echo '{"json":"obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

Consulte *Interfaz de línea de comandos* para obtener documentación detallada.

Nota: JSON es un subconjunto de [YAML 1.2](#). El JSON producido por la configuración predeterminada de este módulo (en particular, el valor predeterminado *separators*) también es un subconjunto de [YAML 1.0](#) y [1.1](#). Por lo tanto, este módulo también se puede utilizar como un serializador [YAML](#).

Nota: Los codificadores y decodificadores de este módulo conservan el orden de entrada y salida de forma predeterminada. El orden solo se pierde si los contenedores subyacentes no están ordenados.

Antes de Python 3.7, no se garantizaba que `dict` fuera ordenado, por lo que las entradas y salidas se mezclaban a menos que `collections.OrderedDict` se solicitara específicamente. Comenzando con Python 3.7, la clase regular `dict` conserva el orden, por lo que ya no es necesario especificar `collections.OrderedDict` para la generación y análisis JSON.

19.2.1 Uso básico

`json.dump(obj, fp, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)`
Serializa `obj` como una secuencia con formato JSON a `fp` (a `.write()` -supporting *file-like object*) usando esto [conversion table](#).

Si `skipkeys` es verdadero (predeterminado: `False`), entonces las llaves del dict que no son de un tipo básico (`str`, `int`, `float`, `bool`, `None`) se omitirán en lugar de generar un `TypeError`.

El módulo `json` siempre produce objetos `str`, no objetos `bytes`. Por lo tanto, `fp.write()` debe admitir `str` como entrada.

Si `ensure_ascii` es verdadero (el valor predeterminado), se garantiza que la salida tendrá todos los caracteres entrantes no ASCII escapados. Si `ensure_ascii` es falso, estos caracteres se mostrarán tal cual.

Si `check_circular` es falso (predeterminado: `True`), se omitirá la verificación de referencia circular para los tipos de contenedor y una referencia circular dará como resultado `OverflowError` (o peor).

Si `allow_nan` es falso (predeterminado: `True`), entonces serializar los valores fuera de rango `float` (`nan`, `inf`, `-inf`) provocará un `ValueError` en estricto cumplimiento de la especificación JSON. Si `allow_nan` es verdadero, se utilizarán sus equivalentes de JavaScript (`NaN`, `Infinity`, `-Infinity`).

Si `indent` es un entero no negativo o una cadena, los elementos del arreglo JSON y los miembros del objeto se imprimirán con ese nivel de sangría. Un nivel de sangría de 0, negativo o `""` solo insertará nuevas líneas. `None` (el valor predeterminado) selecciona la representación más compacta. El uso de una sangría de entero positivo agrega sangrías de muchos espacios por nivel. Si `indent` es una cadena (como `"\t"`), esa cadena se usa para agregarle sangría a cada nivel.

Distinto en la versión 3.2: Permite cadenas de caracteres para `indent` además de enteros.

Si se especifica, `separators` debe ser una tupla (`separador_elemento`, `separador_llave`). El valor predeterminado es (`' ', ' ', ': '`) si `indent` es `None` y (`' ', ' ', ': '`) de lo contrario. Para obtener la representación JSON más compacta, debe especificar (`' ', ' ', ': '`) para eliminar espacios en blanco.

Distinto en la versión 3.4: Usa (`' ', ' ', ': '`) como predeterminado si `indent` no es `None`.

Si se especifica, `default` debería ser una función que se llama para objetos que de otro modo no se pueden serializar. Debería retornar una versión codificable JSON del objeto o generar un `TypeError`. Si no se especifica, produce `TypeError`.

Si `sort_keys` es verdadero (predeterminado: `False`), la salida de los diccionarios se ordenará por llave.

Para usar una subclase personalizada de `JSONEncoder` (por ejemplo, una que sobre escriba el método `default()` para serializar tipos adicionales), se especifica mediante el argumento por palabra clave `cls`; de lo contrario se usa `JSONEncoder`.

Distinto en la versión 3.6: Todos los parámetros opcionales son ahora *palabra-clave-solamente*.

Nota: A diferencia de `pickle` y `marshal`, JSON no es un protocolo enmarcado, por lo que intentar serializar varios objetos con llamadas repetidas a `dump()` utilizando el mismo `fp` dará como resultado un archivo JSON no válido.

`json.dumps(obj, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)`
Serializa `obj` en un `str` con formato JSON usando esta [conversion table](#). Los argumentos tienen el mismo significado que en `dump()`.

Nota: Las llaves de los pares llave/valor de JSON siempre son del tipo `str`. Cuando un diccionario se convierte en JSON, todas las llaves del diccionario se convierten en cadenas. Como resultado de esto, si un diccionario se convierte en JSON y, a continuación, se convierte nuevamente en un diccionario, el diccionario puede que no sea igual al original. Es decir, `loads(dumps(x)) != x` si `x` tiene llaves que no son de tipo cadena de caracteres.

`json.load(fp, *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)`

Deserializa `fp` (un *text file* o *binary file* que soporte `.read()` y que contiene un documento JSON) a un objeto Python usando esta *conversion table*.

`object_hook` es una función opcional a la que se llamará con el resultado de cualquier literal de objeto decodificado (un *dict*). El valor de retorno de `object_hook` se utilizará en lugar de *dict*. Esta característica se puede utilizar para implementar decodificadores personalizados (por ejemplo, la sugerencia de clase *JSON-RPC*).

`object_pairs_hook` es una función opcional a la que se llamará con el resultado de cualquier literal de objeto decodificado con una lista ordenada de pares. El valor de retorno de `object_pairs_hook` se utilizará en lugar de *dict*. Esta característica se puede utilizar para implementar decodificadores personalizados. Si también se define `object_hook`, el `object_pairs_hook` tiene prioridad.

Distinto en la versión 3.1: Soporte agregado para `object_pairs_hook`.

`parse_float`, si se especifica, se llamará con la cadena de cada flotante JSON que se va a decodificar. De forma predeterminada, esto es equivalente a `float(num_str)`. Esto se puede utilizar para hacer uso de otro tipo de datos o analizador para flotantes JSON (por ejemplo `decimal.Decimal`).

`parse_int`, si se especifica, se llamará con la cadena de cada entero JSON que se va a decodificar. De forma predeterminada, esto es equivalente a `int(num_str)`. Esto se puede utilizar para hacer uso de otro tipo de datos o analizador para enteros JSON (por ejemplo `float`).

Distinto en la versión 3.8.14: The default `parse_int` of `int()` now limits the maximum length of the integer string via the interpreter's *integer string conversion length limitation* to help avoid denial of service attacks.

`parse_constant`, si se especifica, se llamará con una de las siguientes cadenas: `'-Infinity'`, `'Infinity'`, `'NaN'`. Esto se puede utilizar para generar una excepción si se encuentran números JSON inválidos.

Distinto en la versión 3.1: `parse_constant` ya no es llamado en “null”, “true”, “false”.

Para utilizar una subclase personalizada de *JSONDecoder*, especificarlo con el argumento por llave `cls`; de lo contrario, se utilizará *JSONDecoder*. Se pasarán argumentos adicionales de palabra llave al constructor de la clase.

Si los datos que se deserializan no constituyen un documento JSON válido, se generará un *JSONDecodeError*.

Distinto en la versión 3.6: Todos los parámetros opcionales son ahora *palabra-clave-solamente*.

Distinto en la versión 3.6: `fp` ahora puede ser un *binary file*. La codificación de entrada debe ser UTF-8, UTF-16 o UTF-32.

`json.loads(s, *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)`

Deserializa `s` (una instancia *str*, *bytes* o *bytearray* que contiene un documento JSON) en un objeto Python mediante esta *conversion table*.

Los otros argumentos tienen el mismo significado que en `load()`, excepto `encoding` que se omite y está en desuso desde Python 3.1.

Si los datos que se deserializan no constituyen un documento JSON válido, se generará un *JSONDecodeError*.

Deprecated since version 3.1, will be removed in version 3.9: `encoding` argumento de palabra llave.

Distinto en la versión 3.6: `s` ahora puede ser de tipo *bytes* o *bytearray*. La codificación de entrada debe ser UTF-8, UTF-16 o UTF-32.

19.2.2 Codificadores y Decodificadores

class `json.JSONDecoder`(*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

Decodificador JSON simple.

Realiza las siguientes traducciones en la decodificación de forma predeterminada:

JSON	Python
object	dict
array	list
string	str
número (int)	int
número (real)	float
true	True
false	False
null	None

También entiende `NaN`, `Infinity` y `-Infinity` como sus correspondientes valores `float`, que está fuera de la especificación JSON.

object_hook, si se especifica, se llamará con el resultado de cada objeto JSON decodificado y su valor de retorno se utilizará en lugar de la *dict* dada. Esto se puede usar para proporcionar deserializaciones personalizadas (por ejemplo, para admitir sugerencias de clases JSON-RPC).

object_pairs_hook, si se especifica se llamará con el resultado de cada objeto JSON decodificado con una lista ordenada de pares. El valor de retorno de *object_pairs_hook* se utilizará en lugar de *dict*. Esta característica se puede utilizar para implementar decodificadores personalizados. Si también se define *object_hook*, el *object_pairs_hook* tiene prioridad.

Distinto en la versión 3.1: Soporte agregado para *object_pairs_hook*.

parse_float, si se especifica, se llamará con la cadena de cada flotante JSON que se va a decodificar. De forma predeterminada, esto es equivalente a `float(num_str)`. Esto se puede utilizar para hacer uso de otro tipo de datos o analizador para flotantes JSON (por ejemplo `decimal.Decimal`).

parse_int, si se especifica, se llamará con la cadena de cada entero JSON que se va a decodificar. De forma predeterminada, esto es equivalente a `int(num_str)`. Esto se puede utilizar para hacer uso de otro tipo de datos o analizador para enteros JSON (por ejemplo `float`).

parse_constant, si se especifica, se llamará con una de las siguientes cadenas: `'-Infinity'`, `'Infinity'`, `'NaN'`. Esto se puede utilizar para generar una excepción si se encuentran números JSON inválidos.

Si *strict* es falso (`True` es el valor predeterminado), se permitirán caracteres de control dentro de cadenas. Los caracteres de control en este contexto son aquellos con códigos de caracteres en el rango 0–31, incluyendo `'\t'` (tab), `'\n'`, `'\r'` and `'\0'`.

Si los datos que se deserializan no constituyen un documento JSON válido, se generará un *JSONDecodeError*.

Distinto en la versión 3.6: Todos los parámetros son ahora *palabra-clave-solamente*.

decode (*s*)

Retorna la representación Python de *s* (una instancia *str* que contiene un documento JSON).

JSONDecodeError se producirá si el documento JSON entregado es inválido.

raw_decode (*s*)

Decodifica un documento JSON de *s* (un *str* comenzando con un documento JSON) y retorna una tupla de 2 de la representación Python y el índice en *s* donde terminó el documento.

Esto se puede usar para decodificar un documento JSON de una cadena de caracteres que puede tener datos extraños al final.

```
class json.JSONEncoder(*, skipkeys=False, ensure_ascii=True, check_circular=True,
                        allow_nan=True, sort_keys=False, indent=None, separators=None,
                        default=None)
```

Codificador JSON extensible para estructuras de datos de Python.

Admite los siguientes objetos y tipos de forma predeterminada:

Python	JSON
dict	object
list, tuple	array
str	string
int, float, Enums derivadas de int o float	number
True	true
False	false
None	null

Distinto en la versión 3.4: Compatibilidad añadida con las clases Enum derivadas de int y float.

A fin de extender esto para reconocer otros objetos, implementar una subclase con un método `default()` con otro método que retorna un objeto serializable para “o” si es posible, de lo contrario debe llamar a la implementación de superclase (para elevar `TypeError`).

Si `skipkeys` es falso (valor predeterminado), entonces es un `TypeError` intentar codificar claves que no son `str`, `int`, `float` o `None`. Si `skipkeys` es verdadero, estos elementos simplemente se omiten.

Si `ensure_ascii` es verdadero (el valor predeterminado), se garantiza que la salida tendrá todos los caracteres entrantes no ASCII escapados. Si `ensure_ascii` es falso, estos caracteres se mostrarán tal cual.

Si `check_circular` es cierto (valor predeterminado), se comprobarán las listas, los diccionarios y los objetos codificados personalizados en busca de referencias circulares durante la codificación para evitar una recursividad infinita (lo que provocaría un `OverflowError`). De lo contrario, no se realiza ninguna comprobación de este tipo.

Si `allow_nan` es cierto (valor predeterminado), NaN, Infinity y -Infinity se codificarán como tales. Este comportamiento no es compatible con las especificaciones JSON, pero es coherente con la mayoría de los codificadores y decodificadores basados en JavaScript. De lo contrario, codificar dichos puntos flotantes provocará un `ValueError`.

Si `sort_keys` es cierto (predeterminado: `False`), la salida de los diccionarios se ordenará por clave; esto es útil para las pruebas de regresión para garantizar que las serializaciones JSON se pueden comparar en el día a día.

Si `indent` es un entero no negativo o una cadena, los elementos del arreglo JSON y los miembros del objeto se imprimirán con ese nivel de sangría. Un nivel de sangría de 0, negativo o "" solo insertará nuevas líneas. `None` (el valor predeterminado) selecciona la representación más compacta. El uso de una sangría de entero positivo agrega sangrías de muchos espacios por nivel. Si `indent` es una cadena (como "\t"), esa cadena se usa para agregarle sangría a cada nivel.

Distinto en la versión 3.2: Permite cadenas de caracteres para `indent` además de enteros.

Si se especifica, `separators` debe ser una tupla (`separador_elemento`, `separador_llave`). El valor predeterminado es (`'`, `'`, `': '`) si `indent` es `None` y (`'`, `'`, `': '`) de lo contrario. Para obtener la representación JSON más compacta, debe especificar (`'`, `'`, `': '`) para eliminar espacios en blanco.

Distinto en la versión 3.4: Usa (`'`, `'`, `': '`) como predeterminado si `indent` no es `None`.

Si se especifica, `default` debería ser una función que se llama para objetos que de otro modo no se pueden serializar. Debería retornar una versión codificable JSON del objeto o generar un `TypeError`. Si no se especifica, produce `TypeError`.

Distinto en la versión 3.6: Todos los parámetros son ahora *palabra-clave-solamente*.

default (o)

Implemente este método en una subclase de modo que retorne un objeto serializable para `o`, o llame a la implementación base (para generar un `TypeError`).

Por ejemplo, para admitir iteradores arbitrarios, podría implementar un valor predeterminado como este:

```
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 json.JSONEncoder.default(self, o)
```

encode(*o*)

Retorna una representación de cadena de caracteres JSON de una estructura de datos de Python, *o*. Por ejemplo:

```
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)

Codifica el objeto dado, *o*, y produce cada representación de cadena como disponible. Por ejemplo:

```
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

19.2.3 Excepciones

exception `json.JSONDecodeError` (*msg, doc, pos*)

Subclase de `ValueError` con los siguientes atributos adicionales:

msg

El mensaje de error sin formato.

doc

El documento JSON que se está analizando.

pos

El índice de inicio de *doc* donde se produjo un error en el análisis.

lineno

La línea correspondiente a *pos*.

colno

La columna correspondiente a *pos*.

Nuevo en la versión 3.5.

19.2.4 Cumplimiento e interoperabilidad estándar

El formato JSON se especifica mediante [RFC 7159](#) y por [ECMA-404](#). En esta sección se detalla el nivel de cumplimiento de este módulo con la RFC. Para simplificar, no se tienen en cuenta las subclases `JSONEncoder` y `JSONDecoder` y los parámetros distintos de los mencionados explícitamente.

Este módulo no cumple con la RFC de forma estricta, implementando algunas extensiones que son válidas en JavaScript pero no son válidas en JSON. En particular:

- Se aceptan y se envían valores de números Infinitos y NaN;
- Se aceptan nombres repetidos dentro de un objeto y solo se utiliza el valor del último par nombre-valor.

Puesto que el RFC permite a los analizadores compatibles con RFC aceptar textos de entrada que no son compatibles con RFC, el deserializador de este módulo es técnicamente compatible con RFC bajo la configuración predeterminada.

Codificaciones de caracteres

La RFC requiere que JSON se represente mediante UTF-8, UTF-16 o UTF-32, siendo UTF-8 el valor predeterminado recomendado para la máxima interoperabilidad.

Según lo permitido, aunque no es necesario, por la RFC, el serializador de este módulo establece `ensure_ascii=True` de forma predeterminada, escapando así el dato de salida para que las cadenas resultantes solo contengan caracteres ASCII.

Aparte del parámetro `ensure_ascii`, este módulo se define estrictamente en términos de conversión entre objetos Python y *Unicode strings*, y por lo tanto no aborda directamente el problema de las codificaciones de caracteres.

La RFC prohíbe agregar una marca de orden byte (BOM, por sus siglas en inglés) al inicio de un texto JSON y el serializador de este módulo no agrega una BOM a su salida. La RFC permite, pero no requiere, deserializadores JSON para omitir una BOM inicial en su entrada. El deserializador de este módulo genera un `ValueError` cuando hay una lista de materiales inicial.

La RFC no prohíbe explícitamente las cadenas JSON que contienen secuencias de bytes que no corresponden a caracteres Unicode válidos (por ejemplo, sustitutos UTF-16 no espaciados), pero sí tiene en cuenta que pueden causar problemas de interoperabilidad. De forma predeterminada, este módulo acepta y genera puntos de código (cuando está presente en el original *str*) para dichas secuencias.

Valores de número infinito y NaN

El RFC no permite la representación de los valores de número infinito o NaN. A pesar de eso, de forma predeterminada, este módulo acepta y genera `Infinity`, `-Infinity` y `NaN` como si fueran valores literales de número JSON válidos:

```
>>> # 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
```

En el serializador, el parámetro `allow_nan` se puede utilizar para modificar este comportamiento. En el deserializador, se puede utilizar el parámetro `parse_constant` para modificar este comportamiento.

Nombres repetidos dentro de un objeto

La RFC especifica que los nombres dentro de un objeto JSON deben ser únicos, pero no exige cómo se deben controlar los nombres repetidos en los objetos JSON. De forma predeterminada, este módulo no genera una excepción; en su lugar, ignora todo excepto el último par nombre-valor para un nombre dado:

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

El parámetro `object_pairs_hook` se puede utilizar para alterar este comportamiento.

Valores de nivel superior No-Objeto , No-Arreglo

La versión anterior de JSON especificada por el obsoleto [RFC 4627](#) requería que el valor de nivel superior de un texto JSON fuera un objeto JSON o un arreglo (Python *dict* o *list*), y no podía ser un valor JSON nulo, booleano, numérico o de cadena. [RFC 7159](#) eliminó esa restricción, y este módulo no ha implementado ni ha implementado nunca esa restricción en su serializador o en su deserializador.

Independientemente, para lograr la máxima interoperabilidad, es posible que usted desee adherirse voluntariamente a la restricción.

Limitaciones de la implementación

Algunas implementaciones del deserializador JSON pueden establecer límites en:

- el tamaño de los textos JSON aceptados
- el nivel máximo de anidamiento de objetos y arreglos JSON
- el rango y precisión de los números JSON
- el contenido y la longitud máxima de las cadenas de caracteres JSON

Este módulo no impone tales límites más allá de los propios tipos de datos de Python relevantes o del propio intérprete de Python.

Al serializar en JSON, tenga en cuenta las limitaciones en las aplicaciones que pueden consumir su JSON. En particular, es común que los números JSON se deserialicen en números de doble precisión IEEE 754 y, por lo tanto, estén sujetos al rango y las limitaciones de precisión de esa representación. Esto es especialmente relevante cuando se serializan valores de Python *int* de magnitud extremadamente grande, o cuando se serializan instancias de tipos numéricos «exóticos» como *decimal.Decimal*.

19.2.5 Interfaz de línea de comandos

Código fuente: `Lib/json/tool.py`

El módulo `json.tool` proporciona una interfaz de línea de comandos simple para validar e imprimir objetos JSON.

Si no se especifican los argumentos opcionales `infile` y `outfile`, se utilizarán `sys.stdin` y `sys.stdout` respectivamente:

```
$ echo '{"json": "obj"}' | python -m json.tool
{
  "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

Distinto en la versión 3.5: La salida está ahora en el mismo orden que la entrada. Utilice la opción `--sort-keys` para ordenar la salida de los diccionarios alfabéticamente por llave.

Opciones de línea de comandos

infile

El archivo JSON que se va a validar o imprimir con impresión linda:

```
$ python -m json.tool mp_films.json
[
  {
    "title": "And Now for Something Completely Different",
    "year": 1971
  },
  {
    "title": "Monty Python and the Holy Grail",
    "year": 1975
  }
]
```

Si no se especifica *infile*, lee `sys.stdin`.

outfile

Escribe la salida de *infile* en el *outfile* dado. De lo contrario, lo escribe en `sys.stdout`.

--sort-keys

Ordena la salida de los diccionarios alfabéticamente por llave.

Nuevo en la versión 3.5.

--json-lines

Analiza cada línea de entrada como objeto JSON independiente.

Nuevo en la versión 3.8.

-h, --help

Muestra el mensaje de ayuda.

Notas de pie de página

19.3 mailcap — Manejo de archivos Mailcap

Código fuente: [Lib/mailcap.py](#)

Los archivos mailcap se utilizan para configurar la manera en las que aplicaciones conscientes de formatos MIME como lectores de correo y navegadores web reaccionan a archivos con diferentes tipos MIME. (El nombre «mailcap» viene de la frase «mail capability».) Por ejemplo, un archivo mailcap puede contener una línea como `video/mpeg; xmpeg %s`. Entonces, si el usuario encuentra un mensaje de correo o un documento web con el tipo MIME `video/mpeg`, `%s` será reemplazado por un nombre de archivo (por lo general, uno que pertenezca a un archivo temporal) y el programa **xmpeg** puede ser iniciado de manera automática para visualizar el archivo.

El formato mailcap está documentado en **RFC 1524**, «A User Agent Configuration Mechanism For Multimedia Mail Format Information», pero no es un estándar de Internet. Sin embargo, los archivos mailcap tienen soporte en la mayoría de los sistemas Unix.

`mailcap.findmatch(caps, MIMEtype, key='view', filename='/dev/null', plist=[])`

Retorna una tupla; el primer elemento es una cadena que contiene la línea de comando a ser ejecutada (la cual puede ser pasada a la función `os.system()`), y el segundo elemento es la entrada mailcap para el tipo MIME proporcionado. Si no se encuentra un tipo MIME que coincida, entonces se retornan los valores de `(None, None)`.

key es el nombre del campo deseado, que representa el tipo de actividad a realizar; el valor por defecto es “view”, ya que en el caso más común se quiere simplemente ver el cuerpo de los datos tecleados en MIME.

Otros posibles valores podrían ser “compose” y “edit”, si se quisiera crear un nuevo cuerpo del tipo MIME dado o alterar los datos del cuerpo existente. Consulta [RFC 1524](#) para una lista completa de estos campos.

filename es el nombre de fichero que debe ser sustituido por %s en la línea de comandos; el valor por defecto es '/dev/null' que casi seguro no es lo que quieres, así que normalmente lo anularás especificando un nombre de archivo.

plist puede ser una lista que contenga parámetros con nombre; el valor por defecto es simplemente una lista vacía. Cada entrada de la lista debe ser una cadena que contenga el nombre del parámetro, un signo igual ('=') y el valor del parámetro. Las entradas de mailcap pueden contener parámetros con nombre como {%foo}, que serán reemplazados por el valor del parámetro llamado “foo”. Por ejemplo, si la línea de comandos showpartial {%id} {%number} {%total} estaba en un archivo mailcap, y *plist* estaba establecido como ['id=1', 'number=2', 'total=3'], la línea de comandos resultante sería 'showpartial 1 2 3'.

En un archivo mailcap, el campo «test» puede especificarse opcionalmente para probar alguna condición externa (como la arquitectura de la máquina, o el sistema de ventanas en uso) para determinar si se aplica o no la línea mailcap. La función *findmatch()* comprobará automáticamente dichas condiciones y omitirá la entrada si la comprobación falla.

Distinto en la versión 3.8.16: To prevent security issues with shell metacharacters (symbols that have special effects in a shell command line), *findmatch* will refuse to inject ASCII characters other than alphanumerics and @+=: , . / - _ into the returned command line.

If a disallowed character appears in *filename*, *findmatch* will always return (None, None) as if no entry was found. If such a character appears elsewhere (a value in *plist* or in *MIMEtype*), *findmatch* will ignore all mailcap entries which use that value. A *warning* will be raised in either case.

`mailcap.getcaps()`

Retorna un diccionario que mapea los tipos de MIME a una lista de entradas de archivos mailcap. Este diccionario debe ser pasado a la función *findmatch()*. Una entrada se almacena como una lista de diccionarios, pero no debería ser necesario conocer los detalles de esta representación.

La información se deriva de todos los archivos mailcap que se encuentran en el sistema. Los ajustes en el archivo mailcap del usuario \$HOME/.mailcap anularán los ajustes en los archivos mailcap del sistema /etc/mailcap, /usr/etc/mailcap, y /usr/local/etc/mailcap.

Un ejemplo de uso:

```
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
('xmpeg tmp1223', {'view': 'xmpeg %s'})
```

19.4 mailbox — Manipular buzones de correo en varios formatos

Código fuente: [Lib/mailbox.py](#)

Este módulo define dos clases, *Mailbox* y *Message*, para acceder y manipular en disco los buzones de correo y los mensajes que contienen. *Mailbox* ofrece una asignación similar a un diccionario de claves a mensajes. *Message* extiende la clase *email.message* del módulo *Message* con el estado y el comportamiento específicos del formato. Los formatos de buzón de correo compatibles son Maildir, mbox, MH, Babyl y MMDf.

Ver también:

Módulo *email* Representar y manipular mensajes.

19.4.1 Objetos :class:"Mailbox"

class mailbox.Mailbox

Un buzón de correo, que se puede inspeccionar y modificar.

La clase *Mailbox* define una interfaz y no está diseñada para crear instancias. En su lugar, las subclases específicas del formato deben heredar de *Mailbox* y el código debe crear una instancia de una subclase determinada.

La interfaz *Mailbox* es similar a un diccionario, con pequeñas claves correspondientes a los mensajes. Las claves son emitidas por la instancia *Mailbox* con la que se utilizarán y solo son significativas para esa instancia *Mailbox*. Una clave continúa identificando un mensaje incluso si se modifica el mensaje correspondiente, por ejemplo, sustituyéndolo por otro mensaje.

Los mensajes se pueden agregar a una instancia *Mailbox* utilizando el método como *add()* y quitarse mediante una instrucción *del* o los métodos como *remove()* y *discard()*.

La semántica de la interfaz *Mailbox* difiere de la semántica del diccionario en algunos aspectos notables. Cada vez que se solicita un mensaje, se genera una nueva representación (típicamente una instancia *Message*) basada en el estado actual del buzón de correo. De forma similar, cuando se añade un mensaje a una instancia *Mailbox*, se copia el contenido de la representación del mensaje proporcionado. En ninguno de los dos casos se mantiene una referencia a la representación del mensaje por parte de la instancia *Mailbox*.

El iterador por defecto de *Mailbox* itera sobre las representaciones de los mensajes, no sobre las claves como lo hace el iterador del diccionario por defecto. Además, la modificación de un buzón de correo durante la iteración es segura y bien definida. Los mensajes añadidos al buzón de correo después de que se cree un iterador no serán vistos por el iterador. Los mensajes eliminados del buzón de correo antes de que el iterador los ceda serán omitidos silenciosamente, aunque el uso de una clave de un iterador puede dar lugar a una excepción *KeyError* si el mensaje correspondiente es eliminado posteriormente.

Advertencia: Sea muy cauteloso al modificar los buzones de correo que pueden ser cambiados simultáneamente por algún otro proceso. El formato más seguro de buzón de correo que se puede utilizar para esas tareas es Maildir; trate de evitar el uso de formatos de un solo archivo, como mbox, para la escritura simultánea. Si estás modificando un buzón de correo, *debes* bloquearlo llamando a los métodos *lock()* y *unlock()* antes de leer cualquier mensaje en el fichero o hacer cualquier cambio añadiendo o borrando un mensaje. Si no se bloquea el buzón se corre el riesgo de perder mensajes o de corromper todo el buzón.

Las instancias de *Mailbox* tienen los siguientes métodos:

add (*message*)

Añade *message* al buzón de correo y retorna la clave que se le ha asignado.

El parámetro *message* puede ser una instancia *Message*, una instancia *email.message.Message*, una cadena, una cadena de bytes o un objeto tipo archivo (que debe estar abierto en modo binario). Si *message* es una instancia de la subclase *Message* con el formato apropiado (por ejemplo, si es una instancia *mboxMessage* y ésta es una instancia *mbox*), se utiliza su información de formato específico. En caso contrario, se utilizan valores por defecto razonables para la información específica del formato.

Distinto en la versión 3.2: Se añadió el soporte para la entrada binaria.

remove (*key*)

__delitem__ (*key*)

discard (*key*)

Borre el mensaje correspondiente a la *key* del buzón de correo.

Si no existe tal mensaje, se levanta una excepción *KeyError* si el método se llamó *remove()* o *__delitem__()* pero no se levanta una excepción si el método se llamó *discard()*. El comportamiento de *discard()* puede ser preferido si el formato de buzón subyacente soporta la modificación concurrente por otros procesos.

__setitem__ (*key*, *message*)

Reemplaza el mensaje correspondiente a *key* por *message*. Levante una excepción *KeyError* si ningún

mensaje ya corresponde a *key*.

Al igual que `add()`, el parámetro *message* puede ser una instancia `Message`, una instancia `email.message.Message`, una cadena, una cadena de bytes, o un objeto tipo archivo (que debe estar abierto en modo binario). Si *message* es una instancia de la subclase `Message` con el formato apropiado (por ejemplo, si es una instancia `mboxMessage` y ésta es una instancia `mbox`), se utiliza su información de formato específico. En caso contrario, la información específica del formato del mensaje que actualmente corresponde a *key* se deja sin cambios.

iterkeys()

keys()

Retorna un iterador sobre todas las claves si se llama `iterkeys()` o retorna una lista de claves si se llama `keys()`.

intervalues()

__iter__()

values()

Retorna un iterador sobre las representaciones de todos los mensajes si se llama `intervalues()` o `__iter__()` o retorna una lista de tales representaciones si se llama `values()`. Los mensajes se representan como instancias de la subclase `Message` específica del formato, a menos que se haya especificado una fábrica de mensajes personalizados cuando se haya inicializado la instancia `Mailbox`.

Nota: El comportamiento de `__iter__()` es diferente al de los diccionarios, que iteran sobre las claves.

iteritems()

items()

Retorna un iterador sobre los pares (*key*, *message*), donde *key* es una clave y *message* es una representación de un mensaje, si se llama como `iteritems()` o retorna una lista de tales pares si se llama como `items()`. Los mensajes se representan como instancias de la subclase `Message` específica del formato, a menos que se haya especificado una fábrica de mensajes personalizados cuando se haya inicializado la instancia `Mailbox`.

get(key, default=None)

__getitem__(key)

Retorna una representación del mensaje correspondiente a *key*. Si no existe tal mensaje, se retorna *default* si el método fue llamado como `get()` y se produce una excepción `KeyError` si el método fue llamado como `__getitem__()`. El mensaje se representa como una instancia de la subclase `Message` específica del formato, a menos que se especificara una fábrica de mensajes personalizados cuando se inicializa la instancia `Mailbox`.

get_message(key)

Retorna una representación del mensaje correspondiente a *key* como una instancia de la subclase `Message` específica del formato, o lanza una excepción `KeyError` si no existe tal mensaje.

get_bytes(key)

Retorna una representación en bytes del mensaje correspondiente a *key*, o levanta una excepción `KeyError` si no existe tal mensaje.

Nuevo en la versión 3.2.

get_string(key)

Retorna una representación en cadena del mensaje correspondiente a *key*, o lanza una excepción `KeyError` si no existe tal mensaje. El mensaje se procesa a través de `email.message.Message` para convertirlo en una representación limpia de 7 bits.

get_file(key)

Retorna una representación en forma de archivo del mensaje correspondiente a *key*, o lanza una excepción `KeyError` si no existe tal mensaje. El objeto tipo archivo se comporta como si estuviera abierto en modo binario. Este archivo debería cerrarse una vez que ya no se necesite.

Distinto en la versión 3.2: El objeto del archivo es realmente un archivo binario; anteriormente fue retornado incorrectamente en modo de texto. Además, el objeto tipo archivo ahora soporta el protocolo de gestión de contexto: puedes usar una sentencia `with` para cerrarlo automáticamente.

Nota: A diferencia de otras representaciones de mensajes, las representaciones en forma de archivo no son necesariamente independientes de la instancia `Mailbox` que las creó o del buzón de correo subyacente. Cada subclase proporciona una documentación más específica.

__contains__ (*key*)

Retorna `True` si «*key*» corresponde a un mensaje, si no `False`.

__len__ ()

Retorna un recuento de los mensajes en el buzón de correo.

clear ()

Borrar todos los mensajes del buzón.

pop (*key*, *default=None*)

Retorna una representación del mensaje correspondiente a *key* y borra el mensaje. Si no existe tal mensaje, retorna *default*. El mensaje se representa como una instancia de la subclase `Message` con el formato apropiado, a menos que se haya especificado una fábrica de mensajes personalizados al inicializar la instancia `Mailbox`.

popitem ()

Retorna un par arbitrario (*key*, *message*), donde *key* es una clave y *message* es una representación de un mensaje, y borra el mensaje correspondiente. Si el buzón de correo está vacío, lanza una excepción `KeyError`. El mensaje se representa como una instancia de la subclase `Message` específica del formato, a menos que se haya especificado una fábrica de mensajes personalizados al inicializar la instancia `Mailbox`.

update (*arg*)

El parámetro *arg* debe ser un mapa de *key* a *message* o un iterable de pares (*key*, *message*). Actualiza el buzón de correo para que, por cada *key* y *message* dados, el mensaje correspondiente a *key* será establecido a *message* como si se usara `__setitem__()`. Como con `__setitem__()`, cada *key* debe corresponder ya a un mensaje en el buzón de correo o de lo contrario se lanzará una excepción `KeyError`, por lo que en general es incorrecto que *arg* sea una instancia de `Mailbox`.

Nota: A diferencia de los diccionarios, los argumentos de las palabras clave no están soportados.

flush ()

Escribe cualquier cambio pendiente en el sistema de archivos. Para algunas subclases de `Mailbox`, los cambios siempre se escriben inmediatamente y `flush()` no hace nada, pero aún así deberías tener el hábito de llamar a este método.

lock ()

Adquiera un aviso exclusivo de bloqueo en el buzón de correo para que otros procesos sepan que no deben modificarlo. Un `ExternalClashError` se lanza si el bloqueo no está disponible. Los mecanismos de bloqueo particulares utilizados dependen del formato del buzón de correo. Deberías *siempre* bloquear el buzón antes de hacer cualquier modificación a su contenido.

unlock ()

Libera el bloqueo del buzón de correo, si lo hay.

close ()

Limpia el buzón de correo, y lo desbloquea si es necesario, y cierra cualquier archivo abierto. Para algunas subclases de `Mailbox`, este método no hace nada.

Maildir

class mailbox.**Maildir** (*dirname*, *factory=None*, *create=True*)

Una subclase de *Mailbox* para los buzones de correo en formato Maildir. El parámetro *factory* es un objeto invocable que acepta una representación de mensaje tipo archivo (que se comporta como si se abriera en modo binario) y retorna una representación personalizada. Si *factory* es *None*, *MaildirMessage* se utiliza como representación de mensaje por defecto. Si *create* es *True*, el buzón se crea si no existe.

Si *create* es *True* y la ruta de *dirname* existe, será tratado como un *maildir* existente sin intentar verificar su diseño de directorio.

Es por razones históricas que *dirname* es nombrado como tal en lugar de *path*.

Maildir es un formato de buzón de correo basado en un directorio inventado para el agente de transferencia de correo qmail y ahora ampliamente soportado por otros programas. Los mensajes en un buzón de correo de Maildir se almacenan en archivos separados dentro de una estructura de directorio común. Este diseño permite que los buzones de Maildir sean accedidos y modificados por múltiples programas no relacionados sin corrupción de datos, por lo que el bloqueo de archivos es innecesario.

Los buzones de correo de Maildir contienen tres subdirectorios, a saber: *tmp*, *new*, y *cur*. Los mensajes se crean momentáneamente en el subdirectorio *tmp* y luego se mueven al subdirectorio *new* para finalizar la entrega. Un agente de usuario de correo puede posteriormente mover el mensaje al subdirectorio *cur* y almacenar la información sobre el estado del mensaje en una sección especial «info» adjunta a su nombre de archivo.

Se admiten también carpetas del estilo introducido por el agente de transferencia de correo Courier. Cualquier subdirectorio del buzón de correo principal se considera una carpeta si *'.'* es el primer carácter de su nombre. Los nombres de las carpetas están representados por *Maildir* sin la palabra *'.'*. Cada carpeta es en sí misma un buzón de correo de Maildir pero no debe contener otras carpetas. En su lugar, se indica un anidamiento lógico usando *'.'* para delimitar los niveles, por ejemplo, «Archived.2005.07».

Nota: La especificación Maildir requiere el uso de dos puntos (*':'*) en ciertos nombres de archivos de mensajes. Sin embargo, algunos sistemas operativos no permiten este carácter en los nombres de archivo, si desea utilizar un formato similar a Maildir en dicho sistema operativo, debe especificar otro carácter para utilizarlo en su lugar. El signo de exclamación (*'!'*) es una elección popular. Por ejemplo:

```
import mailbox
mailbox.Maildir.colon = '!'
```

El atributo *colon* también puede ser establecido para cada instancia.

Las instancias de *Maildir* tienen todos los métodos de *Mailbox* además de los siguientes:

list_folders ()

Retorna una lista con los nombres de todas las carpetas.

get_folder (*folder*)

Retorna una instancia *Maildir* que representa la carpeta cuyo nombre es *folder*. Una excepción *NoSuchMailboxError* se lanza si la carpeta no existe.

add_folder (*folder*)

Crea una carpeta cuyo nombre sea *folder* y retorna una instancia *Maildir* que la represente.

remove_folder (*folder*)

Elimina la carpeta cuyo nombre es *folder*. Si la carpeta contiene algún mensaje, se lanzará una excepción *NotEmptyError* y la carpeta no se borrará.

clean ()

Borra los archivos temporales del buzón de correo que no han sido accedidos en las últimas 36 horas. La especificación Maildir dice que los programas de lectura de correo deben hacer esto ocasionalmente.

Algunos métodos de *Mailbox* implementados por *Maildir* merecen comentarios especiales:

```
add(message)
__setitem__(key, message)
update(arg)
```

Advertencia: Estos métodos generan nombres de archivo únicos basados en el ID del proceso actual. Cuando se utilizan varios hilos, pueden producirse conflictos de nombres no detectados y causar la corrupción del buzón de correo a menos que se coordinen los hilos para evitar que se utilicen estos métodos para manipular el mismo buzón de correo simultáneamente.

flush()
 Todos los cambios en los buzones de Maildir se aplican inmediatamente, así que este método no hace nada.

lock()
unlock()
 Los buzones de Maildir no admiten (o requieren) bloqueo, por lo que estos métodos no hacen nada.

close()
 Las instancias de *Maildir* no mantienen ningún archivo abierto y los buzones subyacentes no soportan el bloqueo, por lo que este método no hace nada.

get_file(*key*)
 Dependiendo de la plataforma del host, puede que no sea posible modificar o eliminar el mensaje subyacente mientras el archivo retornado permanezca abierto.

Ver también:

pagina web maildir de Courier A specification of the format. Describes a common extension for supporting folders.

Utilizando el formato maildir Notas sobre Maildir por su inventor. Incluye un esquema actualizado de creación de nombres y detalles sobre la «info» de la semántica».

mbox

```
class mailbox.mbox(path, factory=None, create=True)
```

Una subclase de *Mailbox* para los buzones de correo en formato mbox. El parámetro *factory* es un objeto invocable que acepta una representación de mensaje tipo archivo (que se comporta como si se abriera en modo binario) y retorna una representación personalizada. Si *factory* es *None*, *mboxMessage* se utiliza como representación de mensaje por defecto. Si *create* es *True*, el buzón de correo se crea si no existe.

El formato mbox es el formato clásico para almacenar correo en sistemas Unix. Todos los mensajes de un buzón de correo mbox se almacenan en un único archivo con el comienzo de cada mensaje indicado por una línea cuyos cinco primeros caracteres son «From».

Existen varias variaciones del formato mbox para abordar las deficiencias percibidas en el original. En aras de la compatibilidad, *mbox* implementa el formato original, que a veces se denomina *mboxo*. Esto significa que el encabezado *Content-Length*, si está presente, se ignora y que cualquier ocurrencia de «From » al principio de una línea en el cuerpo de un mensaje se transforma en «>From » al almacenar el mensaje, aunque las ocurrencias de «>From » no se transforman en «From » al leer el mensaje.

Algunos métodos de *Mailbox* implementados por *Maildir* merecen comentarios especiales:

get_file(*key*)
 Usar el archivo después de llamar a *flush()* o *close()* en la instancia *mbox* puede producir resultados impredecibles o lanzar una excepción.

lock()

unlock()

Se utilizan tres mecanismos de bloqueo... el bloqueo por puntos y, si está disponible, las llamadas del sistema `flock()` y `lockf()`.

Ver también:

pagina web mbox de tin A specification of the format, with details on locking.

Configurando el correo de Netscape en Unix: Por qué el formato de longitud de contenido es malo Un argumento para usar el formato original mbox en lugar de una variación.

«mbox» es una familia de varios formatos de buzón de correo mutuamente incompatibles <<https://www.loc.gov/preservation>>
Una historia de variaciones de mbox.

MH

class mailbox.MH(*path*, *factory=None*, *create=True*)

Una subclase de *Mailbox* para los buzones de correo en formato MH. El parámetro *factory* es un objeto invocable que acepta una representación de mensaje tipo archivo (que se comporta como si se abriera en modo binario) y retorna una representación personalizada. Si *factory* es *None*, *MHMessage* se utiliza como representación de mensaje por defecto. Si *create* es *True*, el buzón de correo se crea si no existe.

MH es un formato de buzón de correo basado en un directorio inventado para el Sistema de Manejo de Mensajes MH, un agente de usuario de correo. Cada mensaje de un buzón de correo MH reside en su propio archivo. Un buzón de correo MH puede contener otros buzones de correos MH (llamados *folders*) además de los mensajes. Las carpetas pueden anidarse indefinidamente. Los buzones de correo MH también soportan *sequences*, que son listas con nombre usadas para agrupar lógicamente los mensajes sin moverlos a subcarpetas. Las secuencias se definen en un archivo llamado `.mh_sequences` en cada carpeta.

La clase *MH* manipula los buzones de correos de MH, pero no intenta emular todos los comportamientos de *mh*. En particular, no modifica ni se ve afectado por los archivos de `context` o `.mh_profile` que utiliza *mh* para almacenar su estado y configuración.

Las instancias de *Maildir* tienen todos los métodos de *Mailbox* además de los siguientes:

list_folders()

Retorna una lista con los nombres de todas las carpetas.

get_folder(folder)

Retorna una instancia *Maildir* que representa la carpeta cuyo nombre es *folder*. Una excepción *NoSuchMailboxError* se lanza si la carpeta no existe.

add_folder(folder)

Crea una carpeta cuyo nombre sea *folder* y retorna una instancia *MH* que la represente.

remove_folder(folder)

Elimina la carpeta cuyo nombre es *folder*. Si la carpeta contiene algún mensaje, se lanzará una excepción *NotEmptyError* y la carpeta no se borrará.

get_sequences()

Retorna un diccionario de nombres de secuencias mapeadas a listas clave. Si no hay secuencias, se retorna el diccionario vacío.

set_sequences(sequences)

Re-define las secuencias que existen en el buzón de correo basado en *sequences*, un diccionario de nombres mapeados a listas de claves, como las retornadas por *get_sequences()*.

pack()

Renombra los mensajes en el buzón de correo según sea necesario para eliminar los huecos en la numeración. Las entradas en la lista de secuencias se actualizan correspondientemente.

Nota: Las llaves ya emitidas quedan invalidadas por esta operación y no deben utilizarse posteriormente.

Algunos métodos de *Mailbox* implementados por *Maildir* merecen comentarios especiales:

remove (*key*)

__delitem__ (*key*)

discard (*key*)

Estos métodos borran inmediatamente el mensaje. No se utiliza la convención del MH de marcar un mensaje para borrarlo poniendo una coma en su nombre.

lock ()

unlock ()

Se utilizan tres mecanismos de bloqueo... el bloqueo por puntos y, si está disponible, las llamadas del sistema `flock()` y `lockf()`. Para los buzones de correos MH, bloquear el buzón de correo significa bloquear el archivo `.mh_sequences` y, sólo durante la duración de cualquier operación que les afecte, bloquear los archivos de mensajes individuales.

get_file (*key*)

Dependiendo de la plataforma anfitriona, puede que no sea posible eliminar el mensaje subyacente mientras el archivo retornado permanezca abierto.

flush ()

Todos los cambios en los buzones de correos de Maildir se aplican inmediatamente, así que este método no hace nada.

close ()

Las instancias de *MH* no mantienen ningún archivo abierto, así que este método es equivalente a `unlock()`.

Ver también:

nmh - Sistema de Manejo de Mensajes Página principal de **nmh**, una versión actualizada del original **mh**.

MH & nmh: Correo electrónico para usuarios y programadores Un libro con licencia GPL sobre **mh** y **nmh**, con alguna información sobre el formato del buzón.

Babyl

class mailbox.Babyl (*path*, *factory=None*, *create=True*)

Una subclase de *Mailbox* para los buzones en formato Babyl. El parámetro *factory* es un objeto invocable que acepta una representación de mensaje tipo archivo (que se comporta como si se abriera en modo binario) y retorna una representación personalizada. Si *factory* es *None*, *BabylMessage* se utiliza como representación de mensaje por defecto. Si *create* es *True*, el buzón se crea si no existe.

Babyl es un formato de buzón de un solo archivo usado por el agente de usuario de correo de Rmail incluido en Emacs. El comienzo de un mensaje se indica con una línea que contiene los dos caracteres Control-Underscore (`'\037'`) y Control-L (`'\014'`). El final de un mensaje se indica con el comienzo del siguiente mensaje o, en el caso del último mensaje, una línea que contiene un carácter Control-Underscore (`'\037'`).

Los mensajes en un buzón de correo de Babyl tienen dos juegos de encabezados, los encabezados originales y los llamados encabezados visibles. Los encabezados visibles son típicamente un subconjunto de los encabezados originales que han sido reformateados o abreviados para ser más atractivos. Cada mensaje de un buzón de correo de Babyl también tiene una lista de *labels*, o cadenas cortas que registran información adicional sobre el mensaje, y una lista de todas las etiquetas definidas por el usuario que se encuentran en el buzón de correo se mantiene en la sección de opciones de Babyl.

Las instancias de *Maildir* tienen todos los métodos de *Mailbox* además de los siguientes:

get_labels ()

Retorna una lista de los nombres de todas las etiquetas definidas por el usuario utilizadas en el buzón de correo.

Nota: Los mensajes actuales se inspeccionan para determinar qué etiquetas existen en el buzón de correo en lugar de consultar la lista de etiquetas en la sección de opciones de Babyl, pero la sección de Babyl se

actualiza cada vez que se modifica el buzón de correo.

Algunos métodos de *Mailbox* implementados por *Maildir* merecen comentarios especiales:

get_file (*key*)

En los buzones de correos de Babyl, los encabezados de un mensaje no se almacenan contiguamente al cuerpo del mensaje. Para generar una representación tipo archivo, las cabeceras y el cuerpo se copian juntos en una instancia *io.BytesIO*, que tiene una API idéntica a la de un archivo. Como resultado, el objeto similar a un archivo es verdaderamente independiente del buzón de correo subyacente, pero no ahorra memoria en comparación con una representación en cadena.

lock ()

unlock ()

Se utilizan tres mecanismos de bloqueo... el bloqueo por puntos y, si está disponible, las llamadas del sistema *flock* () y *lockf* () .

Ver también:

«Formato de la versión 5 de los archivos de Babyl» <<https://quimby.gnus.org/notes/BABYL>>_ Una especificación del formato Babyl.

Leyendo el correo con Rmail El manual de Rmail, con cierta información sobre la semántica Babyl.

MMDF

class mailbox.**MMDF** (*path*, *factory=None*, *create=True*)

Una subclase de *Mailbox* para buzones de correos en formato MMDF. El parámetro *factory* es un objeto al que se puede llamar que acepta una representación de mensaje similar a un archivo (que se comporta como si se abriera en modo binario) y retorna una representación personalizada. Si *factory* es *None*, *MMDFMessage* se utiliza como representación de mensaje predeterminada. Si *create* es *True*, el buzón de correo se crea si no existe.

MMDF es un formato de buzón de correo de un solo archivo inventado para el centro de distribución de memorandos multicanal, un agente de transferencia de correo. Cada mensaje está en la misma forma que un mensaje de mbox, pero está entre corchetes antes y después por líneas que contienen cuatro caracteres Control-A (' \001 '). Al igual que con el formato mbox, el principio de cada mensaje se indica mediante una línea cuyos primeros cinco caracteres son «From », pero las apariciones adicionales de «From» no se transforman en «>From» al almacenar mensajes porque las líneas de separador de mensajes adicionales impiden confundir tales ocurrencias para los inicios de los mensajes posteriores.

Algunos métodos *Mailbox* implementados por *MMDF* merecen comentarios especiales:

get_file (*key*)

Usar el archivo después de llamar a *flush* () o *close* () en la instancia *MMDF* puede producir resultados impredecibles o generar una excepción.

lock ()

unlock ()

Se utilizan tres mecanismos de bloqueo... el bloqueo por puntos y, si está disponible, las llamadas del sistema *flock* () y *lockf* () .

Ver también:

Página web de mmdf por Tin Una especificación del formato MMDF de la documentación de tin, un lector de noticias.

MMDF Un artículo de Wikipedia que describe el Centro de Distribución de Memorandos Multicanal.

19.4.2 Objetos Message

class mailbox.Message (*message=None*)

Una subclase del módulo `email.message` de `Message`. Las subclases de `mailbox.Message` añaden el estado y el comportamiento específicos del formato del buzón de correo.

Si se omite `message`, la nueva instancia se crea en un estado predeterminado, vacío. Si `message` es una instancia `email.message.Message`, se copian sus contenidos; además, cualquier información específica del formato se convierte en la medida de lo posible si `message` es una instancia `Message`. Si `message` es una cadena, una cadena de bytes, o un archivo, debe contener un mensaje conforme [RFC 2822](#), que se lee y analiza. Los archivos deben estar abiertos en modo binario, pero los archivos en modo texto son aceptados para compatibilidad con versiones anteriores.

El estado y los comportamientos específicos del formato que ofrecen las subclases varían, pero en general sólo se admiten las propiedades que no son específicas de un buzón de correo concreto (aunque presumiblemente las propiedades son específicas de un formato de buzón de correo concreto). Por ejemplo, no se conservan las compensaciones de archivos para los formatos de buzón de correo de un solo archivo ni los nombres de archivo para los formatos de buzón de correo basados en directorios, porque sólo son aplicables al buzón de correo original. Pero sí se conservan las declaraciones tales como si un mensaje ha sido leído por el usuario o marcado como importante, porque se aplican al propio mensaje.

No hay ningún requisito de que las instancias de `Message` se usen para representar los mensajes recuperados usando las instancias de `Mailbox`. En algunas situaciones, el tiempo y la memoria necesarios para generar representaciones de `Message` podrían no ser aceptables. Para estas situaciones, las instancias de `Mailbox` también ofrecen representaciones en forma de cadenas y archivos, y se puede especificar una fábrica de mensajes personalizados cuando se inicializa una instancia de `Mailbox`.

MaildirMessage

class mailbox.MaildirMessage (*message=None*)

Un mensaje con comportamientos específicos de Maildir. El parámetro `message` tiene el mismo significado que con el constructor `Message`.

Típicamente, una aplicación de agente de usuario de correo mueve todos los mensajes del subdirectorio `new` al subdirectorio `cur` después de la primera vez que el usuario abre y cierra el buzón de correo, registrando que los mensajes son antiguos, tanto si han sido leídos como si no. Cada mensaje en `cur` tiene una sección «info» añadida a su nombre de archivo para almacenar información sobre su estado. (Algunos lectores de correo también pueden añadir una sección «info» a los mensajes en `new`.) La sección «info» puede tomar una de dos formas: puede contener «2», seguido de una lista de flags estandarizados (por ejemplo, «2,FR») o puede contener «1», seguido de la llamada información experimental. Los flags normalizados para los mensajes de Maildir son los siguientes:

Flag	Significado	Explicación
D	Borrador	Bajo composición
F	Marcada	Marcado como importante
P	Aprobado	Enviado, reenviado o rebotado
R	Contestado	Contestado a
S	Visto	Leído
T	Destruído	Marcado para su posterior eliminación

Instancias de `MaildirMessage` ofrecen los siguientes métodos:

get_subdir()

Retorna «new» (si el mensaje debe ser almacenado en el subdirectorio `new`) o «cur» (si el mensaje debe ser almacenado en el subdirectorio `cur`).

Nota: Un mensaje es típicamente movido de `nuevo` a `cur` después de que su buzón de correo ha sido accedido, ya sea que el mensaje haya sido leído o no. Un mensaje `msg` ha sido leído si "S" in

`msg.get_flags()` es `True`.

set_subdir (*subdir*)

Establece el subdirectorio en el que debe almacenarse el mensaje. El parámetro *subdir* debe ser «new» o «cur».

get_flags ()

Retorna una cadena que especifica los flags que están actualmente establecidos. Si el mensaje cumple con el formato estándar de Maildir, el resultado es la concatenación en orden alfabético de cero o una ocurrencia de cada una de los flags 'D', 'F', 'P', 'R', 'S', y 'T'. La cadena vacía se retorna si no hay flags o si «info» contiene semántica experimental.

set_flags (*flags*)

Establece los flags especificados por *flags* y desactiva todas las demás.

add_flag (*flag*)

Establece lo(s) flag(s) especificado(s) por *flag* sin cambiar otros flags. Para añadir más de un indicador a la vez, *flag* puede ser una cadena de más de un carácter. La «info» actual se sobrescribe si contiene o no información experimental en lugar de flags.

remove_flag (*flag*)

Deshabilita lo(s) flag(s) especificado(s) por *flag* sin cambiar otros flags. Para quitar más de un indicador a la vez, *flag* puede ser una cadena de más de un carácter. Si «info» contiene información experimental en lugar de flags, la «info» actual no se modifica.

get_date ()

Retorna la fecha de entrega del mensaje como un número de punto flotante que representa los segundos desde la época.

set_date (*date*)

Establece la fecha de entrega del mensaje en *date*, un número de punto flotante que representa los segundos desde la época.

get_info ()

Retorna una cadena que contiene la «info» de un mensaje. Esto es útil para acceder y modificar la «info» que es experimental (es decir, no una lista de flags).

set_info (*info*)

Establece «info» en *info*, que debería ser una cadena.

Cuando se crea una instancia *MaildirMessage* basada en una instancia *mbxMessage* o *MMDFMessage*, se omiten las cabeceras *Status* y *X-Status* y se producen las siguientes conversiones:

Estado resultante	Estado <i>mbxMessage</i> o <i>MMDFMessage</i>
subdirectorio «cur»	indicador O
indicador F	indicador F
indicador R	indicador A
indicador S	indicador R
indicador T	indicador D

Cuando se crea una instancia *MaildirMessage* basada en una instancia *MHMessage*, se producen las siguientes conversiones:

Estado resultante	Estado <i>MHMessage</i>
subdirectorio «cur»	Secuencia « <i>unseen</i> » (no vista)
subdirectorio «cur» e indicador S	no hay una secuencia « <i>unseen</i> » (invisible)
indicador F	secuencia « <i>flagged</i> » (marcada)
indicador R	Secuencia « <i>replied</i> » (respondida)

Cuando se crea una instancia *MaildirMessage* basada en una instancia *BabylMessage*, se producen las siguientes conversiones:

Estado resultante	Estado <i>BabylMessage</i>
subdirectorio «cur»	etiqueta « <i>unseen</i> » (invisible)
subdirectorio «cur» e indicador S	no hay una etiqueta « <i>unseen</i> » (invisible)
indicador P	etiqueta « <i>forwarded</i> » o « <i>resent</i> » (reenviado)
indicador R	etiqueta de « <i>answered</i> » (contestado)
indicador T	etiqueta « <i>deleted</i> » (borrado)

mbboxMessage

class mailbox.**mbboxMessage** (*message=None*)

Un mensaje con comportamientos específicos de mbox. El parámetro *message* tiene el mismo significado que con el constructor *Message*.

Los mensajes en un buzón de correo de mbox se almacenan juntos en un solo archivo. La dirección del sobre del remitente y la hora de entrega se almacenan normalmente en una línea que comienza con «From» que se utiliza para indicar el comienzo de un mensaje, aunque hay una variación considerable en el formato exacto de estos datos entre las implementaciones de mbox. Los flags del estado del mensaje, como por ejemplo si ha sido leído o marcado como importante, se almacenan típicamente en las cabeceras *Status* y *X-Status*.

Los flags convencionales para los mensajes de mbox son los siguientes:

Flag	Significado	Explicación
R	Leído	Leído
O	Antiguo	Anteriormente detectado por MUA
D	Borrado	Marcado para su posterior eliminación
F	Marcada	Marcado como importante
A	Respondido	Contestado a

Los flags «R» y «O» se almacenan en el encabezado *Status* y los flags «D», «F» y «A» se almacenan en el encabezado *X-Status*. Los flags y los encabezados aparecen típicamente en el orden mencionado.

Instancias de *mbboxMessage* ofrecen los siguientes métodos:

get_from()

Retorna una cadena que representa la línea «From» que marca el inicio del mensaje en un buzón de correo de mbox. El «From» inicial y la nueva línea final están excluidas.

set_from (*from_, time_=None*)

Ponga la línea «From» en *from_*, que debe ser especificada sin una línea «From» o una nueva línea posterior. Para mayor comodidad, se puede especificar *time_*, que se formateará adecuadamente y se añadirá a *from_*. Si se especifica *time_*, debe ser una instancia *time.struct_time*, una tupla adecuada para pasar a *time.strftime()*, o True (para usar *time.gmtime()*).

get_flags()

Retorna una cadena que especifica los flags que están actualmente establecidos. Si el mensaje cumple con el formato convencional, el resultado es la concatenación en el siguiente orden de cero o una ocurrencia de cada uno de los flags «R», «O», «D», «F», and «A».

set_flags (*flags*)

Establece los flags especificadas por *flags* y desactiva todos las demás. El parámetro *flags* debe ser la concatenación en cualquier orden de cero o más ocurrencias de cada uno de los flags «R», «O», «D», «F», and «A».

add_flag (*flag*)

Establece lo(s) flag(s) especificado(s) por *flag* sin cambiar otros flags. Para añadir más de un indicador a la vez, *flag* puede ser una cadena de más de un carácter.

remove_flag (*flag*)

Deshabilita lo(s) flag(s) especificado(s) por *flag* sin cambiar otros flags. Para quitar más de un indicador a la vez, *flag* puede ser una cadena de más de un carácter.

Cuando se crea una instancia `mbxMessage` basada en una instancia `MaiIdirMessage`, se genera una línea «From» basada en la fecha de entrega de la instancia `MaiIdirMessage`, y se realizan las siguientes conversiones:

Estado resultante	Estado de <code>MaiIdirMessage</code>
indicador R	indicador S
indicador O	subdirectorio «cur»
indicador D	indicador T
indicador F	indicador F
indicador A	indicador R

Cuando se crea una instancia `mbxMessage` basada en una instancia `MHMessage`, se producen las siguientes conversiones:

Estado resultante	Estado <code>MHMessage</code>
indicador R e indicador O	no hay una secuencia «unseen» (invisible)
indicador O	Secuencia «unseen» (no vista)
indicador F	secuencia «flagged» (marcada)
indicador A	Secuencia «replied» (respondida)

Cuando se crea una instancia `mbxMessage` basada en una instancia `BabylMessage`, se producen las siguientes conversiones:

Estado resultante	Estado <code>BabylMessage</code>
indicador R e indicador O	no hay una etiqueta «unseen» (invisible)
indicador O	etiqueta «unseen» (invisible)
indicador D	etiqueta «deleted» (borrado)
indicador A	etiqueta de «answered» (contestado)

Cuando se crea una instancia `Message` basada en una instancia `MMDFMessage`, la línea «From» se copia y todas los flags se corresponden directamente:

Estado resultante	Estado de <code>MMDFMessage</code>
indicador R	indicador R
indicador O	indicador O
indicador D	indicador D
indicador F	indicador F
indicador A	indicador A

MHMessage

class `mailbox.MHMessage` (*message=None*)

Un mensaje con comportamientos específicos de la HM. El parámetro *message* tiene el mismo significado que con el constructor `Message`.

Los mensajes de MH no soportan marcas o flags en el sentido tradicional, pero sí secuencias, que son agrupaciones lógicas de mensajes arbitrarios. Algunos programas de lectura de correo (aunque no los estándares **mh** y **nmh**) usan secuencias de manera muy similar a los flags que se usan con otros formatos, como sigue:

Secuencia	Explicación
<i>unseen</i> (no visto)	No leído, pero previamente detectado por la MUA
<i>replied</i> (contestado)	Contestado a
<i>flagged</i> (marcado)	Marcado como importante

Instancias de `MHMessage` ofrecen los siguientes métodos:

get_sequences ()

Retorna una lista de los nombres de las secuencias que incluyen este mensaje.

set_sequences (*sequences*)

Establece la lista de secuencias que incluyen este mensaje.

add_sequence (*sequence*)

Añade *sequence* a la lista de secuencias que incluyen este mensaje.

remove_sequence (*sequence*)

Elimina *sequence* de la lista de secuencias que incluyen este mensaje.

Cuando se crea una instancia *MHMessage* basada en una instancia *MaildirMessage*, se producen las siguientes conversiones:

Estado resultante	Estado de <i>MaildirMessage</i>
Secuencia « <i>unseen</i> » (no vista)	no hay indicador S
Secuencia « <i>replied</i> » (respondida)	indicador R
secuencia « <i>flagged</i> » (marcada)	indicador F

Cuando se crea una instancia *MHMessage* basada en una instancia *mboxMessage* o *MMDFMessage*, se omiten las cabeceras *Status* y *X-Status* y se producen las siguientes conversiones:

Estado resultante	Estado <i>mboxMessage</i> o <i>MMDFMessage</i>
Secuencia « <i>unseen</i> » (no vista)	sin indicador R
Secuencia « <i>replied</i> » (respondida)	indicador A
secuencia « <i>flagged</i> » (marcada)	indicador F

Cuando se crea una instancia *MHMessage* basada en una instancia *BabylMessage*, se producen las siguientes conversiones:

Estado resultante	Estado <i>BabylMessage</i>
Secuencia « <i>unseen</i> » (no vista)	etiqueta « <i>unseen</i> » (invisible)
Secuencia « <i>replied</i> » (respondida)	etiqueta de « <i>answered</i> » (contestado)

BabylMessage

class mailbox.*BabylMessage* (*message=None*)

Un mensaje con comportamientos específicos de Babyl. El parámetro *message* tiene el mismo significado que con el constructor *Message*.

Ciertas etiquetas de mensajes, llamadas *attributes*, están definidas por convención para tener significados especiales. Los atributos son los siguientes:

Etiqueta	Explicación
<i>unseen</i> (no visto)	No leído, pero previamente detectado por la MUA
<i>deleted</i> (borrado)	Marcado para su posterior eliminación
<i>filed</i> (archivado)	Copiado a otro archivo o buzón de correo
<i>answered</i> (contestado)	Contestado a
<i>forwarded</i> (reenviado)	Reenviado
<i>edited</i> (editado)	Modificado por el usuario
<i>resent</i> (reenviado)	Reenviado

De forma predeterminada, Rmail sólo muestra las cabeceras visibles. La clase *BabylMessage*, sin embargo, usa los encabezados originales porque son más completos. Se puede acceder a las cabeceras visibles explícitamente si se desea.

Instancias de *BabylMessage* ofrecen los siguientes métodos:

get_labels()

Retorna una lista de etiquetas en el mensaje.

set_labels(labels)Establece la lista de etiquetas del mensaje en *labels*.**add_label(label)**Añade *label* a la lista de etiquetas del mensaje.**remove_label(label)**Eliminar *label* de la lista de etiquetas del mensaje.**get_visible()**Retorna una instancia de *Message* cuyos encabezados son los encabezados visibles del mensaje y cuyo cuerpo está vacío.**set_visible(visible)**Establece los encabezados visibles del mensaje para que sean los mismos que los del *message*. El parámetro *visible* debe ser una instancia *Message*, una instancia *email.message.Message*, una cadena, o un objeto tipo archivo (que debe estar abierto en modo texto).**update_visible()**

Cuando se modifican los encabezados originales de una instancia *BabylMessage*, los encabezados visibles no se modifican automáticamente para que se correspondan. Este método actualiza los encabezados visibles de la siguiente manera: cada encabezado visible con un encabezado original correspondiente se establece como el valor del encabezado original, cada encabezado visible sin un encabezado original correspondiente se elimina, y cualquiera de : *mailheader:Date*, *From*, *Reply-To*, *To*, *CC*, y *Subject* que están presentes en las cabeceras originales pero no las cabeceras visibles se añaden a las cabeceras visibles.

Cuando se crea una instancia *BabylMessage* basada en una instancia *MaildirMessage*, se producen las siguientes conversiones:

Estado resultante	Estado de <i>MaildirMessage</i>
etiqueta « <i>unseen</i> » (invisible)	no hay indicador S
etiqueta « <i>deleted</i> » (borrado)	indicador T
etiqueta de « <i>answered</i> » (contestado)	indicador R
etiqueta « <i>forwarded</i> » (reenviado)	indicador P

Cuando se crea una instancia *BabylMessage* basada en una instancia *mboxMessage* o *MMDfMessage*, se omiten los encabezados *Status* y *X-Status* y se producen las siguientes conversiones:

Estado resultante	Estado <i>mboxMessage</i> o <i>MMDfMessage</i>
etiqueta « <i>unseen</i> » (invisible)	sin indicador R
etiqueta « <i>deleted</i> » (borrado)	indicador D
etiqueta de « <i>answered</i> » (contestado)	indicador A

Cuando se crea una instancia *BabylMessage* basada en una instancia *MHMessage*, se producen las siguientes conversiones:

Estado resultante	Estado <i>MHMessage</i>
etiqueta « <i>unseen</i> » (invisible)	Secuencia « <i>unseen</i> » (no vista)
etiqueta de « <i>answered</i> » (contestado)	Secuencia « <i>replied</i> » (respondida)

MMDFMessage

class mailbox.**MMDFMessage** (*message=None*)

Un mensaje con comportamientos específicos de MMDF. El parámetro *message* tiene el mismo significado que con el constructor *Message*.

Al igual que los mensajes en un buzón de correo de mbox, los mensajes MMDF se almacenan con la dirección del remitente y la fecha de entrega en una línea inicial que comienza con «From». De la misma manera, los flags que indican el estado del mensaje se almacenan típicamente en las cabeceras *Status* y *X-Status*.

Los flags convencionales para los mensajes MMDF son idénticos a las de los mensajes de mbox y son los siguientes:

Flag	Significado	Explicación
R	Leído	Leído
O	Antiguo	Anteriormente detectado por MUA
D	Borrado	Marcado para su posterior eliminación
F	Marcada	Marcado como importante
A	Respondido	Contestado a

Los flags «R» y «O» se almacenan en el encabezado *Status* y los flags «D», «F» y «A» se almacenan en el encabezado *X-Status*. Los flags y los encabezados aparecen típicamente en el orden mencionado.

Las instancias de *MMDFMessage* ofrecen los siguientes métodos, que son idénticos a los ofrecidos por *mboxMessage*:

get_from ()

Retorna una cadena que representa la línea «From» que marca el inicio del mensaje en un buzón de correo de mbox. El «From» inicial y la nueva línea final están excluidas.

set_from (*from_*, *time_=None*)

Ponga la línea «From» en *from_*, que debe ser especificada sin una línea «From» o una nueva línea posterior. Para mayor comodidad, se puede especificar *time_*, que se formateará adecuadamente y se añadirá a *from_*. Si se especifica *time_*, debe ser una instancia *time.struct_time*, una tupla adecuada para pasar a *time.strftime()*, o True (para usar *time.gmtime()*).

get_flags ()

Retorna una cadena que especifica los flags que están actualmente establecidos. Si el mensaje cumple con el formato convencional, el resultado es la concatenación en el siguiente orden de cero o una ocurrencia de cada uno de los flags "R", "O", "D", "F", and "A".

set_flags (*flags*)

Establece los flags especificadas por *flags* y desactiva todos las demás. El parámetro *flags* debe ser la concatenación en cualquier orden de cero o más ocurrencias de cada uno de los flags 'R', 'O', 'D', 'F', and 'A'.

add_flag (*flag*)

Establece lo(s) flag(s) especificado(s) por *flag* sin cambiar otros flags. Para añadir más de un indicador a la vez, *flag* puede ser una cadena de más de un carácter.

remove_flag (*flag*)

Deshabilita lo(s) flag(s) especificado(s) por *flag* sin cambiar otros flags. Para quitar más de un indicador a la vez, *flag* puede ser una cadena de más de un carácter.

Cuando se crea una instancia *MMDFMessage* basada en una instancia *MaildirMessage*, se genera una línea «From » basada en la fecha de entrega de la instancia *MaildirMessage*, y se realizan las siguientes conversiones:

Estado resultante	Estado de <i>MaiMdirMessage</i>
indicador R	indicador S
indicador O	subdirectorío «cur»
indicador D	indicador T
indicador F	indicador F
indicador A	indicador R

Cuando se crea una instancia *MMDFMessage* basada en una instancia *MHMessage*, se producen las siguientes conversiones:

Estado resultante	Estado <i>MHMessage</i>
indicador R e indicador O	no hay una secuencia «unseen» (invisible)
indicador O	Secuencia «unseen» (no vista)
indicador F	secuencia «flagged» (marcada)
indicador A	Secuencia «replied» (respondida)

Cuando se crea una instancia *MMDFMessage* basada en una instancia *BabylMessage*, se producen las siguientes conversiones:

Estado resultante	Estado <i>BabylMessage</i>
indicador R e indicador O	no hay una etiqueta «unseen» (invisible)
indicador O	etiqueta «unseen» (invisible)
indicador D	etiqueta «deleted» (borrado)
indicador A	etiqueta de «answered» (contestado)

Cuando se crea una instancia *MMDFMessage* basada en una instancia *mboxMessage*, la línea «From » se copia y todos los flags se corresponden directamente:

Estado resultante	Estado de <i>mboxMessage</i>
indicador R	indicador R
indicador O	indicador O
indicador D	indicador D
indicador F	indicador F
indicador A	indicador A

19.4.3 Excepciones

Las siguientes clases de excepción están definidas en el módulo *mailbox*:

exception `mailbox.Error`

La clase base para todas las demás excepciones específicas del módulo.

exception `mailbox.NoSuchMailboxError`

Se lanza cuando se espera un buzón de correo pero no se encuentra, como cuando se instancia una subclase *Mailbox* con una ruta que no existe (y con el parámetro *create* establecido en *False*), o cuando se abre una carpeta que no existe.

exception `mailbox.NotEmptyError`

Se lanza cuando un buzón de correo no está vacío, pero se espera que lo esté, como cuando se elimina una carpeta que contiene mensajes.

exception `mailbox.ExternalClashError`

Se lanza cuando alguna condición relacionada con el buzón de correo, fuera del control del programa, hace que éste no pueda proceder, como por ejemplo cuando se falla en la adquisición de un bloqueo que es mantenido por otro programa, o cuando ya existe un nombre de archivo generado de forma única.

exception `mailbox.FormatError`

Se lanza cuando los datos de un archivo no pueden ser analizados, como cuando una instancia `MH` intenta leer un archivo `.mh_sequences` corrupto.

19.4.4 Ejemplos

Un simple ejemplo de impresión de los temas de todos los mensajes en un buzón de correo que parecen interesantes:

```
import mailbox
for message in mailbox.mbox('~/.mbox'):
    subject = message['subject'] # Could possibly be None.
    if subject and 'python' in subject.lower():
        print(subject)
```

Para copiar todo el correo de un buzón de Babyl a un buzón de MH, convirtiendo toda la información de formato específico que puede ser convertida:

```
import mailbox
destination = mailbox.MH('~/.Mail')
destination.lock()
for message in mailbox.Babyl('~/.RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

Este ejemplo clasifica el correo de varias listas de correo en diferentes buzones, teniendo cuidado de evitar la corrupción del correo debido a la modificación simultánea por otros programas, la pérdida de correo debido a la interrupción del programa, o la terminación prematura debido a mensajes malformados en el buzón:

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {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()
```

(continué en la próxima página)

(proviene de la página anterior)

```
        inbox.unlock()
        break                # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```

19.5 mimetypes — Mapea nombres de archivo a tipos MIME

Código fuente: [Lib/mimetypes.py](#)

El módulo *mimetypes* convierte entre un nombre de archivo o URL y el tipo MIME asociado a la extensión del nombre de archivo. Se proporcionan conversiones de nombre de archivo a tipo MIME y de tipo MIME a extensión de nombre de archivo; no se admiten codificaciones para esta última conversión.

El módulo proporciona una clase y varias funciones de conveniencia. Las funciones son la interfaz normal de este módulo, pero algunas aplicaciones pueden estar interesadas en la clase también.

Las funciones que se describen a continuación constituyen la interfaz principal de este módulo. Si el módulo no ha sido inicializado, llamarán *init()* si se basan en la información que *init()* establece.

mimetypes.guess_type(url, strict=True)

Adivina el tipo de un archivo basado en su nombre de archivo, ruta o URL, dado por *url*. La URL puede ser una cadena o un objeto *path-like object*.

El valor de retorno es una tupla (*type*, *encoding*) donde *type* es *None* si el tipo no puede ser adivinado (por sufijo ausente o desconocido) o una cadena de la forma *type/subtype*, utilizable para un encabezado MIME *content-type*.

encoding es *None* para no codificar o el nombre del programa usado para codificar (por ejemplo **compress** o **gzip**). La codificación es adecuada para ser usada como una cabecera de *Content-Encoding*, **no** como una cabecera de *Content-Transfer-Encoding*. Los mapeos son conducidos por tablas. Los sufijos de codificación son sensibles a las mayúsculas y minúsculas; los sufijos de tipo se prueban primero distinguiendo entre mayúsculas y minúsculas, y luego sin dicha distinción.

El argumento opcional *strict* es una flag que especifica si la lista de tipos MIME conocidos se limita sólo a los tipos oficiales [registrados en IANA](#). Cuando *strict* es *True* (el valor por defecto), sólo se soportan los tipos de IANA; cuando *strict* es *False*, también se reconocen algunos tipos MIME adicionales no estándar pero de uso común.

Distinto en la versión 3.8: Añadido soporte para que la URL sea un objeto *path-like object*.

mimetypes.guess_all_extensions(type, strict=True)

Adivina las extensiones de un archivo basadas en su tipo MIME, dadas por *type*. El valor de retorno es una lista de cadenas que dan todas las extensiones posibles del nombre del archivo, incluyendo el punto inicial ('.'). No se garantiza que las extensiones hayan sido asociadas con ningún flujo de datos en particular, pero serían mapeadas al tipo MIME *type* por *guess_type()*.

El argumento opcional *strict* tiene el mismo significado que con la función *guess_type()*.

mimetypes.guess_extension(type, strict=True)

Adivina la extensión de un archivo basada en su tipo MIME, dada por *type*. El valor de retorno es una cadena que da una extensión de nombre de archivo, incluyendo el punto inicial ('.'). No se garantiza que la extensión haya sido asociada con ningún flujo de datos en particular, pero sería mapeada al tipo MIME *type* por *guess_type()*. Si no se puede adivinar ninguna extensión para *type*, se retorna *None*.

El argumento opcional *strict* tiene el mismo significado que con la función *guess_type()*.

Algunas funciones adicionales y elementos de datos están disponibles para controlar el comportamiento del módulo.

`mimetypes.init(files=None)`

Inicializa las estructuras de datos internos. Si se proporciona *files* debe ser una secuencia de nombres de archivos que deben utilizarse para aumentar el mapa de tipo por defecto. Si se omite, los nombres de archivo a utilizar se toman de *knownfiles*; en Windows, se cargan las configuraciones actuales del registro. Cada archivo nombrado en *files* o *knownfiles* tiene prioridad sobre los nombrados antes de él. Se permite llamar repetidamente a *init()*.

Si se especifica una lista vacía para *files* se evitará que se apliquen los valores predeterminados del sistema: sólo estarán presentes los valores conocidos de una lista incorporada.

Si *files* es `None` la estructura interna de datos se reconstruye completamente a su valor inicial por defecto. Esta es una operación estable y producirá los mismos resultados cuando se llame varias veces.

Distinto en la versión 3.2: Anteriormente, la configuración del registro de Windows se ignoraba.

`mimetypes.read_mime_types(filename)`

Carga el mapa de tipo dado en el archivo *filename*, si existe. El mapa de tipos es retornado como un diccionario que mapea las extensiones de los nombres de archivo, incluyendo el punto inicial ('.'), a las cadenas de la forma 'type/subtype'. Si el archivo *filename* no existe o no puede ser leído, se retorna `None`.

`mimetypes.add_type(type, ext, strict=True)`

Añade un mapeo del tipo MIME *type* a la extensión *ext*. Cuando la extensión ya se conoce, el nuevo tipo reemplazará al antiguo. Cuando el tipo ya se conoce la extensión se añadirá a la lista de extensiones conocidas.

Cuando *strict* es `True` (el valor por defecto), el mapeo se añadirá a los tipos MIME oficiales, de lo contrario a los no estándar.

`mimetypes.inited`

Flag que indica si se han inicializado o no las estructuras de datos globales. Esto se establece como «True» por *init()*.

`mimetypes.knownfiles`

Lista de los nombres de los archivos de mapas de tipo comúnmente instalados. Estos archivos se llaman típicamente `mime.types` y se instalan en diferentes lugares por diferentes paquetes.

`mimetypes.suffix_map`

Diccionario que mapea sufijos a sufijos. Se utiliza para permitir el reconocimiento de archivos codificados cuya codificación y tipo se indican con la misma extensión. Por ejemplo, la extensión `.tgz` se mapea a `.tar.gz` para permitir que la codificación y el tipo se reconozcan por separado.

`mimetypes.encodings_map`

El diccionario mapea las extensiones de los nombres de archivo a los tipos de codificación.

`mimetypes.types_map`

Diccionario que mapea extensiones de los nombres de archivo a tipos MIME.

`mimetypes.common_types`

Diccionario que mapea extensiones de los nombres de archivo a tipos MIME no estándar, pero comúnmente encontrados.

Un ejemplo de utilización del módulo:

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

19.5.1 Objetos MimeTypes

La clase `MimeTypes` puede ser útil para aplicaciones que quieran más de una base de datos de tipo MIME; proporciona una interfaz similar a la del módulo `mimetypes`.

class `mimetypes.MimeTypes` (*filenames=()*, *strict=True*)

Esta clase representa una base de datos de tipos MIME. Por defecto, proporciona acceso a la misma base de datos que el resto de este módulo. La base de datos inicial es una copia de la proporcionada por el módulo, y puede ser extendida cargando archivos adicionales de tipo `mime.types` en la base de datos usando los métodos `read()` o `readfp()`. Los diccionarios de mapeo también pueden ser borrados antes de cargar datos adicionales si no se desean los datos por defecto.

El parámetro opcional *filenames* puede utilizarse para hacer que se carguen archivos adicionales «encima» de la base de datos predeterminada.

suffix_map

El diccionario mapea sufijos a sufijos. Se utiliza para permitir el reconocimiento de archivos codificados cuya codificación y tipo se indican con la misma extensión. Por ejemplo, la extensión `.tgz` se mapea a `.tar.gz` para permitir que la codificación y el tipo se reconozcan por separado. Esto es inicialmente una copia del global `suffix_map` definido en el módulo.

encodings_map

El diccionario mapea las extensiones de los nombres de archivo a los tipos de codificación. Es inicialmente una copia del global `encodings_map` definido en el módulo.

types_map

Una tupla que contiene dos diccionarios, mapeando las extensiones de los nombres de archivo a los tipos MIME: el primer diccionario es para los tipos no estándar y el segundo para los tipos estándar. Están inicializados por `common_types` y `types_map`.

types_map_inv

Una tupla que contiene dos diccionarios, mapeando los tipos MIME a una lista de extensiones de nombres de archivos: el primer diccionario es para los tipos no estándar y el segundo para los tipos estándar. Están inicializados por `common_types` y `types_map`.

guess_extension (*type*, *strict=True*)

Similar a la función `guess_extension()`, usando las tablas almacenadas como parte del objeto.

guess_type (*url*, *strict=True*)

Similar a la función `guess_type()`, usando las tablas almacenadas como parte del objeto.

guess_all_extensions (*type*, *strict=True*)

Similar a la función `guess_all_extensions()`, usando las tablas almacenadas como parte del objeto.

read (*filename*, *strict=True*)

Carga información MIME de un archivo llamado *filename*. Esto usa `readfp()` para analizar el archivo.

Si *strict* es `True`, la información se añadirá a la lista de tipos estándar, si no a la lista de tipos no estándar.

readfp (*fp*, *strict=True*)

Carga información de tipo MIME de un archivo abierto *fp*. El archivo debe tener el formato de los archivos estándar `mime.types`.

Si *strict* es `True`, la información se va a añadir a la lista de tipos estándar, de otro modo se añadirá a la lista de tipos no estándar.

read_windows_registry (*strict=True*)

Carga información desde el registro de Windows del tipo de metadato MIME.

Disponibilidad: Windows.

Si *strict* es `True`, la información se va a añadir a la lista de tipos estándar, de otro modo se añadirá a la lista de tipos no estándar.

Nuevo en la versión 3.2.

19.6 base64 — Codificaciones de datos Base16, Base32, Base64, y Base85

Código fuente: [Lib/base64.py](#)

Este módulo proporciona funciones para codificar datos binarios en caracteres ASCII imprimibles y decodificar dichas codificaciones en datos binarios. Proporciona funciones de codificación y decodificación para las codificaciones especificadas en [RFC 3548](#), que define los algoritmos Base16, Base32 y Base64, y para las codificaciones estándar de facto Ascii85 y Base85.

Las codificaciones [RFC 3548](#) son adecuadas para codificar datos binarios para que puedan enviarse de forma segura por correo electrónico, usarse como partes de URL o incluirse como parte de una solicitud HTTP POST. El algoritmo de codificación no es el mismo que el programa **uuencode**.

Hay dos interfaces proporcionadas por este módulo. La interfaz moderna admite la codificación de *objetos similares a bytes* a ASCII *bytes*, y decodificación *objetos similares a bytes* o cadenas de caracteres que contienen ASCII a *bytes*. Ambos alfabetos de base 64 definidos en [RFC 3548](#) (normal y seguro para URL y sistema de archivos) son compatibles.

La interfaz heredada no admite la decodificación desde cadenas de caracteres, pero sí proporciona funciones para codificar y decodificar desde y hacia *objetos de archivo*. Solo admite el alfabeto estándar Base64 y agrega nuevas líneas cada 76 caracteres según [RFC 2045](#). Tenga en cuenta que si está buscando soporte de [RFC 2045](#), probablemente desee ver el paquete *email* en su lugar.

Distinto en la versión 3.3: Las cadenas de caracteres Unicode de solo ASCII ahora son aceptadas por las funciones de decodificación de la interfaz moderna.

Distinto en la versión 3.4: Cualquier *objeto similar a bytes* ahora son aceptados por todas las funciones de codificación y decodificación en este módulo. Ascii85/Base85 soporte agregado.

Las interfaces modernas proporcionan:

`base64.b64encode(s, altchars=None)`

Codifica el *objeto similar a bytes* *s* utilizando Base64 y retorna los *bytes* codificados.

Los *altchars* opcionales deben ser un *objeto similar a bytes* de al menos longitud 2 (se ignoran los caracteres adicionales) que especifica un alfabeto alternativo para los caracteres `+` y `/`. Esto permite que una aplicación, por ejemplo, generar URL o cadenas de caracteres de Base64 seguras para el sistema de archivos. El valor predeterminado es `None`, para el que se utiliza el alfabeto estándar Base64.

`base64.b64decode(s, altchars=None, validate=False)`

Decodifica el *objeto similar a bytes* codificado en Base64 o cadena de caracteres ASCII *s* y retorna los *bytes* decodificados.

Los *altchars* opcionales deben ser *objetos similares a byte* o cadena de caracteres ASCII de al menos longitud 2 (se ignoran los caracteres adicionales) que especifica el alfabeto alternativo utilizado en lugar de los caracteres `+` y `/`.

Una excepción `binascii.Error` se lanza si *s* está incorrectamente relleno (*padded*).

Si *validate* es `False` (el valor predeterminado), los caracteres que no están en el alfabeto normal de base 64 ni en el alfabeto alternativo se descartan antes de la verificación del relleno. Si *validate* es `True`, estos caracteres no alfabéticos en la entrada dan como resultado `binascii.Error`.

`base64.standard_b64encode(s)`

Codifica el *objeto similar a bytes* *s* usando el alfabeto estándar Base64 y retorna los *bytes* codificados.

`base64.standard_b64decode(s)`

Decodifica un *bytes-like object* o cadena de caracteres ASCII *s* utilizando el alfabeto estándar Base64 y retorna los *bytes* decodificados.

`base64.urlsafe_b64encode(s)`

Codifica el *objeto similar a bytes* *s* usando el alfabeto seguro para URL y sistemas de archivos, que sustituye

a – en lugar de + y _ en lugar de / en el alfabeto estándar de Base64, y retorna los *bytes* codificados. El resultado aún puede contener =.

`base64.urlsafe_b64decode(s)`

Decodifica *objeto similar a bytes* o cadena de caracteres ASCII *s* utilizando el alfabeto seguro para URL y sistema de archivos, que sustituye – en lugar de + y _ en lugar de / en el alfabeto estándar de Base64, y retorna los *bytes* decodificados.

`base64.b32encode(s)`

Codifica el *objeto similar a bytes* *s* utilizando Base32 y retorna los *bytes* codificados.

`base64.b32decode(s, casefold=False, map01=None)`

Decodifica el *objeto similar a bytes* codificado en Base32 o cadena de caracteres ASCII *s* y retorna los *bytes* decodificados.

El opcional *casefold* es un flag que especifica si un alfabeto en minúscula es aceptable como entrada. Por motivos de seguridad, el valor predeterminado es `Falso`.

RFC 3548 permite el mapeo opcional del dígito 0 (cero) a la letra O (oh), y el mapeo opcional del dígito 1 (uno) a la letra I (eye) o la letra L (el). El argumento opcional *map01* cuando no es `None`, especifica a qué letra se debe asignar el dígito 1 (cuando *map01* no es `None`, el dígito 0 siempre se asigna a la letra O). Por motivos de seguridad, el valor predeterminado es `None`, por lo que 0 y 1 no están permitidos en la entrada.

Una *binascii.Error* se lanza si *s* está incorrectamente rellenado (*padded*) o si hay caracteres no alfabéticos presentes en la entrada.

`base64.b16encode(s)`

Codifica el *objeto similar a bytes* *s* utilizando Base16 y retorna los *bytes* codificados.

`base64.b16decode(s, casefold=False)`

Decodifica el *objeto similar a bytes* codificado en Base16 o cadena de caracteres ASCII *s* y retorna los *bytes* decodificados.

El opcional *casefold* es un flag que especifica si un alfabeto en minúscula es aceptable como entrada. Por motivos de seguridad, el valor predeterminado es `Falso`.

Una *binascii.Error* se lanza si *s* está incorrectamente rellenado (*padded*) o si hay caracteres no alfabéticos presentes en la entrada.

`base64.a85encode(b, *, foldspaces=False, wrapcol=0, pad=False, adobe=False)`

Codifica el *objeto similar a bytes* *b* utilizando Ascii85 y retorna los *bytes* codificados.

foldspaces es un flag opcional que utiliza la secuencia corta especial “y” en lugar de 4 espacios consecutivos (ASCII 0x20) como lo admite “btoa”. Esta característica no es compatible con la codificación Ascii85 «estándar».

wrapcol controla si la salida debe tener caracteres de nueva línea (b' \n ') agregados. Si esto no es cero, cada línea de salida tendrá como máximo esta cantidad de caracteres.

pad controla si la entrada se rellena (*padded*) a un múltiplo de 4 antes de la codificación. Tenga en cuenta que la implementación de `btoa` siempre es rellena (*pads*).

adobe controla si la secuencia de bytes codificada está enmarcada con <~ y ~>, que es utilizada por la implementación de Adobe.

Nuevo en la versión 3.4.

`base64.a85decode(b, *, foldspaces=False, adobe=False, ignorechars=b' \t\n\r\v')`

Decodifica el *objeto similar a bytes* codificado en Ascii85 o cadena de caracteres ASCII *b* y retorna los *bytes* decodificados.

foldspaces es un flag que especifica si la secuencia corta “y” debe aceptarse como abreviatura durante 4 espacios consecutivos (ASCII 0x20). Esta característica no es compatible con la codificación Ascii85 «estándar».

adobe controla si la secuencia de entrada está en formato Adobe Ascii85 (es decir, se enmarca con <~ y ~>).

`ignorechars` debe ser un *objeto similar a byte* o cadena de caracteres ASCII que contiene caracteres para ignorar desde la entrada. Esto solo debe contener caracteres de espacio en blanco, y por defecto contiene todos los caracteres de espacio en blanco en ASCII.

Nuevo en la versión 3.4.

`base64.b85encode(b, pad=False)`

Codifica el *objeto similar a bytes* `b` utilizando base85 (como se usa en por ejemplo, diferencias binarias de estilo git) y retorna los *bytes* codificados.

Si `pad` es verdadero, la entrada se rellena con `b'\0'`, por lo que su longitud es un múltiplo de 4 bytes antes de la codificación.

Nuevo en la versión 3.4.

`base64.b85decode(b)`

Decodifica el *objeto similar a bytes* codificado en base85 o cadena de caracteres ASCII `b` y retorna los *bytes* decodificados. El relleno se elimina implícitamente, si es necesario.

Nuevo en la versión 3.4.

La interfaz antigua:

`base64.decode(input, output)`

Decodifica el contenido del archivo binario `input` y escribe los datos binarios resultantes en el archivo `output`. `input` y `output` deben ser *objetos archivo*. `input` se leerá hasta que `input.readline()` retorne un objeto de bytes vacío.

`base64.decodebytes(s)`

Decodifica el *objeto similar a bytes* `s`, que debe contener una o más líneas de datos codificados en base64, y retornará los *bytes* decodificados.

Nuevo en la versión 3.1.

`base64.decodestring(s)`

Alias deprecado de `decodebytes()`.

Obsoleto desde la versión 3.1.

`base64.encode(input, output)`

Codifica el contenido del archivo binario `input` y escribe los datos codificados en base64 resultantes en el archivo `output`. `input` y `output` deben ser *objetos archivos*. `input` se leerá hasta que `input.read()` retorne un objeto de bytes vacío. `encode()` inserta un carácter de nueva línea (`b'\n'`) después de cada 76 bytes de la salida, además de garantizar que la salida siempre termine con una nueva línea, según **RFC 2045** (MIME).

`base64.encodebytes(s)`

Codifica el *objeto similar a bytes* `s`, que puede contener datos binarios arbitrarios, y retorna *bytes* que contienen los datos codificados en base64, con líneas nuevas (`b'\n'`) insertado después de cada 76 bytes de salida, y asegurando que haya una nueva línea final, según **RFC 2045** (MIME).

Nuevo en la versión 3.1.

`base64.encodestring(s)`

Alias deprecado de `encodebytes()`.

Obsoleto desde la versión 3.1.

Un ejemplo de uso del módulo:

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSB1bmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

Ver también:

Módulo *binascii* Módulo de soporte que contiene conversiones de ASCII a binario y binario a ASCII.

RFC 1521 - MIME (Extensiones multipropósito de correo de Internet) Parte uno: Mecanismos para especificar y describir el

La Sección 5.2, «Codificación de transferencia de contenido Base64», proporciona la definición de la codificación base64.

19.7 binhex — Codificar y decodificar archivos binhex4

Código fuente: [Lib/binhex.py](#)

Este módulo codifica y decodifica archivos en formato binhex4, un formato que permite la representación de archivos Macintosh en ASCII. Solo se maneja «data fork».

El módulo *binhex* define las siguientes funciones:

`binhex.binhex(input, output)`

Convierta un archivo binario con nombre de archivo *input* a archivo binhex *output*. El parámetro *output* puede ser un nombre de archivo o un objeto similar a un archivo (cualquier objeto que admita un método `write()` y `close()`).

`binhex.hexbin(input, output)`

Descodificar un archivo binhex *input*. *input* puede ser un nombre de archivo o un objeto similar a un archivo que admita los métodos `read()` y `close()`. El archivo resultante se escribe en un archivo denominado *output*, a menos que el argumento sea `None` en cuyo caso el nombre de archivo de salida se lee desde el archivo binhex.

También se define la siguiente excepción:

exception `binhex.Error`

Excepción que se produce cuando algo no se puede codificar con el formato binhex (por ejemplo, un nombre de archivo demasiado largo para caber en el campo de nombre de archivo) o cuando la entrada no está codificada correctamente como datos binhex.

Ver también:

Módulo *binascii* Módulo de soporte que contiene conversiones ASCII a binario y de binario a ASCII.

19.7.1 Notas

Hay una interfaz alternativa más potente al codificador y decodificador, ver la fuente para más detalles.

Si codifica o decodifica ficheros de texto en plataformas no Macintosh, aún así se usará la antigua convención de nueva línea de Macintosh (retorno de carro al final de la línea).

19.8 binascii — Convertir entre binario y ASCII

El módulo *binascii* contiene una serie de métodos para convertir entre representaciones binarias y varias representaciones binarias codificadas en ASCII. Normalmente, usted no usará estas funciones directamente, en su lugar utilice módulos envoltorios (*wrapper*) como *uu*, *base64*, o *binhex* en su lugar. El módulo *binascii* contiene funciones de bajo nivel escritas en C para una mayor velocidad que son utilizadas por los módulos de nivel superior.

Nota: Las funciones `a2b_*` aceptan cadenas Unicode que contienen solo caracteres ASCII. Otras funciones solo aceptan *objetos tipo binarios* (como *bytes*, *bytearray* y otros objetos que admiten el protocolo de búfer).

Distinto en la versión 3.3: Las cadenas ASCII-only unicode son ahora aceptadas por las funciones `a2b_*`.

El módulo *binascii* define las siguientes funciones:

`binascii.a2b_uu` (*string*)

Convierte una sola línea de datos uuencoded de nuevo a binarios y retorna los datos binarios. Las líneas normalmente contienen 45 bytes (binarios), excepto por la última línea. Los datos de línea pueden ir seguidos de espacios en blanco.

`binascii.b2a_uu` (*data*, *, *backtick=False*)

Convierte datos binarios a una línea de caracteres ASCII, el valor retornado es la línea convertida, incluido un carácter de nueva línea. La longitud de *data* debe ser como máximo 45. Si *backtick* es verdadero, los ceros se representan mediante ' ` ' en lugar de espacios.

Distinto en la versión 3.7: Se ha añadido el parámetro *backtick*.

`binascii.a2b_base64` (*string*)

Convierte un bloque de datos en base64 de nuevo a binario y retorna los datos binarios. Se puede pasar más de una línea a la vez.

`binascii.b2a_base64` (*data*, *, *newline=True*)

Convierte datos binarios en una línea de caracteres ASCII en codificación base64. El valor retornado es la línea convertida, incluido un carácter de nueva línea si *newline* es verdadero. La salida de esta función se ajusta a [RFC 3548](#).

Distinto en la versión 3.6: Se ha añadido el parámetro *newline*.

`binascii.a2b_qp` (*data*, *header=False*)

Convierte un bloque de datos imprimibles entre comillas a binario y retorna los datos binarios. Se puede pasar más de una línea a la vez. Si el argumento opcional *header* está presente y es verdadero, los guiones bajos se decodificarán como espacios.

`binascii.b2a_qp` (*data*, *quotetabs=False*, *istext=True*, *header=False*)

Convierte datos binarios en una(s) línea(s) de caracteres ASCII en codificación imprimible entre comillas. El valor de retorno son las líneas convertidas. Si el argumento opcional *quotetabs* está presente y es verdadero, se codificarán todas los tabs y espacios. Si el argumento opcional *istext* está presente y es verdadero, las nuevas líneas no se codifican, pero se codificarán los espacios en blanco finales. Si el argumento opcional *header* está presente y es verdadero, los espacios se codificarán como guiones bajos por: [rfc: 1522](#). Si el argumento opcional *header* está presente y es falso, los caracteres de nueva línea también se codificarán; de lo contrario, la conversión de salto de línea podría dañar el flujo de datos binarios.

`binascii.a2b_hqx` (*string*)

Convierte datos ASCII con formato binhex4 a binario, sin descomprimir RLE. La cadena debe contener un número completo de bytes binarios o (en el caso de la última porción de los datos binhex4) tener los bits restantes cero.

`binascii.rledecode_hqx` (*data*)

Realiza descompresión RLE en los datos, según el estándar binhex4. El algoritmo usa 0x90 después de un byte como indicador de repetición, seguido de un conteo. Un recuento de 0 especifica un valor de byte de 0x90. La rutina retorna los datos descomprimidos, a menos que los datos de entrada de datos terminen en un indicador de repetición huérfano, en cuyo caso se genera la excepción *Incomplete*.

Distinto en la versión 3.2: Acepta solo objetos *bytestring* o *bytearray* como entrada.

`binascii.rlecode_hqx` (*data*)

Realiza la compresión RLE de estilo binhex4 en *data* y retorna el resultado.

`binascii.b2a_hqx` (*data*)

Realiza la traducción de binario hexbin4 a ASCII y retorna la cadena resultante. El argumento ya debe estar codificado en RLE y tener una longitud divisible por 3 (excepto posiblemente por el último fragmento).

`binascii.crc_hqx` (*data*, *value*)

Calcula un valor CRC de 16 bits de *data*, comenzando con *value* como el CRC inicial, y retorna el resultado. Utiliza el polinomio CRC-CCITT $x^{16} + x^{12} + x^5 + 1$, a menudo representado como 0x1021. Este CRC se utiliza en el formato binhex4.

`binascii.crc32` (*data* [, *value*])

Calcula CRC-32, la suma de comprobación de 32 bits de *data*, comenzando con un CRC inicial de *value*. El CRC inicial predeterminado es cero. El algoritmo es consistente con la suma de verificación del archivo ZIP.

Dado que el algoritmo está diseñado para usarse como un algoritmo de suma de verificación, no es adecuado para usarlo como algoritmo hash general. Úselo de la siguiente manera

```
print(binascii.crc32(b"hello world"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

Distinto en la versión 3.0: El resultado siempre está sin firmar. Para generar el mismo valor numérico en todas las versiones y plataformas de Python, use `crc32(data) & 0xffffffff`.

`binascii.b2a_hex(data[, sep[, bytes_per_sep=1]])`
`binascii.hexlify(data[, sep[, bytes_per_sep=1]])`

Retorna la representación hexadecimal del binario *data*. Cada byte de *data* se convierte en la representación hexadecimal de 2 dígitos correspondiente. Por lo tanto, el objeto de bytes retornado es el doble de largo que la longitud de *data*.

Una funcionalidad similar (pero que retorna una cadena de texto) también es convenientemente accesible usando el método `bytes.hex()`.

Si se especifica *sep*, debe ser un solo carácter *str* o un objeto de bytes. Se insertará en la salida después de cada *bytes_per_sep* bytes de entrada. La ubicación del separador se cuenta desde el extremo derecho de la salida de forma predeterminada; si desea contar desde el izquierdo, proporcione un valor negativo *bytes_per_sep*.

```
>>> import binascii
>>> binascii.b2a_hex(b'\xb9\x01\xef')
b'b901ef'
>>> binascii.hexlify(b'\xb9\x01\xef', '-')
b'b9-01-ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b'_', 2)
b'b9_01ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b' ', -2)
b'b901 ef'
```

Distinto en la versión 3.8: Se agregaron los parámetros *sep* y *bytes_per_sep*.

`binascii.a2b_hex(hexstr)`
`binascii.unhexlify(hexstr)`

Retorna los datos binarios representados por la cadena hexadecimal *hexstr*. Esta función es la inversa de `b2a_hex()`. *hexstr* debe contener un número par de dígitos hexadecimales (que pueden ser mayúsculas o minúsculas), de lo contrario se produce una excepción `Error`.

Funcionalidad similar (aceptar sólo argumentos de cadena de texto, pero más liberal hacia espacios en blanco) también es accesible mediante el método de clase `bytes.fromhex()`.

exception binascii.Error

Excepción provocada por errores. Estos suelen ser errores de programación.

exception binascii.Incomplete

Excepción provocada por datos incompletos. Por lo general, estos no son errores de programación, pero se pueden controlar leyendo un poco más de datos e intentándolo de nuevo.

Ver también:

Módulo `base64` Support for RFC compliant base64-style encoding in base 16, 32, 64, and 85.

Módulo `binhex` Soporte para el formato *binhex* utilizado en Macintosh.

Módulo `uu` Soporte para codificación *UU* usada en Unix.

Módulo `quopri` Soporte para codificación imprimible entre comillas utilizada en mensajes de correo electrónico MIME.

19.9 quopri — Codificar y decodificar datos MIME imprimibles entre comillas

Código fuente: [Lib/quopri.py](#)

Este módulo realiza la codificación y decodificación de transporte imprimible entre comillas, tal como se define en [RFC 1521](#): «MIME (Multipurpose Internet Mail Extensions) Parte Uno: Mecanismos para especificar y describir el formato de los cuerpos de mensajes de Internet». La codificación imprimible entre comillas está diseñada para datos donde hay relativamente pocos caracteres no imprimibles; el esquema de codificación base64 disponible a través del módulo [base64](#) es más compacto si hay muchos caracteres de este tipo, como cuando se envía un archivo gráfico.

`quopri.decode(input, output, header=False)`

Descodificar el contenido del archivo *input* y escribir los datos binarios descodificados resultantes en el archivo *output*. *input* y *output* deben ser *objetos de archivo binario*. Si el argumento opcional *header* está presente y true, el carácter de subrayado se descodificará como espacio. Esto se utiliza para decodificar encabezados codificados en «Q» como se describe en [RFC 1522](#): «MIME (Multipurpose Internet Mail Extensions) Parte dos: Extensiones de encabezado de mensaje para texto no ASCII».

`quopri.encode(input, output, quotetabs, header=False)`

Codifique el contenido del archivo *input* y escriba los datos imprimibles entre comillas resultantes en el archivo *output*. *input* y *output* deben ser *objetos de archivo binario*. *quotetabs*, un indicador no opcional que controla si codificar espacios incrustados y pestañas; cuando true codifica dicho espacio en blanco incrustado, y cuando false los deja sin codificar. Tenga en cuenta que los espacios y pestañas que aparecen al final de las líneas siempre están codificados, según [RFC 1521](#). *header* es un indicador que controla si los espacios están codificados como guiones bajos según [RFC 1522](#).

`quopri.decodestring(s, header=False)`

Como `decode()`, excepto que acepta una fuente *bytes* y retorna el correspondiente *bytes* decodificado.

`quopri.encodestring(s, quotetabs=False, header=False)`

Como `encode()`, excepto que acepta un origen *bytes* y retorna el codificado correspondiente *bytes*. De forma predeterminada, envía un valor False al parámetro *quotetabs* de la función `encode()`.

Ver también:

Módulo [base64](#) Codificar y decodificar datos MIME base64

19.10 uu — Codifica y decodifica archivos UUEncode

Código fuente: [Lib/uu.py](#)

Este módulo codifica y decodifica archivos en formato UUEncode, permitiendo la transmisión de datos binarios arbitrarios sobre conexiones de solo ASCII. Allí donde se espera un archivo como argumento, los métodos aceptan un objeto similar a un archivo. Por compatibilidad con versiones anteriores, una cadena que contenga un nombre de ruta también es aceptada, y el archivo correspondiente será abierto para lectura y escritura; el nombre de ruta '-' es entendido como la entrada o salida estándar. Sin embargo, esta interface es obsoleta; es mejor que el invocador abra el archivo por sí mismo y asegurarse de que, cuando se requiera, el modo sea 'rb' o 'wb' en Windows.

Este código fue contribuido por Lance Ellinghouse y modificado por Jack Jansen.

El módulo *uu* define las siguientes funciones:

`uu.encode(in_file, out_file, name=None, mode=None, *, backtick=False)`

Codificar con UUEncode el archivo *in_file* en el archivo *out_file*. El archivo codificado con UUEncode tendrá un encabezado especificando *name* y *mode* como valores por defecto para los resultados de la decodificación del archivo. Los valores por defecto iniciales son tomados de *in_file*, o '-' y 0o666 respectivamente. Si *backtick* es verdadero, los ceros son representados por '`' en lugar de espacios.

Distinto en la versión 3.7: Agregado el parámetro *backtick*.

`uu.decode(in_file, out_file=None, mode=None, quiet=False)`

Esta invocación decodifica el archivo codificado en UUEncode *in_file* y coloca el resultado en el archivo *out_file*. Si *out_file* es un nombre de ruta, *mode* es usado para establecer los bits de permiso si el archivo debe ser creado. Valores por defecto para *out_file* y *mode* se toman del encabezado UUEncode. Sin embargo, si el archivo especificado en el encabezado existe, se genera un `uu.Error`.

`decode()` puede imprimir una advertencia a error estándar si la entrada fue producida por un UUEncoder incorrecto y Python pudo recobrarse de ese error. Si se establece *quiet* a un valor verdadero se suprime esta advertencia.

exception `uu.Error`

Subclase de `Exception`, esta puede ser generada por `uu.decode()` en diversas situaciones, como las descritas arriba, pero también si se incluye un encabezado mal formado o un archivo de entrada truncado.

Ver también:

Módulo `binascii` Módulo de soporte que contiene conversiones de ASCII a binario y de binario a ASCII.

Herramientas Para Procesar Formatos de Marcado Estructurado

Python soporta una variedad de módulos para trabajar con varias formas de almacenar datos de forma estructurada. Esto incluye módulos para trabajar con el Lenguaje de Marcado Estructurado General (SGML) y el Lenguaje de Marcado de Hipertexto (HTML), y varias interfaces para trabajar con el Lenguaje de Marcado Estructurado Extensible (XML).

20.1 `html` — Compatibilidad con el Lenguaje de marcado de hipertexto

Código fuente: `Lib/html/__init__.py`

Este módulo define utilidades para manipular HTML.

`html.escape(s, quote=True)`

Convierte los caracteres `&`, `<` y `>` de la cadena de caracteres `s` en secuencias seguras HTML. Utilízalo si necesitas mostrar texto que pueda contener tales caracteres en HTML. Si el flag opcional `quote` es `true`, también se traducen los caracteres `"` y `'`; esto ayuda a la inserción en el valor de un atributo HTML delimitado por comillas, como en ``.

Nuevo en la versión 3.2.

`html.unescape(s)`

Convierte todas las referencias de caracteres numéricos y con nombre (por ejemplo `>`, `>`, `>`) de la cadena de caracteres `s` a los caracteres Unicode correspondientes. Esta función utiliza las reglas definidas por el estándar HTML 5 para las referencias de caracteres válidas e inválidas, y la *lista de referencia de caracteres con nombre de HTML 5*.

Nuevo en la versión 3.4.

Los submódulos del paquete `html` son:

- `html.parser` – Analizador sintáctico simple de HTML y XHTML
- `html.entities` – Definición general de entidades HTML

20.2 `html.parser` — Analizador simple de HTML y XHTML

Código fuente: [Lib/html/parser.py](#)

Este módulo define una clase `HTMLParser` que sirve como base para analizar archivos de texto formateados en HTML (*HyperText Mark-up Language*) y XHTML.

class `html.parser.HTMLParser` (*, `convert_charrefs=True`)

Cree una instancia de analizador capaz de analizar marcado no válido.

Si `convert_charrefs` es `True` (el valor predeterminado), todas las referencias de caracteres (excepto las de los elementos `script/style`) se convierten automáticamente en los caracteres Unicode correspondientes.

Una instancia de `HTMLParser` se alimenta de datos HTML y llama a métodos de manejo cuando se encuentran etiquetas de inicio, etiquetas finales, texto, comentarios y otros elementos de marcado. El usuario debe subclassificar `HTMLParser` y anular sus métodos para implementar el comportamiento deseado.

Este analizador no verifica que las etiquetas finales coincidan con las etiquetas iniciales ni llame al manejador de etiquetas finales para los elementos que se cierran implícitamente al cerrar un elemento externo.

Distinto en la versión 3.4: argumento de palabra clave `convert_charrefs` agregado.

Distinto en la versión 3.5: El valor predeterminado para el argumento `convert_charrefs` ahora es `True`.

20.2.1 Aplicación ejemplo de un analizador sintáctico (*parser*) de HTML

Como ejemplo básico, a continuación hay un analizador HTML simple que usa la clase `HTMLParser` para imprimir etiquetas de inicio, etiquetas finales y datos a medida que se encuentran:

```
from html.parser import HTMLParser

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)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
          '<body><h1>Parse me!</h1></body></html>')
```

La salida será entonces:

```
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
```

20.2.2 Métodos `HTMLParser`

instancias de `HTMLParser` tienen los siguientes métodos:

`HTMLParser.feed(data)`

Alimente un poco de texto al analizador. Se procesa en la medida en que consta de elementos completos; los datos incompletos se almacenan en el búfer hasta que se introducen más datos o se llama a `close()`. *data* debe ser *str*.

`HTMLParser.close()`

Fuerce el procesamiento de todos los datos almacenados como si fueran seguidos por una marca de fin de archivo. Este método puede ser redefinido por una clase derivada para definir un procesamiento adicional al final de la entrada, pero la versión redefinida siempre debe llamar a `HTMLParser` método de clase base `close()`.

`HTMLParser.reset()`

Restablecer la instancia. Pierde todos los datos no procesados. Esto se llama implícitamente en el momento de la instanciación.

`HTMLParser.getpos()`

Retorna el número de línea actual y el desplazamiento.

`HTMLParser.get_starttag_text()`

Retorna el texto de la etiqueta de inicio abierta más recientemente. Normalmente, esto no debería ser necesario para el procesamiento estructurado, pero puede ser útil para tratar con HTML «como implementado» o para volver a generar entradas con cambios mínimos (se puede preservar el espacio en blanco entre los atributos, etc.).

Los siguientes métodos se invocan cuando se encuentran datos o elementos de marcado y deben anularse en una subclase. Las implementaciones de la clase base no hacen nada (excepto `handle_startendtag()`):

`HTMLParser.handle_starttag(tag, attrs)`

Este método se llama para manejar el inicio de una etiqueta (por ejemplo, `<div id="main">`).

El argumento *tag* es el nombre de la etiqueta convertida a minúsculas. El argumento *attrs* es una lista de pares (*nombre*, *valor*) que contienen los atributos encontrados dentro de los corchetes `<>` de la etiqueta. El *name* se traducirá a minúsculas, se eliminarán las comillas en el *value* y se reemplazarán las referencias de caracteres y entidades.

Por ejemplo, para la etiqueta ``, este método se llamaría como `handle_starttag('a', [('href', 'https : //www.cwi.nl/ ')])`.

Todas las referencias de entidad de `html.entities` se reemplazan en los valores de los atributos.

`HTMLParser.handle_endtag(tag)`

Este método se llama para manejar la etiqueta final de un elemento (por ejemplo, `</div>`)

El argumento *tag* es el nombre de la etiqueta convertida a minúsculas.

`HTMLParser.handle_startendtag(tag, attrs)`

Similar a `handle_starttag()`, pero llamado cuando el analizador encuentra una etiqueta vacía de estilo XHTML (``). Este método puede ser anulado por subclases que requieren esta información léxica particular; la implementación predeterminada simplemente llama `handle_starttag()` y `handle_endtag()`.

`HTMLParser.handle_data(data)`

Este método se llama para procesar datos arbitrarios (por ejemplo, nodos de texto y el contenido de `<script>...</script>` y `<style>...</style>`).

`HTMLParser.handle_entityref(name)`

Este método se llama para procesar una referencia de caracteres con nombre del formulario `&name;` (por ejemplo, `>`), donde *name* es una referencia de entidad general (por ejemplo, `'gt'`). Este método nunca se llama si `convert_charrefs` es `True`.

`HTMLParser.handle_charref(name)`

Este método se llama para procesar referencias de caracteres numéricos decimales y hexadecimales de la forma

`&#xNN;` y `&#xNNN;`. Por ejemplo, el equivalente decimal para `>` es `>`, mientras que el hexadecimal es `>`; en este caso, el método recibirá `'62'` o `'x3E'`. Este método nunca se llama si `convert_charrefs` es `True`.

`HTMLParser.handle_comment(data)`

Este método se llama cuando se encuentra un comentario (por ejemplo, `<!--comment-->`).

Por ejemplo, el comentario `<!--comment-->` hará que se llame a este método con el argumento `'comment'`.

El contenido de los comentarios condicionales de Internet Explorer (*condcoms*) también se enviará a este método, por lo tanto, para `<!--[if IE 9]>IE9-specific content<![endif]-->`, este método recibirá `'[if IE 9]>IE9-specific content<![endif]'`.

`HTMLParser.handle_decl(decl)`

Este método se llama para manejar una declaración de tipo de documento HTML (por ejemplo, `<!DOCTYPE html>`).

El parámetro *decl* será todo el contenido de la declaración dentro del `<!...>` *markup* (por ejemplo, `'DOCTYPE html'`).

`HTMLParser.handle_pi(data)`

Método llamado cuando se encuentra una instrucción de procesamiento. El parámetro *data* contendrá toda la instrucción de procesamiento. Por ejemplo, para la instrucción de procesamiento `<?proc color='red'>`, este método se llamaría como `handle_pi("proc color='red'")`. Está destinado a ser anulado por una clase derivada; La implementación de la clase base no hace nada.

Nota: La clase `HTMLParser` utiliza las reglas sintácticas SGML para procesar instrucciones. Una instrucción de procesamiento XHTML que use el `'?'` final hará que se incluya el `'?'` en *data*.

`HTMLParser.unknown_decl(data)`

Se llama a este método cuando el analizador lee una declaración no reconocida.

El parámetro *data* será el contenido completo de la declaración dentro del marcado `<![...]>`. A veces es útil ser reemplazado por una clase derivada. La implementación de la clase base no hace nada.

20.2.3 Ejemplos

La siguiente clase implementa un analizador que se utilizará para ilustrar más ejemplos:

```
from html.parser import HTMLParser
from html.entities 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 = chr(name2codepoint[name])
        print("Named ent:", c)
```

(continué en la próxima página)

(proviene de la página anterior)

```

def handle_charref(self, name):
    if name.startswith('#x'):
        c = chr(int(name[1:], 16))
    else:
        c = chr(int(name))
    print("Num ent  :", c)

def handle_decl(self, data):
    print("Decl      :", data)

parser = MyHTMLParser()

```

Analizando un *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"

```

Analizando un elemento con algunos atributos y un título:

```

>>> parser.feed('')
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

```

El contenido de los elementos `script` y `style` se retorna tal cual, sin más análisis

```

>>> 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

```

Analizando comentarios:

```

>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment   : a comment
Comment   : [if IE 9]>IE-specific content<![endif]

```

Analizar referencias de caracteres con nombre y numéricos y convertirlos al carácter correcto (nota: estas 3 referencias son todas equivalentes a '>'):

```

>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent   : >
Num ent   : >

```

La alimentación de fragmentos incompletos a `feed()` funciona, pero `handle_data()` podría llamarse más de una vez (a menos que `convert_charrefs` esté configurado como `True`):

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data      : buff
Data      : ered
Data      : text
End tag   : span
```

Analizar HTML no válido (por ejemplo, atributos sin comillas) también funciona:

```
>>> 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
```

20.3 `html.entities` — Definiciones de entidades generales HTML

Código fuente: [Lib/html/entities.py](#)

Este módulo define cuatro diccionarios `html5`, `name2codepoint`, `codepoint2name`, y `entitydefs`

`html.entities.html5`

Un diccionario que asigna referencias de caracteres con nombre HTML5¹ a los caracteres Unicode equivalentes, p. Ej. `html5['gt;'] == '>'`. Tenga en cuenta que el punto y coma al final está incluido en el nombre (por ejemplo, `'gt;'`), sin embargo, algunos de los nombres son aceptados por el estándar incluso sin el punto y coma: en este caso, el nombre está presente con y sin el `' '`. Consulte también `html.unescape()`.

Nuevo en la versión 3.3.

`html.entities.entitydefs`

Un diccionario que asigna definiciones de entidad XHTML 1.0 a su texto de reemplazo en ISO Latin-1.

`html.entities.name2codepoint`

Un diccionario que asigna nombres de entidades HTML a los puntos de código Unicode.

`html.entities.codepoint2name`

Un diccionario que asigna puntos de código Unicode a nombres de entidades HTML.

Notas al pie

20.4 Módulos de procesamiento XML

Código fuente: [Lib/xml/](#)

Las interfaces de Python para procesar XML están agrupadas en el paquete `xml`.

¹ Vea <https://www.w3.org/TR/html5/syntax.html#named-character-references>

Advertencia: The XML modules are not secure against erroneous or maliciously constructed data. If you need to parse untrusted or unauthenticated data see the [Vulnerabilidades XML](#) and [The defusedxml Package](#) sections.

Es importante tener en cuenta que los módulos del paquete `xml` requieren que haya al menos un analizador XML compatible con SAX disponible. El analizador Expat se incluye con Python, por lo que el módulo `xml.parsers.expat` siempre estará disponible.

La documentación de los paquetes `xml.dom` y `xml.sax` es la definición de los enlaces de Python para las interfaces DOM y SAX.

Los submódulos de manejo de XML son:

- `xml.etree.ElementTree`: la API ElementTree, un procesador de XML simple y ligero
- `xml.dom`: la definición de la API DOM
- `xml.dom.minidom`: una implementación mínima de DOM
- `xml.dom.pulldom`: soporte para la construcción de árboles DOM parciales
- `xml.sax`: clases base SAX2 y funciones de conveniencia
- `xml.parsers.expat`: el enlace del analizador Expat

20.4.1 Vulnerabilidades XML

Los módulos de procesamiento XML no son seguros contra datos contruidos malintencionadamente. Un atacante puede abusar de las características XML para llevar a cabo ataques de denegación de servicio, acceder a archivos locales, generar conexiones de red a otras máquinas o eludir firewalls.

En la tabla siguiente se ofrece una visión general de los ataques conocidos y si los distintos módulos son vulnerables a ellos.

tipo	sax	etree	minidom	pulldom	xmlrpc
mil millones de risas	Vulnerable (1)	Vulnerable (1)	Vulnerable (1)	Vulnerable (1)	Vulnerable (1)
explosión cuadrática	Vulnerable (1)	Vulnerable (1)	Vulnerable (1)	Vulnerable (1)	Vulnerable (1)
expansión de entidad externa	Safe (5)	Safe (2)	Safe (3)	Safe (5)	Seguro (4)
Recuperación de DTD	Safe (5)	Seguro	Seguro	Safe (5)	Seguro
bomba de descompresión	Seguro	Seguro	Seguro	Seguro	Vulnerable
large tokens	Vulnerable (6)	Vulnerable (6)	Vulnerable (6)	Vulnerable (6)	Vulnerable (6)

1. Expat 2.4.1 and newer is not vulnerable to the «billion laughs» and «quadratic blowup» vulnerabilities. Items still listed as vulnerable due to potential reliance on system-provided libraries. Check `pyexpat.EXPAT_VERSION`.
2. `xml.etree.ElementTree` no expande entidades externas y lanza un `ParserError` cuando se produce una entidad.
3. `xml.dom.minidom` no expande entidades externas y simplemente retorna la entidad no expandida literalmente.
4. `xmlrpclib` no expande entidades externas y las omite.
5. Desde Python 3.7.1, las entidades generales externas ya no se procesan de forma predeterminada.
6. Expat 2.6.0 and newer is not vulnerable to denial of service through quadratic runtime caused by parsing large tokens. Items still listed as vulnerable due to potential reliance on system-provided libraries. Check `pyexpat.EXPAT_VERSION`.

mil millones de risas / expansión exponencial de entidad El ataque [Billion Laughs](#), también conocido como expansión exponencial de entidades, utiliza varios niveles de entidades anidadas. Cada entidad hace referencia a otra entidad varias veces y la definición de entidad final contiene una cadena pequeña. La expansión exponencial da como resultado varios gigabytes de texto y consume mucha memoria y tiempo de CPU.

expansión de entidad de explosión cuadrática Un ataque de explosión cuadrática es similar a un ataque de [Billion Laughs](#); también abusa de la expansión de entidad. En lugar de entidades anidadas, repite una entidad grande con un par de miles de caracteres una y otra vez. El ataque no es tan eficaz como el caso exponencial, pero evita desencadenar contramedidas del analizador que prohíben entidades profundamente anidadas.

expansión de entidad externa Las declaraciones de entidad pueden contener algo más que texto para su reemplazo. También pueden apuntar a recursos externos o archivos locales. El analizador XML tiene acceso al recurso e incrusta el contenido en el documento XML.

Recuperación de DTD Algunas bibliotecas XML como `xml.dom.pulldom` de Python recuperan definiciones de tipo de documento de ubicaciones remotas o locales. La característica tiene implicaciones similares a las del problema de expansión de entidades externas.

bomba de descompresión Las bombas de descompresión (también conocidas como [ZIP bomb](#)) se aplican a todas las bibliotecas XML que pueden analizar secuencias XML comprimidas, como secuencias HTTP comprimidas con gzip o archivos comprimidos por LZMA. Para un atacante puede reducir la cantidad de datos transmitidos en magnitudes de tres o más.

large tokens Expat needs to re-parse unfinished tokens; without the protection introduced in Expat 2.6.0, this can lead to quadratic runtime that can be used to cause denial of service in the application parsing XML. The issue is known as [CVE-2023-52425](#).

La documentación de [defusedxml](#) en PyPI tiene más información sobre todos los vectores de ataque conocidos con ejemplos y referencias.

20.4.2 The defusedxml Package

[defusedxml](#) es un paquete Python puro con subclases modificadas de todos los analizadores XML stdlib que impiden cualquier operación potencialmente malintencionada. Se recomienda el uso de este paquete para cualquier código de servidor que analice datos XML que no sean de confianza. El paquete también incluye ataques de ejemplo y documentación ampliada sobre más vulnerabilidades XML, como la inyección de XPath.

20.5 xml.etree.ElementTree — The ElementTree XML API

Source code: [Lib/xml/etree/ElementTree.py](#)

The `xml.etree.ElementTree` module implements a simple and efficient API for parsing and creating XML data.

Distinto en la versión 3.3: This module will use a fast implementation whenever available. The `xml.etree.cElementTree` module is deprecated.

Advertencia: The `xml.etree.ElementTree` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [Vulnerabilidades XML](#).

20.5.1 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 can import this data by reading from a file:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Or directly 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'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

Children are nested, and we can access specific child nodes by index:

```
>>> root[0][1].text
'2008'
```

Nota: Not all elements of the XML input will end up as elements of the parsed tree. Currently, this module skips over any XML comments, processing instructions, and document type declarations in the input. Nevertheless, trees built using this module's API rather than parsing from XML text can have comments and processing instructions in them; they will be included when generating XML output. A document type declaration may be accessed by passing a custom *TreeBuilder* instance to the *XMLParser* constructor.

Pull API for non-blocking parsing

Most parsing functions provided by this module require the whole document to be read at once before returning any result. It is possible to use an *XMLParser* and feed data into it incrementally, but it is a push API that calls methods on a callback target, which is too low-level and inconvenient for most needs. Sometimes what the user really wants is to be able to parse XML incrementally, without blocking operations, while enjoying the convenience of fully constructed *Element* objects.

The most powerful tool for doing this is *XMLPullParser*. It does not require a blocking read to obtain the XML data, and is instead fed with data incrementally with *XMLPullParser.feed()* calls. To get the parsed XML elements, call *XMLPullParser.read_events()*. Here is an example:

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
```

The obvious use case is applications that operate in a non-blocking fashion where the XML data is being received from a socket or read incrementally from some storage device. In such cases, blocking reads are unacceptable.

Because it's so flexible, *XMLPullParser* can be inconvenient to use for simpler use-cases. If you don't mind your application blocking on reading XML data but would still like to have incremental parsing capabilities, take a look at *iterparse()*. It can be useful when you're reading a large XML document and don't want to hold it wholly in memory.

Where *immediate* feedback through events is wanted, calling method *XMLPullParser.flush()* can help reduce delay; please make sure to study the related security notes.

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>
```

(continué en la próxima página)

(proviene de la página anterior)

```

    <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'):
...     # using root.findall() to avoid removal during traversal
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')

```

Note that concurrent modification while iterating can lead to problems, just like when iterating and modifying Python lists or dicts. Therefore, the example first collects all matching elements with `root.findall()`, and only then iterates over the list of matches.

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.

20.5.2 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]")
```

For XML with namespaces, use the usual qualified `{namespace}tag` notation:

```
# All dublin-core "title" tags in the document
root.findall("./{http://purl.org/dc/elements/1.1/}title")
```

Supported XPath syntax

Syntax	Meaning
<code>tag</code>	Selects all child elements with the given tag. For example, <code>spam</code> selects all child elements named <code>spam</code> , and <code>spam/egg</code> selects all grandchildren named <code>egg</code> in all children named <code>spam</code> . <code>{namespace}*</code> selects all tags in the given namespace, <code>{*}</code> <code>spam</code> selects tags named <code>spam</code> in any (or no) namespace, and <code>{*}</code> only selects tags that are not in a namespace. Distinto en la versión 3.8: Support for star-wildcards was added.
<code>*</code>	Selects all child elements, including comments and processing instructions. For example, <code>*/egg</code> selects all grandchildren named <code>egg</code> .
<code>.</code>	Selects the current node. This is mostly useful at the beginning of the path, to indicate that it's a relative path.
<code>//</code>	Selects all subelements, on all levels beneath the current element. For example, <code>./egg</code> selects all <code>egg</code> elements in the entire tree.
<code>..</code>	Selects the parent element. Returns <code>None</code> if the path attempts to reach the ancestors of the start element (the element <code>find</code> was called on).
<code>[@attrib]</code>	Selects all elements that have the given attribute.
<code>[@attrib='value']</code>	Selects all elements for which the given attribute has the given value. The value cannot contain quotes.
<code>[tag]</code>	Selects all elements that have a child named <code>tag</code> . Only immediate children are supported.
<code>[.='text']</code>	Selects all elements whose complete text content, including descendants, equals the given <code>text</code> . Nuevo en la versión 3.7.
<code>[tag='text']</code>	Selects all elements that have a child named <code>tag</code> whose complete text content, including descendants, equals the given <code>text</code> .
<code>[position]</code>	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression <code>last()</code> (for the last position), or a position relative to the last position (e.g. <code>last()-1</code>).

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.

20.5.3 Reference

Functions

`xml.etree.ElementTree.canonicalize` (*xml_data=None*, ***, *out=None*, *from_file=None*, ***options*)

C14N 2.0 transformation function.

Canonicalization is a way to normalise XML output in a way that allows byte-by-byte comparisons and digital signatures. It reduced the freedom that XML serializers have and instead generates a more constrained XML representation. The main restrictions regard the placement of namespace declarations, the ordering of attributes, and ignorable whitespace.

This function takes an XML data string (*xml_data*) or a file path or file-like object (*from_file*) as input, converts it to the canonical form, and writes it out using the *out* file(-like) object, if provided, or returns it as a text string if not. The output file receives text, not bytes. It should therefore be opened in text mode with `utf-8` encoding.

Typical uses:

```
xml_data = "<root>...</root>"
print(canonicalize(xml_data))

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
```

(continué en la próxima página)

(proviene de la página anterior)

```
canonicalize(xml_data, out=out_file)

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(from_file="inputfile.xml", out=out_file)
```

The configuration *options* are as follows:

- *with_comments*: set to true to include comments (default: false)
- *strip_text*: set to true to strip whitespace before and after text content (default: false)
- *rewrite_prefixes*: set to true to replace namespace prefixes by «n{number}» (default: false)
- *qname_aware_tags*: a set of qname aware tag names in which prefixes should be replaced in text content (default: empty)
- *qname_aware_attrs*: a set of qname aware attribute names in which prefixes should be replaced in text content (default: empty)
- *exclude_attrs*: a set of attribute names that should not be serialised
- *exclude_tags*: a set of tag names that should not be serialised

In the option list above, «a set» refers to any collection or iterable of strings, no ordering is expected.

Nuevo en la versión 3.8.

`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.

Note that *XMLParser* skips over comments in the input instead of creating comment objects for them. An *ElementTree* will only contain comment nodes if they have been inserted into to the tree using one of the *Element* methods.

`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.

Distinto en la versión 3.8: The *dump()* function now preserves the attribute order specified by the user.

`xml.etree.ElementTree.fromstring` (*text, parser=None*)

Parses an XML section from a string constant. Same as *XML()*. *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.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.

Nuevo en la versión 3.2.

`xml.etree.ElementTree.iselement` (*element*)

Check if an object appears to be a valid element object. *element* is an element instance. Return *True* 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 sequence of events to report back. The supported events are the strings "start", "end", "comment", "pi", "start-ns" and "end-ns" (the «ns» events are used to get detailed namespace information). If *events* is omitted, only "end" events are reported. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. *parser* must be a

subclass of *XMLParser* and can only use the default *TreeBuilder* as a target. Returns an *iterator* providing (event, elem) pairs.

Note that while *iterparse()* builds the tree incrementally, it issues blocking reads on *source* (or the file it names). As such, it's unsuitable for applications where blocking reads can't be made. For fully non-blocking parsing, see *XMLPullParser*.

Nota: *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.

Obsoleto desde la versión 3.4: The *parser* argument.

Distinto en la versión 3.8: The *comment* and *pi* events were added.

`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.

Note that *XMLParser* skips over processing instructions in the input instead of creating comment objects for them. An *ElementTree* will only contain processing instruction nodes if they have been inserted into the tree using one of the *Element* methods.

`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.

Nuevo en la versión 3.2.

`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", *,
xml_declaration=None, default_namespace=None,
short_empty_elements=True)`

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). Use *encoding="unicode"* to generate a Unicode string (otherwise, a bytestring is generated). *method* is either "xml", "html" or "text" (default is "xml"). *xml_declaration*, *default_namespace* and *short_empty_elements* has the same meaning as in *ElementTree.write()*. Returns an (optionally) encoded string containing the XML data.

Nuevo en la versión 3.4: The *short_empty_elements* parameter.

Nuevo en la versión 3.8: The *xml_declaration* and *default_namespace* parameters.

Distinto en la versión 3.8: The *tostring()* function now preserves the attribute order specified by the user.

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.tostringlist(element, encoding="us-ascii", method="xml", *,  
                                   xml_declaration=None, default_namespace=None,  
                                   short_empty_elements=True)
```

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). Use *encoding*="unicode" to generate a Unicode string (otherwise, a bytestring is generated). *method* is either "xml", "html" or "text" (default is "xml"). *xml_declaration*, *default_namespace* and *short_empty_elements* has the same meaning as in *ElementTree.write()*. Returns a list of (optionally) encoded strings containing the XML data. It does not guarantee any specific sequence, except that `b"".join(tostringlist(element)) == tostring(element)`.

Nuevo en la versión 3.2.

Nuevo en la versión 3.4: The *short_empty_elements* parameter.

Nuevo en la versión 3.8: The *xml_declaration* and *default_namespace* parameters.

Distinto en la versión 3.8: The *tostringlist()* function now preserves the attribute order specified by the user.

```
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.

20.5.4 XInclude support

This module provides limited support for *XInclude* directives, via the `xml.etree.ElementInclude` helper module. This module can be used to insert subtrees and text strings into element trees, based on information in the tree.

Example

Here's an example that demonstrates use of the *XInclude* module. To include an XML document in the current document, use the `{http://www.w3.org/2001/XInclude}include` element and set the **parse** attribute to "xml", and use the **href** attribute to specify the document to include.

```
<?xml version="1.0"?>  
<document xmlns:xi="http://www.w3.org/2001/XInclude">  
  <xi:include href="source.xml" parse="xml" />  
</document>
```

By default, the **href** attribute is treated as a file name. You can use custom loaders to override this behaviour. Also note that the standard helper does not support XPointer syntax.

To process this file, load it as usual, and pass the root element to the `xml.etree.ElementTree` module:

```
from xml.etree import ElementTree, ElementInclude  
  
tree = ElementTree.parse("document.xml")  
root = tree.getroot()  
  
ElementInclude.include(root)
```

The *ElementInclude* module replaces the `{http://www.w3.org/2001/XInclude}include` element with the root element from the **source.xml** document. The result might look something like this:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <para>This is a paragraph.</para>
</document>
```

If the **parse** attribute is omitted, it defaults to «xml». The **href** attribute is required.

To include a text document, use the {http://www.w3.org/2001/XInclude}include element, and set the **parse** attribute to «text»:

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) <xi:include href="year.txt" parse="text" />.
</document>
```

The result might look something like:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) 2003.
</document>
```

20.5.5 Reference

Functions

`xml.etree.ElementInclude.default_loader(href, parse, encoding=None)`

Default loader. This default loader reads an included resource from disk. *href* is a URL. *parse* is for parse mode either «xml» or «text». *encoding* is an optional text encoding. If not given, encoding is `utf-8`. Returns the expanded resource. If the parse mode is "xml", this is an `ElementTree` instance. If the parse mode is «text», this is a Unicode string. If the loader fails, it can return `None` or raise an exception.

`xml.etree.ElementInclude.include(elem, loader=None)`

This function expands XInclude directives. *elem* is the root element. *loader* is an optional resource loader. If omitted, it defaults to `default_loader()`. If given, it should be a callable that implements the same interface as `default_loader()`. Returns the expanded resource. If the parse mode is "xml", this is an `ElementTree` instance. If the parse mode is «text», this is a Unicode string. If the loader fails, it can return `None` or raise an exception.

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 element's internal list of subelements. Raises `TypeError` if *subelement* is not an `Element`.

extend(subelements)

Appends *subelements* from a sequence object with zero or more elements. Raises `TypeError` if a subelement is not an `Element`.

Nuevo en la versión 3.2.

find(match, namespaces=None)

Finds the first subelement matching *match*. *match* may be a tag name or a `path`. Returns an element instance or `None`. *namespaces* is an optional mapping from namespace prefix to full name. Pass `' '` as prefix to move all unprefix tag names in the expression into the given namespace.

findall(match, namespaces=None)

Finds all matching subelements, by tag name or `path`. Returns a list containing all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name. Pass `' '` as prefix to move all unprefix tag names in the expression into the given namespace.

findtext(match, default=None, namespaces=None)

Finds text for the first subelement matching *match*. *match* may be a tag name or a `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. *namespaces* is an optional mapping from namespace prefix to full name. Pass `' '` as prefix to move all unprefix tag names in the expression into the given namespace.

getchildren()

Deprecated since version 3.2, will be removed in version 3.9: Use `list(elem)` or iteration.

getiterator(tag=None)

Deprecated since version 3.2, will be removed in version 3.9: Use method `Element.iter()` instead.

insert(index, subelement)

Inserts *subelement* at the given position in this element. Raises `TypeError` if *subelement* is not an `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.

Nuevo en la versión 3.2.

iterfind(match, namespaces=None)

Finds all matching subelements, by tag name or *path*. Returns an iterable yielding all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

Nuevo en la versión 3.2.

itertext()

Creates a text iterator. The iterator loops over this element and all subelements, in document order, and returns all inner text.

Nuevo en la versión 3.2.

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")
```

Prior to Python 3.8, the serialisation order of the XML attributes of elements was artificially made predictable by sorting the attributes by their name. Based on the now guaranteed ordering of dicts, this arbitrary reordering was removed in Python 3.8 to preserve the order in which attributes were originally parsed or created by user code.

In general, user code should try not to depend on a specific ordering of attributes, given that the [XML Information Set](#) explicitly excludes the attribute order from conveying information. Code should be prepared to deal with any ordering on input. In cases where deterministic XML output is required, e.g. for cryptographic signing or test data sets, canonical serialisation is available with the `canonicalize()` function.

In cases where canonical output is not applicable but a specific attribute order is still desirable on output, code should aim for creating the attributes directly in the desired order, to avoid perceptual mismatches for readers of the code. In cases where this is difficult to achieve, a recipe like the following can be applied prior to serialisation to enforce an order independently from the `Element` creation:

```
def reorder_attributes(root):
    for el in root.iter():
        attrib = el.attrib
        if len(attrib) > 1:
            # adjust attribute order, e.g. by sorting
            attribs = sorted(attrib.items())
            attrib.clear()
            attrib.update(attribs)
```

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, namespaces=None*)

Same as `Element.find()`, starting at the root of the tree.

findall (*match, namespaces=None*)

Same as `Element.findall()`, starting at the root of the tree.

findtext (*match, default=None, namespaces=None*)

Same as `Element.findtext()`, starting at the root of the tree.

getiterator (*tag=None*)

Deprecated since version 3.2, will be removed in version 3.9: 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, namespaces=None*)

Same as `Element.iterfind()`, starting at the root of the tree.

Nuevo en la versión 3.2.

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", *, short_empty_elements=True*)

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 or Unicode (default is `None`). *default_namespace* sets the default XML namespace (for «xmlns»). *method* is either "xml", "html" or "text" (default is "xml"). The keyword-only *short_empty_elements* parameter controls the formatting of elements that contain no content. If `True` (the default), they are emitted as a single self-closed tag, otherwise they are emitted as a pair of start/end tags.

The output is either a string (*str*) or binary (*bytes*). This is controlled by the *encoding* argument. If *encoding* is "unicode", the output is a string; otherwise, it's binary. Note that this may conflict with

the type of *file* if it's an open *file object*; make sure you do not try to write a string to a binary stream and vice versa.

Nuevo en la versión 3.4: The *short_empty_elements* parameter.

Distinto en la versión 3.8: The *write()* method now preserves the attribute order specified by the user.

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*, *, *comment_factory=None*, *pi_factory=None*, *insert_comments=False*, *insert_pis=False*)

Generic element structure builder. This builder converts a sequence of start, data, end, comment and pi 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.

element_factory, when given, must be a callable accepting two positional arguments: a tag and a dict of attributes. It is expected to return a new element instance.

The *comment_factory* and *pi_factory* functions, when given, should behave like the *Comment()* and *ProcessingInstruction()* functions to create comments and processing instructions. When not given, the default factories will be used. When *insert_comments* and/or *insert_pis* is true, comments/pis will be inserted into the tree if they appear within the root element (but not outside of it).

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.

comment (*text*)

Creates a comment with the given *text*. If `insert_comments` is true, this will also add it to the tree.

Nuevo en la versión 3.8.

pi (*target*, *text*)

Creates a comment with the given *target* name and *text*. If `insert_pis` is true, this will also add it to the tree.

Nuevo en la versión 3.8.

In addition, a custom `TreeBuilder` object can provide the following methods:

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.

Nuevo en la versión 3.2.

start_ns (*prefix*, *uri*)

Is called whenever the parser encounters a new namespace declaration, before the `start()` callback for the opening element that defines it. *prefix* is `' '` for the default namespace and the declared namespace prefix name otherwise. *uri* is the namespace URI.

Nuevo en la versión 3.8.

end_ns (*prefix*)

Is called after the `end()` callback of an element that declared a namespace prefix mapping, with the name of the *prefix* that went out of scope.

Nuevo en la versión 3.8.

```
class xml.etree.ElementTree.C14NWriterTarget (write, *, with_comments=False,
                                             strip_text=False, rewrite_prefixes=False,
                                             qname_aware_tags=None, qname_aware_attrs=None,
                                             exclude_tags=None)
```

A C14N 2.0 writer. Arguments are the same as for the `canonicalize()` function. This class does not build a tree but translates the callback events directly into a serialised form using the `write` function.

Nuevo en la versión 3.8.

XMLParser Objects

```
class xml.etree.ElementTree.XMLParser (*, target=None, encoding=None)
```

This class is the low-level building block of the module. It uses `xml.parsers.expat` for efficient, event-based parsing of XML. It can be fed XML data incrementally with the `feed()` method, and parsing events are translated to a push API - by invoking callbacks on the *target* object. If *target* is omitted, the standard `TreeBuilder` is used. If *encoding*¹ is given, the value overrides the encoding specified in the XML file.

Distinto en la versión 3.8: Parameters are now *keyword-only*. The `html` argument no longer supported.

close ()

Finishes feeding data to the parser. Returns the result of calling the `close()` method of the *target* passed during construction; by default, this is the toplevel document element.

feed(data)

Feeds data to the parser. *data* is encoded data.

flush()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of *flush()* temporarily disables reparsing deferral with Expat (if currently enabled) and triggers a reparsing. Disabling reparsing deferral has security consequences; please see *xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()* for details.

Note that *flush()* has been backported to some prior releases of CPython as a security fix. Check for availability of *flush()* using *hasattr()* if used in code running across a variety of Python versions.

Nuevo en la versión 3.8.19.

XMLParser.feed() calls *target's* *start(tag, attrs_dict)* method for each opening tag, its *end(tag)* method for each closing tag, and data is processed by method *data(data)*. For further supported callback methods, see the *TreeBuilder* class. *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.
...         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
```

XMLPullParser Objects

class `xml.etree.ElementTree.XMLPullParser` (*events=None*)

A pull parser suitable for non-blocking applications. Its input-side API is similar to that of `XMLParser`, but instead of pushing calls to a callback target, `XMLPullParser` collects an internal list of parsing events and lets the user read from it. *events* is a sequence of events to report back. The supported events are the strings "start", "end", "comment", "pi", "start-ns" and "end-ns" (the «ns» events are used to get detailed namespace information). If *events* is omitted, only "end" events are reported.

feed (*data*)

Feed the given bytes data to the parser.

flush ()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of `flush()` temporarily disables reparsing deferral with Expat (if currently enabled) and triggers a reparsing. Disabling reparsing deferral has security consequences; please see `xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()` for details.

Note that `flush()` has been backported to some prior releases of CPython as a security fix. Check for availability of `flush()` using `hasattr()` if used in code running across a variety of Python versions.

Nuevo en la versión 3.8.19.

close ()

Signal the parser that the data stream is terminated. Unlike `XMLParser.close()`, this method always returns `None`. Any events not yet retrieved when the parser is closed can still be read with `read_events()`.

read_events ()

Return an iterator over the events which have been encountered in the data fed to the parser. The iterator yields (*event*, *elem*) pairs, where *event* is a string representing the type of event (e.g. "end") and *elem* is the encountered `Element` object, or other context value as follows.

- start, end: the current `Element`.
- comment, pi: the current comment / processing instruction
- start-ns: a tuple (*prefix*, *uri*) naming the declared namespace mapping.
- end-ns: `None` (this may change in a future version)

Events provided in a previous call to `read_events()` will not be yielded again. Events are consumed from the internal queue only when they are retrieved from the iterator, so multiple readers iterating in parallel over iterators obtained from `read_events()` will have unpredictable results.

Nota: `XMLPullParser` 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.

Nuevo en la versión 3.4.

Distinto en la versión 3.8: The `comment` and `pi` events were added.

Exceptions

class `xml.etree.ElementTree.ParseError`

XML parse error, raised by the various parsing methods in this module when parsing fails. The string representation of an instance of this exception will contain a user-friendly error message. In addition, it will have the following attributes available:

code

A numeric error code from the expat parser. See the documentation of `xml.parsers.expat` for the list of error codes and their meanings.

position

A tuple of *line*, *column* numbers, specifying where the error occurred.

20.6 `xml.dom` — El API del Modelo de Objetos del Documento

Código Fuente: `Lib/xml/dom/__init__.py`

El Modelo de Objetos del Documento, o «DOM» por sus siglas en inglés, es un lenguaje API del Consorcio *World Wide Web* (W3C) para acceder y modificar documentos XML. Una implementación del DOM presenta los documentos XML como un árbol, o permite al código cliente construir dichas estructuras desde cero para luego darles acceso a la estructura a través de un conjunto de objetos que implementaron interfaces conocidas.

El DOM es extremadamente útil para aplicaciones de acceso directo. SAX sólo te permite la vista de una parte del documento a la vez. Si estás mirando un elemento SAX, no tienes acceso a otro. Si estás viendo un nodo de texto, no tienes acceso al elemento contenedor. Cuando desarrollas una aplicación SAX, necesitas registrar la posición de tu programa en el documento en algún lado de tu código. SAX no lo hace por ti. Además, desafortunadamente no podrás mirar hacia adelante (*look ahead*) en el documento XML.

Algunas aplicaciones son imposibles en un modelo orientado a eventos sin acceso a un árbol. Por supuesto que puedes construir algún tipo de árbol por tu cuenta en eventos SAX, pero el DOM te evita escribir ese código. El DOM es una representación de árbol estándar para datos XML.

El Modelo de Objetos del Documento es definido por el W3C en fases, o «niveles» en su terminología. El mapeado de Python de la API está basado en la recomendación del DOM nivel 2.

Las aplicaciones DOM típicamente empiezan al diseccionar (*parse*) el XML en un DOM. Cómo esto funciona no está incluido en el DOM nivel 1, y el nivel 2 provee mejoras limitadas. Existe una clase objeto llamada `DOMImplementation` que da acceso a métodos de creación de `Document`, pero de ninguna forma da acceso a los constructores (*builders*) de `reader/parser/Document` de una forma independiente a la implementación. No hay una forma clara para acceder a estos métodos sin un objeto `Document` existente. En Python, cada implementación del DOM proporcionará una función `getDOMImplementation()`. El DOM de nivel 3 añade una especificación para Cargar (*Load*)/Guardar (*Store*), que define una interfaz al lector (*reader*), pero no está disponible aún en la librería estándar de Python.

Una vez que tengas un objeto del documento del DOM, puedes acceder a las partes de tu documento XML a través de sus propiedades y métodos. Estas propiedades están definidas en la especificación del DOM; esta porción del manual describe la interpretación de la especificación en Python.

La especificación estipulada por el W3C define la *DOM API* para Java, ECMAScript, y OMG IDL. El mapeo de Python definido aquí está basado en gran parte en la versión IDL de la especificación, pero no se requiere el cumplimiento estricto (aunque las implementaciones son libres de soportar el mapeo estricto de IDL). Véase la sección *Conformidad* para una discusión detallada del mapeo de los requisitos.

Ver también:

Document Object Model (DOM) Level 2 Specification La recomendación del W3C con la cual se basa el *DOM API* de Python.

Document Object Model (DOM) Level 1 Specification La recomendación del W3C para el DOM soportada por `xml.dom.minidom`.

Python Language Mapping Specification Este documento especifica el mapeo de OMG IDL a Python.

20.6.1 Contenido del Módulo

El módulo `xml.dom` contiene las siguientes funciones:

`xml.dom.registerDOMImplementation(name, factory)`

Registra la función *factory* con el nombre *name*. La función fábrica (*factory*) debe retornar un objeto que implemente la interfaz `DOMImplementation`. La función fábrica puede retornar el mismo objeto cada vez que se llame, o uno nuevo por cada llamada, según sea apropiado para la implementación específica (e.g. si la implementación soporta algunas personalizaciones).

`xml.dom.getDOMImplementation(name=None, features=())`

Retorna una implementación del DOM apropiada. El *name* es o bien conocido, el nombre del módulo de una implementación DOM, o `None`. Si no es `None` importa el módulo correspondiente y retorna un objeto `DOMImplementation` si la importación tiene éxito. Si no se le pasa un nombre, y el entorno de variable `PYTHON_DOM` ha sido puesto, dicha variable es usada para encontrar la información de la implementación.

Si no se le pasa un nombre, examina las implementaciones disponibles para encontrar uno con el conjunto de características requeridas. Si no se encuentra ninguna implementación, levanta una excepción `ImportError`. La lista de características debe ser una secuencia de pares (*feature, version*) que son pasados al método `hasFeature()` en objetos disponibles de `DOMImplementation`.

Algunas constantes convenientes son proporcionadas:

`xml.dom.EMPTY_NAMESPACE`

El valor usado para indicar que ningún espacio de nombres es asociado con un nodo en el DOM. Se encuentra típicamente con el `namespaceURI` de un nodo, o usado como el parámetro *namespaceURI* para un método específico del *namespace*.

`xml.dom.XML_NAMESPACE`

El espacio de nombres de la URI asociada con el prefijo `xml`, como se define por [Namespaces in XML](#) (section 4).

`xml.dom.XMLNS_NAMESPACE`

El espacio de nombres del URI para declaraciones del espacio de nombres, como se define en [Document Object Model \(DOM\) Level 2 Core Specification](#) (section 1.1.8).

`xml.dom.XHTML_NAMESPACE`

El URI del espacio de nombres del XHTML como se define en [XHTML 1.0: The Extensible HyperText Markup Language](#) (section 3.1.1).

Además, `xml.dom` contiene una clase base `Node` y las clases de excepciones del DOM. La clase `Node` proporcionada por este módulo no implementa ninguno de los métodos o atributos definidos en la especificación DOM; las implementaciones del DOM concretas deben definirlos. La clase `Node` propuesta por este módulo sí proporciona las constantes usadas por el atributo `nodeType` en objetos concretos de `Node`; estas son localizadas dentro de la clase en vez de estar al nivel del módulo para cumplir las especificaciones del DOM.

20.6.2 Objetos en el DOM

La documentación definitiva para el DOM es la especificación del DOM del W3C.

Note que los atributos del DOM también pueden ser manipulados como nodos en vez de simples cadenas de caracteres (*strings*). Sin embargo, es bastante raro que tengas que hacer esto, por lo que su uso aún no está documentado.

Interfaz	Sección	Propósito
<code>DOMImplementation</code>	<i>Objetos <code>DOMImplementation</code></i>	Interfaz para las implementaciones subyacentes.
<code>Node</code>	<i>Objetos <code>Nodo</code></i>	Interfaz base para la mayoría de objetos en un documento.
<code>NodeList</code>	<i>Objetos <code>NodeList</code></i>	Interfaz para una secuencia de nodos.
<code>DocumentType</code>	<i>Objetos <code>DocumentType</code></i>	Información acerca de la declaraciones necesarias para procesar un documento.
<code>Document</code>	<i>Objetos <code>Documento</code></i>	Objeto que representa un documento entero.
<code>Element</code>	<i>Objetos <code>Elemento</code></i>	Nodos elemento en la jerarquía del documento.
<code>Attr</code>	<i>Objetos <code>Atributo</code></i>	Nodos de los valores de los atributos en los elementos nodo.
<code>Comment</code>	<i>Objetos <code>Comentario</code></i>	Representación de los comentarios en el documento fuente.
<code>Text</code>	<i>Objetos <code>Texto</code> y <code>CDATASection</code></i>	Nodos con contenido textual del documento.
<code>ProcessingInstruction</code>	<i>Objetos <code>ProcessingInstruction</code></i>	Representación de instrucción del procesamiento.

Una sección adicional describe las excepciones definidas para trabajar con el *DOM* en Python.

Objetos `DOMImplementation`

La interfaz `DOMImplementation` proporciona una forma para que las aplicaciones determinen la disponibilidad de características particulares en el *DOM* que están usando. El *DOM* nivel 2 añadió la habilidad de crear nuevos objetos `Document` y `DocumentType` usando `DOMImplementation` también.

`DOMImplementation.hasFeature` (*feature*, *version*)

Retorna `True` si la característica identificada por el par de cadenas de caracteres *feature* y *version* está implementada.

`DOMImplementation.createDocument` (*namespaceUri*, *qualifiedName*, *doctype*)

Retorna un nuevo objeto `Document` (la raíz del *DOM*), con un hijo objeto `Element` teniendo el *namespaceUri* y *qualifiedName* dados. El *doctype* debe ser un `DocumentType` creado por `createDocumentType()` o `None`. En la *DOM API* de Python, los primeros argumentos pueden ser `None` para indicar que ningún hijo `Element` va a ser creado.

`DOMImplementation.createDocumentType` (*qualifiedName*, *publicId*, *systemId*)

Retorna un nuevo objeto `DocumentType` que encapsula las cadenas de caracteres *qualifiedName*, *publicId*, y *systemId* dadas, representando la información contenida en un tipo de declaración de documento XML.

Objetos `Nodo`

Todos los componentes de un documento XML son sub-clases de `Node`.

`Node.nodeType`

Un entero representando el tipo de nodo. Las Constantes simbólicas para los tipos están en el objeto `Nodo`: `ELEMENT_NODE`, `ATTRIBUTE_NODE`, `TEXT_NODE`, `CDATA_SECTION_NODE`, `ENTITY_NODE`, `PROCESSING_INSTRUCTION_NODE`, `COMMENT_NODE`, `DOCUMENT_NODE`, `DOCUMENT_TYPE_NODE`, `NOTATION_NODE`. Este es un atributo sólo de lectura.

`Node.parentNode`

El padre del nodo actual, o `None` para el nodo del documento. El valor es siempre un objeto `Node` o `None`. Para los nodos `Element`, este será el elemento padre, excepto para el elemento raíz, en cuyo caso será el objeto `Document`. Para los nodos `Attr`, este siempre es `None`. Este es un atributo de sólo lectura.

`Node.attributes`

Un `NamedNodeMap` de objetos de atributos. Sólo los elementos tienen un valor real para esto; otros nodos proporcionan `None` para este atributo. Este es un atributo de sólo lectura.

`Node.previousSibling`

El nodo que precede inmediatamente este nodo con el mismo padre. Por ejemplo el elemento con una etiqueta

final que viene justo antes de la etiqueta del comienzo del elemento *self*. Por supuesto, los documentos XML está hechos de más que sólo elementos por lo que el hermano anterior puede ser un texto, un comentario, o algo más. Si este nodo es el primer hijo del padre, este atributo será `None`. Este es un atributo de sólo lectura.

Node.`nextSibling`

El nodo que sigue inmediatamente este nodo con el mismo padre. Véase también [previousSibling](#). Si este es el último hijo del padre, este atributo será `None`. Este es un atributo de sólo lectura.

Node.`childNodes`

Una lista de nodos contenidos dentro de este nodo. Este es un atributo de sólo lectura.

Node.`firstChild`

El primer hijo del nodo, si hay alguno, o `None`. Este es un atributo de sólo lectura.

Node.`lastChild`

El último hijo del nodo, si hay alguno, o `None`. Este es un atributo de sólo lectura.

Node.`localName`

La parte del `tagName` seguido de los dos puntos si hay uno, si no, el `tagName` entero. El valor es una cadena de caracteres.

Node.`prefix`

La parte del `tagName` antes de los dos puntos si hay uno, si no, la cadena de caracteres vacía. El valor es una cadena, o `None`.

Node.`namespaceURI`

El espacio de nombres asociado con el nombre del elemento. Este será una cadena de caracteres o `None`. Este es un atributo de sólo lectura.

Node.`nodeName`

Este tiene un significado diferente para cada tipo de nodo; véase la especificación del DOM para los detalles. Siempre puedes obtener la información que obtendrías aquí desde otra propiedad como la propiedad `tagName` para elementos o la propiedad `name` para atributos. Para todos los tipos de nodo, el valor de este atributo tendrá o una cadena de caracteres o `None`. Este es un atributo de sólo lectura.

Node.`nodeValue`

Este tiene un significado diferente para cada tipo de nodo; véase la especificación del DOM para los detalles. La situación es similar a ese con [nodeName](#). El valor es una cadena de caracteres o `None`.

Node.`hasAttributes` ()

Retorna `True` si el nodo tiene algún atributo.

Node.`hasChildNodes` ()

Retorna `True` si el nodo tiene algún nodo hijo.

Node.`isSameNode` (*other*)

Retorna `True` si *other* hace referencia al mismo nodo como este nodo. Esto es especialmente útil para las implementaciones DOM que usan una arquitectura *proxy* de cualquier tipo (porque más de un objeto puede hacer referencia al mismo nodo).

Nota: Esto se basa en una *DOM API* de nivel 3 propuesta que está en la etapa de «borrador de trabajo» (*«working draft»*), pero esta interfaz en particular no parece controversial. Los cambios del *W3C* necesariamente no afectaran este método en la interfaz del *DOM* del Python (aunque cualquier nueva *API* del *W3C* para esto también sería soportado).

Node.`appendChild` (*newChild*)

Añade un nuevo nodo hijo a este nodo al final de la lista de hijos, retornando *newChild*. Si el nodo ya estaba en el árbol, este se remueve primero.

Node.`insertBefore` (*newChild*, *refChild*)

Inserta un nuevo nodo hijo antes de un hijo existente. Debe ser el caso que *refChild* sea un hijo de este nodo; si no, `ValueError` es lanzado. *newChild* es retornado. Si *refChild* es `None`, se inserta a *newChild* al final de la lista de hijos.

`Node.removeChild(oldChild)`

Elimina un nodo hijo. *oldChild* debe ser un hijo de este nodo; si no, `ValueError` es lanzado. *oldChild* es retornado si tiene éxito. Si *oldChild* no será usado más adelante, se debe llamar a su método `unlink()`.

`Node.replaceChild(newChild, oldChild)`

Reemplaza un nodo existente con un nuevo nodo. Debe ser el caso que *oldChild* sea un hijo de este nodo; si no, `ValueError` es lanzado.

`Node.normalize()`

Une nodos de texto adyacentes para que todos los tramos de texto sean guardados como únicas instancias de `Text`. Esto simplifica el procesamiento de texto de un árbol del *DOM* para muchas aplicaciones.

`Node.cloneNode(deep)`

Clona este nodo. Poner *deep* significa clonar todos los nodos hijo también. Esto retorna el clon.

Objetos *NodeList*

Un *NodeList* representa una secuencia de nodos. Estos objetos se usan de dos formas en la recomendación principal del *DOM*: un objeto *Element* proporciona uno como su lista de nodos hijo, y los métodos `getElementsByTagName()` y `getElementsByTagNameNS()` de *Node* retornan objetos con esta interfaz para representar resultados de consulta.

La recomendación del *DOM* nivel 2 define un método y un atributo para estos objetos:

`NodeList.item(i)`

Retorna el *i*-ésimo *item* de la secuencia, si hay uno, o `None`. El índice *i* no puede ser menor que cero o mayor o igual que el tamaño de la secuencia.

`NodeList.length`

El número de nodos en la secuencia.

Además, la interfaz *DOM* de Python requiere que un algún soporte adicional sea proporcionado para que los objetos *NodeList* puedan ser usados como secuencias de Python. Todas las implementaciones de *NodeList* deben incluir soporte para `__len__()` y `__getitem__()`; esto permite la iteración de *NodeList* en sentencias con `for` y un soporte apropiado para la función incorporada `len()`.

Si una implementación *DOM* soporta la modificación del documento, la implementación de *NodeList* debe también soportar los métodos `__setitem__()` y `__delitem__()`.

Objetos *DocumentType*

La información acerca de las notaciones y entidades declaradas por un documento (incluido el subconjunto externo si el analizador lo usa y puede proporcionar información) está disponible desde un objeto *DocumentType*. El *DocumentType* para un documento está disponible desde el atributo `doctype` del objeto *Document*; si no hay ninguna declaración *DOCTYPE* para el documento, el atributo `doctype` del documento se pondrá como `None` en vez de una instancia de esta interfaz.

DocumentType es una especialización de *Node*, y añade los siguientes atributos:

`DocumentType.publicId`

El identificador público para el subconjunto externo de la definición del tipo de documento. Esto será una cadena de caracteres o `None`.

`DocumentType.systemId`

El identificador del sistema para el subconjunto externo de la definición del tipo de documento. Esto será una *URI* como una cadena de caracteres, o `None`.

`DocumentType.internalSubset`

Una cadena de caracteres proporcionando el subconjunto interno completo del documento. Esto no incluye los paréntesis que cierran el subconjunto. Si el documento no tiene ningún subconjunto interno, debe ser `None`.

`DocumentType.name`

El nombre del elemento raíz como se indica en la declaración *DOCTYPE*, si está presente.

DocumentType.entities

Este es un `NamedNodeMap` proporcionando las definiciones de las entidades externas. Para nombres de entidades definidas más de una vez, sólo la primera definición es proporcionada (el resto es ignorado como se requiere por la recomendación de *XML*). Puede ser `None` si el analizador no proporciona la información, o si ninguna entidad es definida.

DocumentType.notations

Este es un `NamedNodeMap` proporcionando las definiciones de las notaciones. Para nombres de notaciones definidas más de una vez, sólo la primera definición es proporcionada (el resto es ignorado como se requiere por la recomendación *XML*). Puede ser `None` si el analizador no proporciona la información, o si no hay notaciones definidas.

Objetos Documento

Un `Documento` representa un documento *XML* entero, incluyendo sus elementos constituyentes, atributos, instrucciones de procesamiento, comentarios, etc. Recuerda que este hereda propiedades de `Node`.

Document.documentElement

El único elemento raíz del documento.

Document.createElement (tagName)

Crea y retorna un nuevo elemento nodo. El elemento no se inserta en el documento cuando es creado. Necesitas insertarlo explícitamente con uno de los otros métodos como `insertBefore()` o `appendChild()`.

Document.createElementNS (namespaceURI, tagName)

Crea y retorna un nuevo elemento con un espacio de nombres. El *tagName* puede tener un prefijo. El elemento no se inserta en el documento cuando es creado. Necesitas insertarlo explícitamente con uno de los otros métodos como `insertBefore()` o `appendChild()`.

Document.createTextNode (data)

Crea y retorna un nodo texto conteniendo los datos pasados como parámetros. Como con los otros métodos de creación, este no inserta el nodo en el árbol.

Document.createComment (data)

Crea y retorna un nodo comentario conteniendo los datos pasados como parámetros. Como con los otros métodos de creación, este no inserta el nodo en el árbol.

Document.createProcessingInstruction (target, data)

Crea y retorna una instrucción de procesamiento conteniendo el *target* y *data* pasados como parámetros. Como con los otros métodos de creación, este no inserta en nodo en el árbol.

Document.createAttribute (name)

Crea y retorna un nodo atributo. Este método no asocia el nodo atributo con ningún elemento particular. Debes usar `setAttributeNode()` en el objeto `Element` apropiado para usar la instancia del atributo recién creada.

Document.createAttributeNS (namespaceURI, qualifiedName)

Crea y retorna un nodo atributo con un espacio de nombres. El *tagName* puede ser un prefijo. Este método no asocia el nodo atributo con ningún elemento en particular. Debes usar `setAttributeNode()` en el objeto `Element` apropiado para usar la instancia del atributo recién creada.

Document.getElementsByTagName (tagName)

Busca todos los descendientes (hijos directos, hijos de los hijos, etc.) con un nombre del tipo de elemento particular.

Document.getElementsByTagNameNS (namespaceURI, localName)

Busca todos los descendientes (hijos directos, hijos de hijos, etc.) con un espacio de nombres URI particular (*namespaceURI*) y nombre local (*localname*). El nombre local es parte del espacio de nombres después del prefijo.

Objetos Elemento

`Element` es una subclase de `Node`, por lo que hereda todos los atributos de esa clase.

`Element.tagName`

El nombre del tipo de elemento. En un documento que usa espacios de nombres este puede tener varios dos puntos en él. El valor es una cadena de caracteres.

`Element.getElementsByTagName(tagName)`

Igual al método equivalente en la clase `Document`.

`Element.getElementsByTagNameNS(namespaceURI, localName)`

Igual al método equivalente en la clase `Document`.

`Element.hasAttribute(name)`

Retorna `True` si el elemento tiene un atributo nombrado `name`.

`Element.hasAttributeNS(namespaceURI, localName)`

Retorna `True` si el elemento tiene un atributo nombrado por `namespaceURI` y `localName`.

`Element.getAttribute(name)`

Retorna el valor del atributo nombrado por `name` como una cadena de caracteres. Si no existe dicho atributo, una cadena vacía es retornada, como si el atributo no tuviera valor.

`Element.getAttributeNode(attrname)`

Retorna el nodo `Attr` para el atributo nombrado por `attrname`.

`Element.getAttributeNS(namespaceURI, localName)`

Retorna el valor del atributo nombrado por `namespaceURI` y `localName` como una cadena de caracteres. Si no existe dicho atributo, una cadena vacía es retornada, como si el atributo no tuviera valor.

`Element.getAttributeNodeNS(namespaceURI, localName)`

Retorna un valor de atributo como nodo, dado un `namespaceURI` y `localName`.

`Element.removeAttribute(name)`

Remueve un atributo por nombre (`name`). Si no hay un atributo correspondiente, un `NotFoundErr` es levantado.

`Element.removeAttributeNode(oldAttr)`

Remueve y retorna `oldAttr` de la lista de atributos, si está presente. Si `oldAttr` no está presente, `NotFoundErr` es levantado.

`Element.removeAttributeNS(namespaceURI, localName)`

Remueve un atributo por nombre (`name`). Note que esto usa un `localName`, no un `qname`. Ninguna excepción es levantada si no existe el atributo correspondiente.

`Element.setAttribute(name, value)`

Pone un valor de atributo como una cadena de caracteres.

`Element.setAttributeNode(newAttr)`

Añade un nuevo atributo nodo al elemento, reemplazando un atributo existente si es necesario si el atributo `name` coincide. Si un reemplazo ocurre, el viejo atributo será retornado. Si `newAttr` ya está en uso, `InuseAttributeErr` será lanzado.

`Element.setAttributeNodeNS(newAttr)`

Añade un nuevo nodo atributo al elemento, reemplazando un atributo existente si es necesario, si el `namespaceURI` y `localName` coinciden. Si un reemplazo ocurre, el viejo atributo será retornado. Si `newAttr` está en uso, `InuseAttributeErr` será levantado.

`Element.setAttributeNS(namespaceURI, qname, value)`

Pone un valor de atributo a partir de una cadena de caracteres, dados un `namespaceURI` y `qname`. Note que un `qname` es el nombre completo del atributo. Esto es diferente al de arriba.

Objetos Atributo

`Attr` hereda de `Node`, por lo que hereda todos sus atributos.

`Attr.name`

El nombre del atributo. En un documento que usa espacio de nombres, puede incluir dos puntos.

`Attr.localName`

La parte del nombre seguido después de los dos puntos si hay uno, si no el nombre entero. Este es un atributo de sólo lectura.

`Attr.prefix`

La parte del nombre que precede los dos puntos si hay uno, si no la cadena de caracteres vacía.

`Attr.value`

El valor textual del atributo. Este es un sinónimo para el atributo `nodeValue`.

Objetos *NamedNodeMap*

`NamedNodeMap` *no* hereda de `Node`.

`NamedNodeMap.length`

La longitud de la lista de atributos.

`NamedNodeMap.item(index)`

Retorna un atributo con un índice particular. El orden en los que obtienes los atributos es arbitrario pero será consistente en la vida de un *DOM*. Cada *item* es un nodo atributo. Obtén su valor con el atributo `value`.

También hay métodos experimentales que dan a esta clase más comportamiento de mapeado. Puedes usarlos o puedes usar la familia de métodos estandarizados `getAttribute*()` en los objetos `Element`.

Objetos Comentario

`Comment` representa un comentario en el documento *XML*. Es una subclase de `Node`, pero no puede tener hijos nodo.

`Comment.data`

El contenido del comentario como una cadena de caracteres. El atributo contiene todos los caracteres entre el `<!--` que empieza y el `-->` que termina, pero no los incluye.

Objetos Texto y *CDATASection*

La interfaz `Text` representa el texto en el documento *XML*. Si el analizador y la implementación del *DOM* soporta la extensión *XML* del *DOM*, las porciones de texto rodeadas secciones marcadas como *CDATA* se guardan en objetos `CDATASection`. Estas dos interfaces son idénticas, pero proveen valores diferentes para el atributo `nodeType`.

Estas interfaces extienden la interfaz `Node`. No pueden tener nodos hijo.

`Text.data`

El contenido del nodo texto como una cadena de caracteres.

Nota: El uso de un nodo `CDATASection` no indica que el nodo represente una sección completa marcada como *CDATA*, sólo que el contenido del nodo fue parte de una sección *CDATA*. Una sola sección *CDATA* puede ser representada por más de un nodo en el árbol del documento. No hay manera de determinar si dos nodos adyacentes `CDATASection` son representados diferentes a secciones marcadas como *CDATA*.

Objetos *ProcessingInstruction*

Representa una instrucción de procesamiento en el documento *XML*; hereda de la interfaz `Node` y no puede tener hijos.

`ProcessingInstruction.target`

El contenido de la instrucción de procesamiento hasta el carácter en blanco. Este es un atributo de sólo lectura.

`ProcessingInstruction.data`

El contenido de la instrucción de procesamiento después del primer carácter en blanco.

Excepciones

La recomendación del *DOM* nivel 2 define una sola excepción, *DOMException*, y un número de constantes que permite que las aplicaciones determinen qué tipo de error ocurrió. las instancias de *DOMException* llevan un atributo *code* que proporciona el valor apropiado para la excepción específica.

La interfaz *DOM* de Python provee las constantes, pero también expande el conjunto de excepciones para que exista una excepción específica para cada uno de los códigos de excepción definidos por el *DOM*. Las implementaciones deben lanzar la excepción específica apropiada, cada uno de los cuales lleva el valor apropiado para el atributo *code*.

exception `xml.dom.DOMException`

Clase base de excepción usada para todas las excepciones del *DOM* específicas. Esta clase de excepción no puede ser instanciada directamente.

exception `xml.dom.DomstringSizeErr`

Lanzado cuando un rango de texto específico no cabe en una cadena de caracteres. No se sabe si se usa in las implementación *DOM* de Python, pero puede ser recibido de otras implementaciones *DOM* que no hayan sido escritas en Python.

exception `xml.dom.HierarchyRequestErr`

Lanzado cuando se intenta insertar un nodo donde el tipo de nodo no es permitido.

exception `xml.dom.IndexSizeErr`

Lanzado cuando un parámetro del índice o tamaño de un método es negativo o excede los valores permitidos.

exception `xml.dom.InuseAttributeErr`

Lanzado cuando se intenta insertar un nodo `Attr` que está presente en algún lado en el documento.

exception `xml.dom.InvalidAccessErr`

Lanzado si un parámetro o una operación no es soportada por el objeto subyacente.

exception `xml.dom.InvalidCharacterErr`

Esta excepción es lanzada cuando un parámetro de cadena de caracteres contiene un carácter que no está permitido en el contexto que está siendo usado por la recomendación *XML 1.0*. Por ejemplo, intentar crear un nodo `Element` con un espacio en el nombre del tipo de elemento causará que se lance este error.

exception `xml.dom.InvalidModificationErr`

Lanzado cuando se intenta modificar el tipo de un nodo.

exception `xml.dom.InvalidStateErr`

Lanzado cuando se intenta usar un objeto que no está definido o ya no es usable.

exception `xml.dom.NamespaceErr`

Si se intenta cambiar cualquier objeto de forma que no sea permitida con respecto a la recomendación *Namespaces in XML*, esta excepción es lanzada.

exception `xml.dom.NotFoundErr`

Excepción cuando un nodo no existe en el contexto referenciado. Por ejemplo, `NamedNodeMap.removeNamedItem()` será lanzado si el nodo pasado no existe en el mapa.

exception `xml.dom.NotSupportedErr`

Lanzado cuando la implementación no soporta el tipo requerido del objeto u operación.

exception `xml.dom.NoDataAllowedErr`

Es lanzado si se especifican datos para un nodo que no soporta datos.

exception `xml.dom.NoModificationAllowedErr`

Lanzado cuando se intenta modificar un objeto donde las modificaciones no son permitidas (tal como los nodos de sólo-lectura).

exception `xml.dom.SyntaxErr`

Lanzado cuando se especifica una cadena de caracteres inválida o ilegal.

exception `xml.dom.WrongDocumentErr`

Lanzado cuando un nodo es insertado en un documento diferente al que este actualmente pertenece, y la implementación no soporta migrar el nodo de un documento a otro.

Los códigos de excepción definidos en la recomendación del *DOM* se mapean a las excepciones descritas arriba de acuerdo a esta tabla:

Constante	Excepción
DOMSTRING_SIZE_ERR	<i>DomstringSizeErr</i>
HIERARCHY_REQUEST_ERR	<i>HierarchyRequestErr</i>
INDEX_SIZE_ERR	<i>IndexSizeErr</i>
INUSE_ATTRIBUTE_ERR	<i>InuseAttributeErr</i>
INVALID_ACCESS_ERR	<i>InvalidAccessErr</i>
INVALID_CHARACTER_ERR	<i>InvalidCharacterErr</i>
INVALID_MODIFICATION_ERR	<i>InvalidModificationErr</i>
INVALID_STATE_ERR	<i>InvalidStateErr</i>
NAMESPACE_ERR	<i>NamespaceErr</i>
NOT_FOUND_ERR	<i>NotFoundErr</i>
NOT_SUPPORTED_ERR	<i>NotSupportedErr</i>
NO_DATA_ALLOWED_ERR	<i>NoDataAllowedErr</i>
NO_MODIFICATION_ALLOWED_ERR	<i>NoModificationAllowedErr</i>
SYNTAX_ERR	<i>SyntaxErr</i>
WRONG_DOCUMENT_ERR	<i>WrongDocumentErr</i>

20.6.3 Conformidad

Esta sección describe los requisitos de conformidad y las relaciones entre el *DOM API* de Python, las recomendaciones del *DOM* del *W3C*, y el mapeo *OMG IDL* para Python.

Mapeo de tipos

Los tipos IDL usados en la especificación del *DOM* son mapeados a los tipos de tipos de Python de acuerdo a la siguiente tabla.

Tipo IDL	Tipo en Python
boolean	<code>bool</code> o <code>int</code>
int	<code>int</code>
long int	<code>int</code>
unsigned int	<code>int</code>
DOMString	<code>str</code> o <code>bytes</code>
null	<code>None</code>

Métodos de acceso (*accessor*)

El mapeo de *OMG IDL* a python define funciones de acceso para las declaraciones del atributo *IDL* de la que misma forma en que el mapeo de Java lo hace. Mapear las declaraciones IDL:

```
readonly attribute string someValue;
    attribute string anotherValue;
```

produce tres funciones de acceso: un método «get» para `someValue` (`_get_someValue()`), y métodos «get» y «set» para `anotherValue` (`_get_anotherValue()` y `_set_anotherValue()`). El mapeado, en particular, no requiere que los atributos *IDL* sean accesibles como los atributos normales de Python: No es obligatorio que `object.someValue` funcione, y puede lanzar un `AttributeError`.

El *DOM API* de Python, sin embargo, *si* requiere que los atributos de acceso normales funcionen. Esto significa que no es probable que los típicos sustitutos generados por compiladores de *IDL* en Python funcionen, y los objetos envoltorio (*wrapper*) pueden ser necesarios en el cliente si los objetos del *DOM* son accedidos mediante *CORBA*. Mientras que esto requiere consideraciones adicionales para clientes *DOM* en *CORBA*, los implementadores con experiencia que usen *DOM* por encima de *CORBA* desde Python no lo consideran un problema. Los atributos que se declaran `readonly` pueden no restringir el acceso de escritura en todas las implementaciones *DOM*.

En el *DOM API* de Python, las funciones de acceso no son obligatorias. Si se proveen, deben tomar la forma definida por el mapeo *IDL* de Python, pero estos métodos se consideran innecesarios debido a que los atributos son accesibles directamente desde Python. Nunca se deben proporcionar métodos de acceso (*accessor*) «Set» para los atributos `readonly`.

Las definiciones de IDL no encarnan los requisitos del *DOM API* del W3C por completo, como las nociones de ciertos objetos, como el valor de retorno `getElementsByTagName()`, siendo «live». El *DOM API* de Python no requiere que las implementaciones hagan cumplir tales requisitos.

20.7 `xml.dom.minidom` — Implementación mínima del DOM

Código fuente: [Lib/xml/dom/minidom.py](#)

`xml.dom.minidom` es una implementación mínima de la interfaz Document Object Model (Modelo de objetos del documento), con una API similar a la de otros lenguajes. Está destinada a ser más simple que una implementación completa del DOM y también significativamente más pequeña. Aquellos usuarios que aún no dominen el DOM deberían considerar usar el módulo `xml.etree.ElementTree` en su lugar para su procesamiento XML.

Advertencia: El módulo `xml.dom.minidom` no es seguro contra datos construidos maliciosamente. Si necesitas analizar datos que no son de confianza o no autenticados, consulta [Vulnerabilidades XML](#).

Las aplicaciones DOM suelen comenzar analizando algún XML en un DOM. Con `xml.dom.minidom`, esto se hace a través de las funciones de análisis sintáctico:

```
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>')
```

La función `parse()` puede tomar un nombre de archivo o un objeto de archivo previamente abierto.

`xml.dom.minidom.parse(filename_or_file, parser=None, bufsize=None)`

Retorna un `Document` a partir de la entrada dada. `filename_or_file` puede ser un nombre de archivo o un

objeto similar a un archivo. *parser*, si se proporciona, debe ser un objeto de un analizador sintáctico SAX2. Esta función intercambiará el controlador de documentos del analizador sintáctico y activará el soporte con el espacio de nombres. Otras configuraciones del analizador sintáctico (como configurar un solucionador de entidades) deben haberse realizado de antemano.

Si tienes XML en una cadena de caracteres, puedes usar la función `parseString()` en su lugar:

```
xml.dom.minidom.parseString(string, parser=None)
```

Retorna un objeto `Document` que representa a *string*. Este método crea un objeto `io.StringIO` para la cadena de caracteres y lo pasa a `parse()`.

Ambas funciones retornan un objeto `Document` que representa el contenido del documento.

Lo que hacen las funciones `parse()` y `parseString()` es conectar un analizador sintáctico de XML con un «constructor DOM» que puede aceptar eventos de análisis de cualquier analizador sintáctico SAX y convertirlos en un árbol DOM. El nombre de las funciones es quizás engañoso, pero es fácil de entender cuando se comprenden las interfaces. El análisis sintáctico del documento se completará antes de que retornen estas funciones, dichas funciones simplemente no proporcionan una implementación del analizador sintáctico por sí mismas.

También puedes crear un objeto `Document` invocando a un método en un objeto de la «Implementación del DOM». Puedes obtener este objeto llamando a la función `getDOMImplementation()` del paquete `xml.dom` o del módulo `xml.dom.minidom`. Una vez que tengas un objeto `Document`, puedes agregarle nodos secundarios para llenar el 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)
```

Una vez que tengas un objeto del documento DOM, puedes acceder a las partes de tu documento XML a través de sus propiedades y métodos. Estas propiedades se definen en la especificación DOM. La propiedad principal del objeto del documento es `documentElement`. Te proporciona el elemento principal en el documento XML: el que contiene a todos los demás. Aquí hay un programa de ejemplo:

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

Cuando hayas terminado con un árbol DOM, puedes invocar opcionalmente el método `unlink()` para forzar la limpieza temprana de los objetos ahora innecesarios. `unlink()` es una extensión de la API del DOM específica del módulo `xml.dom.minidom`, que hace que el nodo y sus descendientes sean esencialmente inútiles. En caso de no hacer uso de esto, eventualmente el recolector de basura de Python se hará cargo de los objetos en el árbol.

Ver también:

Document Object Model (DOM) Level 1 Specification La recomendación del W3C para el DOM soportada por el módulo `xml.dom.minidom`.

20.7.1 Objetos del DOM

La definición de la API del DOM para Python se proporciona como parte de la documentación del módulo `xml.dom`. Esta sección simplemente enumera las diferencias entre esta API y el módulo `xml.dom.minidom`.

`Node.unlink()`

Rompe las referencias internas dentro del DOM para recolectarlo como basura en las versiones de Python sin recolector de basura cíclico. Incluso cuando se dispone del mismo, su uso puede hacer que grandes cantidades de memoria estén disponibles antes, por lo que es una buena práctica invocar este método en objetos DOM, tan pronto como ya no se necesiten. Solo necesita ser invocado en el objeto `Document`, pero se puede llamar en los nodos hijos para descartar los hijos de ese nodo concreto.

Puedes evitar invocar este método explícitamente utilizando la declaración `with`. El siguiente código desvinculará automáticamente `dom` cuando se salga del bloque `with`:

```
with xml.dom.minidom.parse(datasource) as dom:
    ... # Work with dom.
```

Node.**writexml** (*writer*, *indent=""*, *addindent=""*, *newl=""*)

Escribe XML en el objeto escritor. El escritor recibe texto pero no bytes como entrada, debe tener un método `write()` que coincida con el de la interfaz del objeto de archivo. El parámetro *indent* es la sangría del nodo actual. El parámetro *addindent* es la sangría incremental que se utilizará para los subnodos del nodo actual. El parámetro *newl* especifica la cadena que se utilizará como terminación de las nuevas líneas.

Para el nodo `Document`, se puede usar el argumento por palabra clave adicional *encoding* para especificar el valor del campo de codificación del encabezado XML.

Distinto en la versión 3.8: El método `writexml()` ahora conserva el orden de los atributos especificado por el usuario.

Node.**toxml** (*encoding=None*)

Retorna una cadena de caracteres o una cadena de bytes que contiene el XML representado por el nodo DOM.

Si se proporciona de forma explícita un valor para el argumento *encoding*¹, el resultado es una cadena de bytes con la codificación especificada. Si no se proporciona el argumento *encoding*, el resultado es una cadena Unicode y la declaración XML en la cadena resultante no especifica una codificación. Codificar esta cadena en una codificación que no sea UTF-8 probablemente sea una práctica incorrecta, ya que UTF-8 es la codificación predeterminada para XML.

Distinto en la versión 3.8: El método `toxml()` ahora conserva el orden de los atributos especificado por el usuario.

Node.**toprettyxml** (*indent="\t"*, *newl="\n"*, *encoding=None*)

Retorna una versión impresa elegante del documento. *indent* especifica la cadena de caracteres a usar como sangría y es una tabulación por defecto; *newl* especifica la cadena de caracteres emitida al final de cada línea y es `\n` por defecto.

El argumento *encoding* se comporta como el argumento correspondiente del método `toxml()`.

Distinto en la versión 3.8: El método `toprettyxml()` ahora conserva el orden de los atributos especificado por el usuario.

20.7.2 Ejemplo de DOM

Este programa de ejemplo es una demostración bastante realista de un programa simple. En este caso particular, no aprovechamos mucho la flexibilidad del 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>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
```

(continué en la próxima página)

¹ El nombre de codificación incluido en la salida XML debe cumplir con los estándares apropiados. Por ejemplo, «UTF-8» es válido, pero «UTF8» no es válido en la declaración de un documento XML, aunque Python lo acepta como nombre de codificación. Para más detalles, consulta <https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl> y <https://www.iana.org/assignments/character-sets/character-sets.xhtml>.

(proviene de la página anterior)

```
<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)
```

20.7.3 minidom y el estándar DOM

El módulo `xml.dom.minidom` es esencialmente un DOM compatible con DOM 1.0, con algunas características de DOM 2 (principalmente características del espacio de nombres).

El uso de la interfaz DOM en Python es sencillo. Se aplican las siguientes reglas de mapeo:

- Se accede a las interfaces a través de objetos de instancia. Las aplicaciones no deben instanciar las clases en sí mismas; deben usar las funciones de creación disponibles en el objeto `Document`. Las interfaces derivadas admiten todas las operaciones (y atributos) de las interfaces base, además de cualquier operación nueva.
- Las operaciones se utilizan como métodos. Dado que el DOM usa solo parámetros `in`, los argumentos se pasan en el orden normal (de izquierda a derecha). No hay argumentos opcionales. Las operaciones `void` retornan `None`.
- Los atributos IDL se asignan a atributos de instancia. Por compatibilidad con el mapeo del lenguaje OMG IDL para Python, también se puede acceder a un atributo `foo` a través de los métodos de acceso `_get_foo()` y `_set_foo()`. Los atributos `readonly` no deben modificarse; esto no se aplica en tiempo de ejecución.
- Los tipos `short int`, `unsigned int`, `unsigned long long` y `boolean` se asignan todos a objetos enteros de Python.
- El tipo `DOMString` se asigna a cadenas de caracteres de Python. El módulo `xml.dom.minidom` admite bytes o cadenas de caracteres, pero normalmente producirá cadenas de caracteres. Los valores de tipo `DOMString` también pueden ser `None` cuando la especificación DOM del W3C permite tener el valor IDL `null`.
- Las declaraciones `const` se asignan a variables en su ámbito respectivo (por ejemplo, `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`); no deben modificarse.
- `DOMException` no está actualmente soportado por el módulo `xml.dom.minidom`. En su lugar, `xml.dom.minidom` usa excepciones estándar de Python como `TypeError` y `AttributeError`.
- Los objetos de la clase `NodeList` se implementan usando el tipo lista incorporado de Python. Estos objetos proporcionan la interfaz definida en la especificación DOM, pero en versiones anteriores de Python no son compatibles con la API oficial. Sin embargo, son mucho más «pythónicas» que la interfaz definida en las recomendaciones del W3C.

Las siguientes interfaces no están implementadas en el módulo `xml.dom.minidom`:

- `DOMTimeStamp`
- `EntityReference`

La mayoría de ellas reflejan información en el documento XML que generalmente no es de utilidad para la mayoría de los usuarios de DOM.

Notas al pie

20.8 xml.dom.pulldom — Soporte para la construcción parcial de árboles DOM

Source code: `Lib/xml/dom/pulldom.py`

El módulo `xml.dom.pulldom` proporciona un «pull parser» al que también se le puede pedir que produzca DOM-fragmentos del documento accesibles cuando sea necesario. El concepto básico implica extraer «eventos» desde una secuencia (*stream*) de entrada XML y procesarlos. A diferencia de SAX, que también emplea un modelo de procesamiento orientado a eventos junto con callbacks (retrollamada), el usuario de un analizador de extracción (*pull parser*) es responsable de extraer explícitamente eventos de la secuencia, recorriendo esos eventos hasta que finalice el procesamiento o se produzca una condición de error.

Advertencia: El módulo `xml.dom.pulldom` no es seguro contra datos maliciosamente contruidos . Si necesita analizar datos que no son confiables o no autenticados, consulte [Vulnerabilidades XML](#).

Distinto en la versión 3.7.1: El analizador SAX ya no procesa entidades externas generales de forma predeterminada para aumentar la seguridad de forma predeterminada. Para habilitar el procesamiento de entidades externas, pase una instancia de analizador personalizada (*custom parser instance in:*)

```
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges

parser = make_parser()
parser.setFeature(feature_external_ges, True)
parse(filename, parser=parser)
```

Ejemplo:

```
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'item':
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

event es una constante y puede ser uno de:

- `START_ELEMENT` (Iniciar elemento)
- `END_ELEMENT` (Finalizar elemento)
- `COMMENT` (comentario)
- `START_DOCUMENT` (Iniciar documento)
- `END_DOCUMENT` (finalizar documento)
- `CHARACTERS` (caracteres)
- `PROCESSING_INSTRUCTION` (instrucción de procesamiento)
- `IGNORABLE_WHITESPACE` (Espacio en blanco que puede ignorarse)

node es un objeto del tipo `xml.dom.minidom.Document`, `xml.dom.minidom.Element` ó `xml.dom.minidom.Text`.

Puesto que el documento se trata como una secuencia «flat» (plana) de eventos, el documento «tree» (árbol) se atraviesa implícitamente y los elementos deseados se encuentran independientemente de su profundidad en el árbol. En otras palabras, no es necesario tener en cuenta cuestiones jerárquicas como la búsqueda recursiva de los nodos de documento, aunque si el contexto de los elementos fuera importante, es necesario mantener algún estado relacionado con el contexto (es decir, recordar dónde se encuentra en el documento en un momento dado) o hacer uso del método `DOMEventStream.expandNode()` y cambiar al procesamiento relacionado con DOM.

class `xml.dom.pulldom.PullDom` (*documentFactory=None*)
Subclase de `xml.sax.handler.ContentHandler`.

class `xml.dom.pulldom.SAX2DOM` (*documentFactory=None*)
Subclase de `xml.sax.handler.ContentHandler`.

`xml.dom.pulldom.parse` (*stream_or_string, parser=None, bufsize=None*)

Retorna un `DOMEventStream` de la entrada dada. *stream_or_string* (secuencia o cadena) puede ser un nombre de archivo o un objeto similar a un archivo, *parser*, si se indica, debe ser un objeto `XMLReader`. Esta función cambiará el controlador de documentos del analizador y activará el soporte de espacios de nombres;

otra configuración del analizador (como establecer un solucionador de entidades) debe haberse realizado de antemano.

Si tiene XML en una cadena, puede usar en su lugar la función `parseString()`:

```
xml.dom.pulldom.parseString(string, parser=None)
```

Retorna una: clase `DOMEventStream` que representa la cadena (Unicode) *string* (cadena)

```
xml.dom.pulldom.default_bufsize
```

Valor predeterminado para el parámetro *bufsize* para `parse()`.

El valor de las variables puede ser cambiado antes de llamar a `parse()` y el nuevo valor tendrá efecto.

20.8.1 Objetos DOMEventStream

```
class xml.dom.pulldom.DOMEventStream(stream, parser, bufsize)
```

Obsoleto desde la versión 3.8: El soporte para sequence protocol está obsoleto.

```
getEvent()
```

Retorna el contenido de la tupla *event* y del *node* corriente como `xml.dom.minidom.Document` si el evento es igual a `START_DOCUMENT`, `xml.dom.minidom.Element` si el evento es igual a `START_ELEMENT` o `END_ELEMENT` o `xml.dom.minidom.Text` si el evento es igual a `CHARACTERS`. El nodo actual no contiene información sobre sus hijos a menos que se llame a la función `expandNode()`.

```
expandNode(node)
```

Expande todos los hijos de *node* en *node* (nodo en nodo). Ejemplo:

```
from xml.dom import pulldom

xml = '<html><title>Foo</title> <p>Some text <div>and more</div></p> </html>'
doc = pulldom.parseString(xml)
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'p':
        # Following statement only prints '<p/>'
        print(node.toxml())
        doc.expandNode(node)
        # Following statement prints node with all its children '<p>Some_
    text <div>and more</div></p>'
    print(node.toxml())
```

```
reset()
```

20.9 XML.sax— Soporte para analizadores SAX2

Código fuente: `Lib/xml/sax/_init__.py`

El paquete `xml.sax` provee un número de módulos que implementan la API Simple para la interfaz XML (SAX) para Python. El paquete mismo provee las excepciones SAX y las funciones de conveniencia que serán las más usadas por los usuarios de la API SAX.

Advertencia: El módulo `XML.sax` no es seguro contra datos contruidos maliciosamente. Si necesita analizar datos no autenticados o no confiables, mirar [Vulnerabilidades XML](#).

Distinto en la versión 3.7.1: El analizador SAX ya no procesa entidades generales externas por defecto para incrementar seguridad. Antes, el analizador creaba conexiones de red para buscar archivos remotos o archivos locales

cargados del sistema de archivos para DTD y entidades. La característica puede ser activadas de nuevo con el método `setFeature()` en el objeto analizador y el argumento `feature_external_ges`.

Las funciones de conveniencia son:

xml.sax.make_parser (*parser_list=[]*)

Crea y retorna un objeto SAX `XMLReader`. El primer analizador encontrado será el que se use. Si se provee *parser_list*, debe ser un iterable de cadenas de caracteres el cual nombra módulos que tienen una función llamada `create_parser()`. Los módulos listados en *parser_list* serán usados antes de los módulos en la lista de analizadores por defecto.

Distinto en la versión 3.8: El argumento *parser_list* puede ser cualquier iterable, no sólo una lista.

xml.sax.parse (*filename_or_stream, handler, error_handler=handler.ErrorHandler()*)

Crea un analizador SAX y úsalo para analizar un documento. El documento, aprobado como *filename_or_stream*, puede ser un nombre de archivo o un objeto de archivo. El parámetro *handler* necesita ser una instancia SAX `ContentHandler`. Si se da *error_handler*, debe ser una instancia `ErrorHandler` SAX; si es omitido, se lanzará `SAXParseException` en todos los errores. No hay valor retornado; toda tarea debe ser realizada por el *handler* aprobado.

xml.sax.parseString (*string, handler, error_handler=handler.ErrorHandler()*)

Similar a `parser()`, pero analiza desde un búfer *string* recibido como un parámetro. *string* debe ser una instancia `str` o un *bytes-like object*.

Distinto en la versión 3.5: Agregado soporte de instancias `str`.

Una aplicación SAX típica usa tres tipos de objetos: lectores, gestores y fuentes de entrada. «Lector» en este contexto es otro término para analizador, por ejemplo, alguna pieza de código que lee los bytes o caracteres de la fuente de entrada, y produce una secuencia de eventos. Los eventos luego se distribuyen a los objetos gestores, por ejemplo el lector invoca un método en el gestor. Una aplicación SAX debe por tanto obtener un objeto lector, crear o abrir una fuente de entrada, crear los gestores, y conectar esos objetos juntos. Como paso final de preparación, el lector es llamado para analizar la entrada. Durante el análisis, los métodos en los objetos gestores son llamados basados en eventos estructurales y sintácticos de los datos introducidos.

Para estos objetos, sólo las interfaces son relevantes; éstos normalmente no son instanciados por la aplicación misma. Ya que Python no tiene una noción explícita de interfaz, éstas son introducidas formalmente como clases, pero las aplicaciones suelen usar implementaciones que no heredan de las clases provistas. Las interfaces `InputSource`, `Locator`, `Attributes`, `AttributesNS`, y `XMLReader` son definidas en el módulo `xml.sax.xmlreader`. Las interfaces de gestión son definidas en `xml.sax.handler`. Por conveniencia, `InputSource` (el cual suele ser instanciado directamente) y el gestor de clases están también disponibles desde `xml.sax`. Estas interfaces son descritas a continuación.

En adición a esas clases, `xml.sax` provee las siguientes clases de excepción.

exception xml.sax.SAXException (*msg, exception=None*)

Encapsula un error XML o advertencia. Esta clase puede contener errores básicos o información de advertencias ya sea para el analizador XML o la aplicación: esto puede ser heredado para proveer funcionalidad adicional o para agregar localización. Nota que a pesar de los analizadores definidos en la interfaz `ErrorHandler` recibe instancias de esta excepción, no es requerido para lanzar la excepción — esto es algo útil como un contenedor para información.

Cuando es instanciado, *msg* debería ser una descripción del error legible para humanos. El parámetro opcional *exception*, si es dado, debería ser `None` o una excepción que fue atrapada por el código analizador y se transmite como información.

Esta es la clase base para las otras clases excepción SAX.

exception xml.sax.SAXParseException (*msg, exception, locator*)

Subclase de `SAXException` levantada en errores de análisis. Las instancias de esta clase son pasadas a los métodos de las interfaces SAX `ErrorHandler` para proveer información sobre el error de análisis. Esta clase soporta la interfaz SAX `Locator` así como la interfaz `SAXException`.

exception xml.sax.SAXNotRecognizedException (*msg, exception=None*)

Subclase de `SAXException` lanzada cuando una SAX `XMLReader` es confrontada con una propiedad

o característica no reconocida. Las aplicaciones SAX y extensiones pueden usar esta clase para propósitos similares.

exception `xml.sax.SAXNotSupportedException` (*msg, exception=None*)

Las subclases de `SAXException` se lanzan cuando un SAX *sax* se pregunta para habilitar una característica que no tiene soporte, o para establecer una propiedad a un valor que la implementación no da soporte. Las aplicaciones SAX y las extensiones pueden usar esta clase para propósitos similares.

Ver también:

SAX: The Simple API for XML Este sitio es el punto focal para la definición de la API SAX. Provee una implementación Java y documentación en línea. Los enlaces para implementaciones e información histórica también están disponibles.

Módulo `xml.sax.handler` Definiciones de las interfaces para objetos proporcionados por aplicaciones.

Módulo `xml.sax.saxutils` Funciones de conveniencia para usar en aplicaciones SAX.

Módulo `xml.sax.xmlreader` Definiciones de las interfaces para objetos que proveen analizadores.

20.9.1 Objetos SAXException

La clase de excepción `SAXException` da soporte a los siguientes métodos:

`SAXException.getMessage()`

Retorna un mensaje legible para humanos describiendo la condición de error.

`SAXException.getException()`

Retorna un objeto excepción encapsulado, o `None`.

20.10 `xml.sax.handler` — Base classes for SAX handlers

Source code: `Lib/xml/sax/handler.py`

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.

20.10.1 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 [La Interfaz Attributes](#)) 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 [La Interfaz AttributesNS](#)) 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 string or bytes instance; the `expat` reader module always produces strings.

Nota: 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.

20.10.2 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.

20.10.3 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*.

20.10.4 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.

20.11 `xml.sax.saxutils` — Utilidades SAX

Código fuente: [Lib/xml/sax/saxutils.py](#)

El módulo `xml.sax.saxutils` contiene una serie de clases y funciones que son comúnmente útiles al crear aplicaciones SAX, ya sea como uso directo o como clases base.

`xml.sax.saxutils.escape(data, entities={})`
Escapar '&', '<' y '>' en una cadena de datos.

Puede escapar otras cadenas de datos pasando un diccionario como el parámetro opcional *entities*. Las claves y los valores deben ser cadenas; cada clave será reemplazada por su valor correspondiente. Los caracteres '&', '<' y '>' siempre se escapan, incluso si se proporciona *entities*.

`xml.sax.saxutils.unescape(data, entities={})`
Quitar el escape '&', '<', y '>' en una cadena de datos.

Puede quitar el escape de otras cadenas de datos pasando un diccionario como el parámetro opcional *entities*. Las claves y los valores deben ser cadenas; cada clave será reemplazada por su valor correspondiente. A '&', '<' y '>' se les quita siempre el escape, incluso si se proporcionan *entidades*.

`xml.sax.saxutils.quoteattr(data, entities={})`
Similar a `escape()`, pero también prepara *data* para usarse como un valor de atributo. El valor devuelto es una versión entrecomillada de *data* con los reemplazos adicionales necesarios. `quoteattr()` seleccionará un carácter de comillas basado en el contenido de *data*, intentando evitar codificar los caracteres de comillas en la cadena. Si los caracteres de comillas simples y dobles ya están en *data*, los caracteres de comillas dobles se codificarán y *data* se envolverá entre comillas dobles. La cadena resultante se puede utilizar directamente como un valor de atributo:

```
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

Esta función es útil al generar valores de atributo para HTML o cualquier SGML utilizando la sintaxis concreta de referencia.

class `xml.sax.saxutils.XMLGenerator` (*out=None*, *encoding='iso-8859-1'*,
short_empty_elements=False)

Esta clase implementa la interfaz `ContentHandler` escribiendo eventos SAX en un documento XML. En otras palabras, el uso de un `XMLGenerator` como controlador de contenido reproducirá el documento original que se está analizando. *out* debe ser un objeto similar a un archivo que por defecto sea `sys.stdout`. *encoding* es la codificación de la secuencia de salida que tiene como valor predeterminado `'iso-8859-1'`. *short_empty_elements* controla el formato de los elementos que no contienen contenido: si `False` (valor predeterminado) se emiten como un par de etiquetas de inicio/fin, si se establece en `True` se emiten como una sola etiqueta autocerrada.

Nuevo en la versión 3.2: El parámetro *short_empty_elements*.

class `xml.sax.saxutils.XMLFilterBase` (*base*)

Esta clase está diseñada para situarse entre un `XMLReader` y los manejadores de eventos de la aplicación cliente. Por defecto, no hace más que pasar las peticiones al lector y los eventos a los manejadores sin modificar, pero las subclases pueden sobrescribir métodos específicos para modificar el flujo de eventos o las peticiones de configuración a medida que pasan.

`xml.sax.saxutils.prepare_input_source(source, base="")`

Esta función toma una fuente de entrada y una URL base opcional y devuelve un objeto `InputSource` totalmente resuelto y listo para ser leído. La fuente de entrada puede ser dada como una cadena, un objeto tipo archivo, o un objeto `InputSource`; los analizadores usarán esta función para implementar el argumento polimórfico *fuentes* a su método `parse()`.

20.12 `xml.sax.xmlreader` — Interfaz para analizadores XML

Código fuente: [Lib/xml/sax/xmlreader.py](#)

Los analizadores SAX implementan la interfaz `XMLReader`. Están implementados en un módulo Python, que debe proveer una función `create_parser()`. Esta función es invocada por `xml.sax.make_parser()` sin argumentos para crear un nuevo objeto analizador.

class `xml.sax.xmlreader.XMLReader`

Clase base que puede ser heredada por analizadores SAX.

class `xml.sax.xmlreader.IncrementalParser`

En algunos casos, es deseable no analizar una fuente de entrada a la vez, si no alimentar partes del documento a medida que estén disponibles. Tenga en cuenta que el lector normalmente no leerá el fichero completo, si no que también lo leerá por partes, aún así `parse()` no retornará hasta que el documento por completo es procesado. Por lo tanto, estas interfaces deben utilizarse si el comportamiento de bloqueo `parse()` no es deseable.

Cuando se crea una instancia del analizador, está listo para comenzar a aceptar información desde el método de alimentación inmediatamente. Después de que el análisis ha finalizado con una llamada para cerrar, se debe llamar al método de reinicio para que el analizador esté listo para aceptar información nueva, ya sea de la fuente o utilizando el método de análisis.

Tenga en cuenta que estos métodos *no* deben ser llamados durante el análisis, es decir, después de que el análisis ha sido llamado y antes de que regrese.

Por defecto, la clase también implementa el método de análisis de la interfaz `XMLReader` utilizando los métodos de alimentación, cierre y reinicio de la interfaz `IncrementalParser` en conveniencia a los escritores de controlador SAX 2.0.

class `xml.sax.xmlreader.Locator`

La interfaz para asociar un evento SAX con una ubicación del documento. Un objeto localizador retornará resultados válidos sólo durante llamadas a métodos `DocumentHandler`; en cualquier otro momento, los resultados son impredecibles. Si la información no está disponible, los métodos pueden retornar `None`.

class `xml.sax.xmlreader.InputSource` (*system_id=None*)

La encapsulación de la información necesaria por el `XMLReader` para leer entidades.

Esta clase puede incluir información sobre el identificador público, identificador del sistema, flujo de bytes (posiblemente con la información de codificación de caracteres) y/o el flujo de caracteres de una entidad.

Las aplicaciones crearán objetos de esta clase para uso en el método `XMLReader.parse()` y para retornar desde `EntityResolver.resolveEntity`.

Una `InputSource` pertenece a la aplicación, el `XMLReader` no tiene permitido modificar objetos `InputSource` pasados desde la aplicación, a pesar de que puede hacer copias y modificarlas.

class `xml.sax.xmlreader.AttributesImpl` (*attrs*)

Esta es una implementación de la interfaz `Attributes` (vea la sección [La Interfaz Attributes](#)). Este es un objeto de tipo diccionario que representa los atributos de elemento en una llamada `startElement()`. En adición a las operaciones de diccionario más útiles, soporta una serie de otros métodos como se describe en la interfaz. Los lectores deben crear una instancia de los objetos de esta clase; *attrs* debe ser un objeto de tipo diccionario que contenga un mapeo de nombres de atributo a valores de atributo.

class `xml.sax.xmlreader.AttributesNSImpl` (*attrs, qnames*)

Variante consciente del espacio de nombres de `AttributesImpl`, que se pasará a `startElementNS()`. Es derivada de `AttributesImpl`, pero entiende nombres de atributo como dos tuplas de `namespaceURI` y `localname`. En adición, provee una serie de métodos esperando nombres calificados como aparecen en el documento original. Esta clase implementa la interfaz `AttributesNS` (vea la sección [La Interfaz AttributesNS](#)).

20.12.1 Objetos XMLReader

La interfaz *XMLReader* soporta los siguientes métodos:

XMLReader.parse (*source*)

Procesa una fuente de entrada, produciendo eventos SAX. El objeto *source* puede ser un identificador de sistema (una cadena identificando la fuente de entrada – típicamente un nombre de fichero o una URL), un *pathlib.Path* o un objeto *path-like*, o un objeto *InputSource*. Cuando *parse()* retorna, la entrada es procesada completamente, y el objeto analizador puede ser descartado o reiniciado.

Distinto en la versión 3.5: Agregado soporte de flujo de caracteres.

Distinto en la versión 3.8: Agregado soporte de objetos *path-like*.

XMLReader.getContentHandler ()

Retorna el *ContentHandler* actual.

XMLReader.setContentHandler (*handler*)

Establece el *ContentHandler* actual. Si ningún *ContentHandler* es establecido, los eventos de contenido serán descartados.

XMLReader.getDTDHandler ()

Retorna el *DTDHandler* actual.

XMLReader.setDTDHandler (*handler*)

Establece el *DTDHandler* actual. Si ningún *DTDHandler* es establecido, los eventos DTD serán descartados.

XMLReader.getEntityResolver ()

Retorna el *EntityResolver* actual.

XMLReader.setEntityResolver (*handler*)

Establece el *EntityResolver* actual. Si ningún *EntityResolver* es establecido, los intentos de resolver una entidad externa resultarán en la apertura del identificador de sistema para la entidad, y un error si no está disponible.

XMLReader.getErrorHandler ()

Retorna el *ErrorHandler* actual.

XMLReader.setErrorHandler (*handler*)

Establece el manejador de errores actual. Si ningún *ErrorHandler* es establecido, se lanzarán errores como excepciones, y se imprimirán alertas.

XMLReader.setLocale (*locale*)

Permite a una aplicación establecer la configuración local para errores y alertas.

Analizadores SAX no son requeridos para proveer localización para errores y alertas; si no pueden soportar la configuración local solicitada, de cualquier forma, lanzarán una excepción SAX. Las aplicaciones pueden solicitar un cambio local en medio del análisis.

XMLReader.getFeature (*featurename*)

Retorna la configuración actual para la característica *featurename*. Si la característica no es reconocida, *SAXNotRecognizedException* es lanzada. Los bien conocidos *featurenames* son listados en el módulo *xml.sax.handler*.

XMLReader.setFeature (*featurename*, *value*)

Establece el *featurename* a *value*. Si la característica no es reconocida, *SAXNotRecognizedException* es lanzada. Si la característica o su configuración no es soportada por el analizador, *SAXNotSupportedException* es lanzada.

XMLReader.getProperty (*propertyname*)

Retorna la configuración actual para la propiedad *propertyname*. Si la configuración no es reconocida, una *SAXNotRecognizedException* es lanzada. Las bien conocidas *propertynames* son listadas en el módulo *xml.sax.handler*.

`XMLReader.setProperty(propertyname, value)`

Establece el *propertyname* a *value*. Si la propiedad no es reconocida, `SAXNotRecognizedException` es lanzada. Si la propiedad o su configuración no es soportada por el analizador, `SAXNotSupportedException` es lanzada.

20.12.2 Objetos `IncrementalParser`

Las instancias de `IncrementalParser` ofrecen los siguientes métodos adicionales:

`IncrementalParser.feed(data)`

Procesa una parte de *data*.

`IncrementalParser.close()`

Asume el fin del documento. Eso verificará las condiciones bien formadas que pueden ser verificadas sólo al final, invocar manejadores, y puede limpiar los recursos asignados durante el análisis.

`IncrementalParser.reset()`

Este método es llamado después de que el cierre ha sido llamado para restablecer el analizador, de forma que esté listo para analizar nuevos documentos. Los resultados de llamar `parse` o `feed` después del cierre sin llamar a `reset` son indefinidos.

20.12.3 Objetos localizadores

Las instancias de `Locator` proveen estos métodos:

`Locator.getColumnNumber()`

Retorna el número de columna donde el evento actual comienza.

`Locator.getLineNumber()`

Retorna el número de línea donde el evento actual comienza.

`Locator.getPublicId()`

Retorna el identificador público para el evento actual.

`Locator.getSystemId()`

Retorna el identificador de sistema para el evento actual.

20.12.4 Objetos `InputSource`

`InputSource.setPublicId(id)`

Establece el identificador público de esta `InputSource`.

`InputSource.getPublicId()`

Retorna el identificador público para esta `InputSource`.

`InputSource.setSystemId(id)`

Establece el identificador de sistema de esta `InputSource`.

`InputSource.getSystemId()`

Retorna el identificador de sistema para esta `InputSource`.

`InputSource.setEncoding(encoding)`

Establece la codificación de caracteres para esta `InputSource`.

La codificación debe ser una cadena aceptable para una declaración de codificación XML (vea la sección 4.3.3 de la recomendación XML).

El atributo de codificación de la `InputSource` es ignorado si la `InputSource` contiene también un flujo de caracteres.

`InputSource.getEncoding()`

Obtiene la codificación de caracteres de esta `InputSource`.

`InputSource.setByteStream(bytefile)`

Establece el flujo de bytes (un *binary file*) para esta fuente de entrada.

El analizador SAX ignorará esto si existe también un flujo de caracteres especificado, pero utilizará un flujo de bytes en preferencia para abrir una conexión URI en sí.

Si la aplicación conoce la codificación de caracteres del flujo de bytes, debería establecerla con el método `setEncoding`.

`InputSource.getByteStream()`

Obtiene el flujo de bytes para esta fuente de entrada.

El método `getEncoding` retornará la codificación de caracteres para este flujo de bytes, o `None` si se desconoce.

`InputSource.setCharacterStream(charfile)`

Establece el flujo de caracteres (un *text file*) para esta fuente de entrada.

Si existe un flujo de caracteres especificado, el analizador SAX ignorará cualquier flujo de bytes y no intentará abrir una conexión URI al identificador de sistema.

`InputSource.getCharacterStream()`

Obtiene el flujo de caracteres para esta fuente de entrada.

20.12.5 La Interfaz `Attributes`

Los objetos `Attributes` implementa una porción del *mapping protocol*, incluyendo los métodos `copy()`, `get()`, `__contains__()`, `items()`, `keys()`, y `values()`. Los siguientes métodos también son provistos:

`Attributes.getLength()`

Retorna el número de atributos.

`Attributes.getNames()`

Retorna los nombres de los atributos.

`Attributes.getType(name)`

Retorna el tipo del atributo *name*, que es normalmente `'CDATA'`.

`Attributes.getValue(name)`

Retorna el valor del atributo *name*.

20.12.6 La Interfaz `AttributesNS`

Esta interfaz es un subtipo de la interfaz `Attributes` (vea la sección *La Interfaz `Attributes`*). Todos los métodos soportados por la interfaz están también disponibles en los objetos `AttributesNS`.

Los siguientes métodos están también disponibles:

`AttributesNS.getValueByQName(name)`

Retorna el valor para un nombre cualificado.

`AttributesNS.getNameByQName(name)`

Retorna el par (`namespace`, `localname`) para un *name* cualificado.

`AttributesNS.getQNameByName(name)`

Retorna el nombre cualificado para un par (`namespace`, `localname`).

`AttributesNS.getQNames()`

Retorna los nombres cualificados para todos los atributos.

20.13 `xml.parsers.expat` — Análisis rápido XML usando Expat

Advertencia: El módulo `pyexpat` no es seguro contra datos contruidos maliciosamente. Si necesita analizar datos que no son de confianza o no autenticados, consulte [Vulnerabilidades XML](#).

El módulo `xml.parsers.expat` es una interfaz de Python para el analizador XML no validado de Expat. El módulo proporciona un único tipo de extensión, `xmlparser`, que representa el estado actual de un analizador XML. Después de que se haya creado un objeto `xmlparser`, se pueden establecer varios atributos del objeto en funciones de controlador. Cuando se envía un documento XML al analizador, se llaman a las funciones del controlador para los datos de caracteres y el marcado en el documento XML.

Este módulo utiliza el módulo `pyexpat` para proporcionar acceso al analizador Expat. El uso directo del módulo `pyexpat` está obsoleto.

Este módulo proporciona una excepción y un tipo de objeto:

exception `xml.parsers.expat.ExpatError`

La excepción que se lanza cuando Expat informa un error. Consulte la sección [Excepciones de ExpatError](#) para obtener más información sobre cómo interpretar los errores de Expat.

exception `xml.parsers.expat.error`

Alias para [ExpatError](#).

`xml.parsers.expat.XMLParserType`

El tipo de los valores de retorno de la función `ParserCreate()`.

El modulo `xml.parsers.expat` contiene dos funciones:

`xml.parsers.expat.ErrorString(errno)`

Retorna una cadena explicativa para un número de error dado *errno*.

`xml.parsers.expat.ParserCreate(encoding=None, namespace_separator=None)`

Crea y retorna un nuevo objeto `xmlparser`. *encoding*, si se especifica, debe ser una cadena que nombre la codificación utilizada por los datos XML. Expat no admite tantas codificaciones como Python, y su repertorio de codificaciones no se puede ampliar; es compatible con UTF-8, UTF-16, ISO-8859-1 (Latin1) y ASCII. Si se proporciona *encoding*¹, anulará la codificación implícita o explícita del documento.

Expat puede, opcionalmente, realizar el procesamiento del espacio de nombres XML por usted, habilitado al proporcionar un valor para *namespace_separator*. El valor debe ser una cadena de un carácter; a [ValueError](#) se lanzará si la cadena tiene una longitud ilegal (None se considera lo mismo que una omisión). Cuando el procesamiento de espacios de nombres está habilitado, se expandirán los nombres de tipos de elementos y los nombres de atributos que pertenecen a un espacio de nombres. El nombre del elemento pasado a los controladores de elementos `StartElementHandler` y `EndElementHandler` será la concatenación del URI del espacio de nombres, el carácter separador del espacio de nombres y la parte local del nombre. Si el separador del espacio de nombres es un byte cero (`chr(0)`), el URI del espacio de nombres y la parte local se concatenarán sin ningún separador.

Por ejemplo, si *namespace_separator* se establece en un carácter de espacio (`' '`) y se analiza el siguiente documento:

```
<?xml version="1.0"?>
<root xmlns      = "http://default-namespace.org/"
      xmlns:py   = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

¹ La cadena de codificación incluida en la salida XML debe cumplir con los estándares apropiados. Por ejemplo, «UTF-8» es válido, pero «UTF8» no lo es. Consulte <https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl> y <https://www.iana.org/assignments/character-sets/character-sets.xhtml>.

StartElementHandler recibirá las siguientes cadenas para cada elemento:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

Debido a las limitaciones en la biblioteca Expat utilizada por pyexpat, la instancia `xmlparser` retorna solo se puede usar para analizar un solo documento XML. Llame a `ParserCreate` para cada documento para proporcionar instancias de analizador únicas.

Ver también:

[El Expat XML Parser](#) Página de inicio del proyecto Expat.

20.13.1 Objetos XMLParser

Los objetos `xmlparser` tienen los siguientes métodos:

`xmlparser.Parse(data[, isfinal])`

Analiza el contenido de la cadena *data*, llamando a las funciones del controlador apropiadas para procesar los datos analizados. *isfinal* debe ser verdadero en la última llamada a este método; permite el análisis de un solo archivo en fragmentos, no el envío de varios archivos. *data* puede ser la cadena vacía en cualquier momento.

`xmlparser.ParseFile(file)`

Analizar la lectura de datos XML del objeto *file*. *file* solo necesita proporcionar el método `read(nbytes)`, devolviendo la cadena vacía cuando no hay más datos.

`xmlparser.SetBase(base)`

Establece la base que se utilizará para resolver URIs relativos en identificadores de sistema en declaraciones. La resolución de los identificadores relativos se deja en manos de la aplicación: este valor se pasará como el argumento *base* a las funciones `ExternalEntityRefHandler()`, `NotationDeclHandler()`, y `UnparsedEntityDeclHandler()`.

`xmlparser.GetBase()`

Retorna una cadena que contiene la base establecida por una llamada anterior a `SetBase()`, o `None` si no se ha llamado a `SetBase()`.

`xmlparser.GetInputContext()`

Retorna los datos de entrada que generaron el evento actual como una cadena. Los datos están en la codificación de la entidad que contiene el texto. Cuando se llama mientras un controlador de eventos no está activo, el valor de retorno es `None`.

`xmlparser.ExternalEntityParserCreate(context[, encoding])`

Cree un analizador «child» que se pueda utilizar para analizar una entidad analizada externa a la que hace referencia el contenido analizado por el analizador principal. El parámetro *context* debe ser la cadena pasada a la función del controlador `ExternalEntityRefHandler()`, que se describe a continuación. El analizador secundario se crea con `order_attributes` y `specific_attributes` establecidos en los valores de este analizador.

`xmlparser.SetParamEntityParsing(flag)`

Controle el análisis de las entidades de parámetros (incluido el subconjunto DTD externo). Los posibles valores de *flag* son `XML_PARAM_ENTITY_PARSING_NEVER`, `XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE` y `XML_PARAM_ENTITY_PARSING_ALWAYS`. Retorna verdadero si el establecimiento de la bandera fue exitoso.

`xmlparser.UseForeignDTD([flag])`

Llamar a esto con un valor verdadero para *flag* (el predeterminado) hará que Expat llame a `ExternalEntityRefHandler` con `None` para todos los argumentos para permitir que se cargue una DTD alternativa. Si el documento no contiene una declaración de tipo de documento, se seguirá llamando a `ExternalEntityRefHandler`, pero no se llamará a `StartDoctypeDeclHandler` y `EndDoctypeDeclHandler`.

Pasar un valor falso para *flag* cancelará una llamada anterior que pasó un valor verdadero, pero por lo demás no tiene ningún efecto.

Este método sólo se puede llamar antes de que se llamen los métodos `Parse()` o `ParseFile()`; llamarlo después de que cualquiera de ellos haya sido llamado causa que `ExpatError` se lanza con el atributo `code` establecido en `errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING]`.

`xmlparser.SetReparseDeferralEnabled(enabled)`

Advertencia: Calling `SetReparseDeferralEnabled(False)` has security implications, as detailed below; please make sure to understand these consequences prior to using the `SetReparseDeferralEnabled` method.

Expat 2.6.0 introduced a security mechanism called «reparse deferral» where instead of causing denial of service through quadratic runtime from reparsing large tokens, reparsing of unfinished tokens is now delayed by default until a sufficient amount of input is reached. Due to this delay, registered handlers may — depending of the sizing of input chunks pushed to Expat — no longer be called right after pushing new input to the parser. Where immediate feedback and taking over responsibility of protecting against denial of service from large tokens are both wanted, calling `SetReparseDeferralEnabled(False)` disables reparse deferral for the current Expat parser instance, temporarily or altogether. Calling `SetReparseDeferralEnabled(True)` allows re-enabling reparse deferral.

Note that `SetReparseDeferralEnabled()` has been backported to some prior releases of CPython as a security fix. Check for availability of `SetReparseDeferralEnabled()` using `hasattr()` if used in code running across a variety of Python versions.

Nuevo en la versión 3.8.19.

`xmlparser.GetReparseDeferralEnabled()`

Returns whether reparse deferral is currently enabled for the given Expat parser instance.

Nuevo en la versión 3.8.19.

`xmlparser` los objetos tienen los siguientes atributos:

`xmlparser.buffer_size`

El tamaño del búfer usado cuando `buffer_text` es verdadero. Se puede establecer un nuevo tamaño de búfer asignando un nuevo valor entero a este atributo. Cuando se cambia el tamaño, el búfer se vaciará.

`xmlparser.buffer_text`

Establecer esto en `true` hace que el objeto `xmlparser` almacene el contenido textual retornado por Expat para evitar múltiples llamadas a la devolución de llamada `CharacterDataHandler()` siempre que sea posible. Esto puede mejorar sustancialmente el rendimiento ya que Expat normalmente divide los datos de los caracteres en trozos al final de cada línea. Este atributo es falso por defecto y se puede cambiar en cualquier momento.

`xmlparser.buffer_used`

Si `buffer_text` está habilitado, el número de bytes almacenados en el búfer. Estos bytes representan texto codificado en UTF-8. Este atributo no tiene una interpretación significativa cuando `buffer_text` es falso.

`xmlparser.ordered_attributes`

Establecer este atributo en un número entero distinto de cero hace que los atributos se informen como una lista en lugar de un diccionario. Los atributos se presentan en el orden que se encuentran en el texto del documento. Para cada atributo, se presentan dos entradas de lista: el nombre del atributo y el valor del atributo. (Las versiones anteriores de este módulo también usaban este formato). De forma predeterminada, este atributo es falso; se puede cambiar en cualquier momento.

`xmlparser.specified_attributes`

Si se establece en un número entero distinto de cero, el analizador informará solo los atributos que se especificaron en la instancia del documento y no los que se derivaron de declaraciones de atributos. Las aplicaciones

que establecen esto deben tener especial cuidado al utilizar la información adicional disponible en las declaraciones según sea necesario para cumplir con los estándares para el comportamiento de los procesadores XML. De forma predeterminada, este atributo es falso; se puede cambiar en cualquier momento.

Los siguientes atributos contienen valores relacionados con el error más reciente encontrado por un objeto `xmlparser`, y solo tendrán los valores correctos una vez que una llamada a `Parse()` o `ParseFile()` haya lanzado una excepción `xml.parsers.expat.ExpatError`.

`xmlparser.ErrorByteIndex`

Índice de bytes en el que se produjo un error.

`xmlparser.ErrorCode`

Código numérico que especifica el problema. Este valor puede pasarse a la función `ErrorString()`, o compararse con una de las constantes definidas en el objeto `errors`.

`xmlparser.ErrorColumnNumber`

Número de columna en la que se produjo un error.

`xmlparser.ErrorLineNumber`

Número de línea en la que ocurrió un error.

Los siguientes atributos contienen valores relacionados con la ubicación actual del análisis en un objeto `xmlparser`. Durante una devolución de llamada que informa un evento de análisis, indican la ubicación del primero de la secuencia de caracteres que generó el evento. Cuando se llama fuera de una devolución de llamada, la posición indicada estará justo después del último evento de análisis (independientemente de si hubo una devolución de llamada asociada).

`xmlparser.CurrentByteIndex`

Índice de bytes actual en la entrada del analizador.

`xmlparser.CurrentColumnNumber`

Número de columna actual en la entrada del analizador.

`xmlparser.CurrentLineNumber`

Número de línea actual en la entrada del analizador.

Aquí está la lista de controladores que se pueden configurar. Para configurar un controlador en un objeto `xmlparser o`, use `o.handlername = func`. *handlername* debe tomarse de la siguiente lista, y *func* debe ser un objeto invocable que acepte el número correcto de argumentos. Los argumentos son todas cadenas, a menos que se indique lo contrario.

`xmlparser.XmlDeclHandler (version, encoding, standalone)`

Se llama cuando se analiza la declaración XML. La declaración XML es la declaración (opcional) de la versión aplicable de la recomendación XML, la codificación del texto del documento y una declaración «independiente» opcional. *version* y *encoding* serán cadenas, y *standalone* será 1 si el documento se declara independiente, 0 si se declara no independiente o -1 si se omitió la cláusula independiente. Esto solo está disponible con la versión Expat 1.95.0 o más reciente.

`xmlparser.StartDoctypeDeclHandler (doctypeName, systemId, publicId, has_internal_subset)`

Se llama cuando Expat comienza a analizar la declaración del tipo de documento (`<! DOCTYPE ...`). El *doctypeName* se proporciona exactamente como se presenta. Los parámetros *systemId* y *publicId* dan el sistema y los identificadores públicos si se especifican, o `None` si se omite. *has_internal_subset* será verdadero si el documento contiene un subconjunto de declaración de documento interno. Esto requiere Expat versión 1.2 o más reciente.

`xmlparser.EndDoctypeDeclHandler ()`

Se llama cuando Expat termina de analizar la declaración del tipo de documento. Esto requiere Expat versión 1.2 o más reciente.

`xmlparser.ElementDeclHandler (name, model)`

Se llama una vez para cada declaración de tipo de elemento. *name* es el nombre del tipo de elemento y *model* es una representación del modelo de contenido.

`xmlparser.AttnlistDeclHandler (elname, attname, type, default, required)`

Se llama para cada atributo declarado para un tipo de elemento. Si una declaración de lista de atributos declara tres atributos, este controlador se llama tres veces, una para cada atributo. *elname* es el nombre del elemento al que se aplica la declaración y *attname* es el nombre del atributo declarado. El tipo de atributo es una cadena

pasada como *type*; los valores posibles son 'CDATA', 'ID', 'IDREF', ... *default* da el valor predeterminado para el atributo utilizado cuando el atributo no está especificado por el instancia de documento, o `None` si no hay un valor predeterminado (valores #IMPLIED). Si se requiere que el atributo se proporcione en la instancia del documento, *required* será verdadero. Esto requiere la versión Expat 1.95.0 o más reciente.

`xmlparser.StartElementHandler` (*name*, *attributes*)

Llamado para el inicio de cada elemento. *name* es una cadena que contiene el nombre del elemento, y *attributes* son los atributos del elemento. Si *order_attributes* es verdadero, esta es una lista (ver *order_attributes* para una descripción completa). De lo contrario, es un diccionario que asigna nombres a valores.

`xmlparser.EndElementHandler` (*name*)

Llamado al final de cada elemento.

`xmlparser.ProcessingInstructionHandler` (*target*, *data*)

Llamado para cada instrucción de procesamiento.

`xmlparser.CharacterDataHandler` (*data*)

Llamado para datos de personajes. Esto se llamará para datos de caracteres normales, contenido marcado CDATA y espacios en blanco ignorables. Las aplicaciones que deben distinguir estos casos pueden usar las devoluciones de llamada *StartCdataSectionHandler*, *EndCdataSectionHandler*, y *ElementDeclHandler* para recopilar la información requerida.

`xmlparser.UnparsedEntityDeclHandler` (*entityName*, *base*, *systemId*, *publicId*, *notationName*)

Se llama para declaraciones de entidad sin analizar (NDATA). Esto solo está presente para la versión 1.2 de la biblioteca Expat; para versiones más recientes, use *EntityDeclHandler* en su lugar. (La función subyacente en la biblioteca Expat se ha declarado obsoleta).

`xmlparser.EntityDeclHandler` (*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*, *notationName*)

Llamado para todas las declaraciones de entidad. Para parámetros y entidades internas, *value* será una cadena que proporciona el contenido declarado de la entidad; esto será `None` para entidades externas. El parámetro *notationName* será `None` para las entidades analizadas y el nombre de la notación para las entidades no analizadas. *is_parameter_entity* será verdadero si la entidad es una entidad de parámetro o falso para las entidades generales (la mayoría de las aplicaciones solo deben preocuparse por las entidades generales). Esto solo está disponible a partir de la versión 1.95.0 de la biblioteca Expat.

`xmlparser.NotationDeclHandler` (*notationName*, *base*, *systemId*, *publicId*)

Se llama para declaraciones de notación. *notationName*, *base* y *systemId* y *publicId* son cadenas si se dan. Si se omite el identificador público, *publicId* será `None`.

`xmlparser.StartNamespaceDeclHandler` (*prefix*, *uri*)

Se llama cuando un elemento contiene una declaración de espacio de nombres. Las declaraciones de espacio de nombres se procesan antes de que se llame a *StartElementHandler* para el elemento en el que se colocan las declaraciones.

`xmlparser.EndNamespaceDeclHandler` (*prefix*)

Se llama cuando se alcanza la etiqueta de cierre para un elemento que contiene una declaración de espacio de nombres. Esto se llama una vez para cada declaración de espacio de nombres en el elemento en el orden inverso al que se llamó *StartNamespaceDeclHandler* para indicar el inicio del alcance de cada declaración de espacio de nombres. Las llamadas a este controlador se realizan después del correspondiente *EndElementHandler* para el final del elemento.

`xmlparser.CommentHandler` (*data*)

Llamado para comentarios. *data* es el texto del comentario, excluyendo el '`<!--`' inicial y el final '`-->`'.

`xmlparser.StartCdataSectionHandler` ()

Llamado al comienzo de una sección CDATA. Esto y *EndCdataSectionHandler* son necesarios para poder identificar el inicio sintáctico y el final de las secciones CDATA.

`xmlparser.EndCdataSectionHandler` ()

Llamado al final de una sección CDATA.

`xmlparser.DefaultHandler` (*data*)

Se invoca por cualquier carácter del documento XML para el que no se ha especificado ningún controlador

aplicable. Esto significa caracteres que forman parte de una construcción que se podría informar, pero para los que no se ha proporcionado ningún controlador.

`xmlparser.DefaultHandlerExpand(data)`

Es lo mismo que `DefaultHandler()`, pero no inhibe la expansión de entidades internas. La referencia de la entidad no se pasará al controlador predeterminado.

`xmlparser.NotStandaloneHandler()`

Se llama si el documento XML no se ha declarado como un documento independiente. Esto sucede cuando hay un subconjunto externo o una referencia a una entidad de parámetro, pero la declaración XML no establece independiente en `yes` en una declaración XML. Si este controlador retorna 0, el analizador lanzará un error `XML_ERROR_NOT_STANDALONE`. Si este controlador no está configurado, el analizador no lanza ninguna excepción para esta condición.

`xmlparser.ExternalEntityRefHandler(context, base, systemId, publicId)`

Llamado para referencias a entidades externas. `base` es la base actual, según lo establecido por una llamada anterior a `SetBase()`. Los identificadores público y del sistema, `systemId` y `publicId`, son cadenas si se dan; si no se proporciona el identificador público, `publicId` será `None`. El valor `context` es opaco y solo debe usarse como se describe a continuación.

Para que se analicen las entidades externas, se debe implementar este controlador. Es responsable de crear el sub-analizador usando `ExternalEntityParserCreate(context)`, inicializándolo con las devoluciones de llamada apropiadas y analizando la entidad. Este controlador debería devolver un número entero; si retorna 0, el analizador lanzará un error `XML_ERROR_EXTERNAL_ENTITY_HANDLING`; de lo contrario, el análisis continuará.

Si no se proporciona este controlador, las entidades externas se informan mediante la devolución de llamada `DefaultHandler`, si se proporciona.

20.13.2 Excepciones de `ExpatriError`

Las excepciones `ExpatriError` tienen una serie de atributos interesantes:

`ExpatriError.code`

Número de error interno del expatriado para el error específico. El diccionario `errors.messages` asigna estos números de error a los mensajes de error de Expat. Por ejemplo:

```
from xml.parsers.expat import ParserCreate, ExpatriError, errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatriError as err:
    print("Error:", errors.messages[err.code])
```

El módulo `errors` también proporciona constantes de mensajes de error y un diccionario `codes` mapeando estos mensajes a los códigos de error, ver más abajo.

`ExpatriError.lineno`

Número de línea en la que se detectó el error. La primera línea está numerada como 1.

`ExpatriError.offset`

Carácter desplazado en la línea donde ocurrió el error. La primera columna está numerada como 0.

20.13.3 Ejemplo

El siguiente programa define tres controladores que simplemente imprimen sus argumentos.

```
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)
```

La salida de este programa es:

```
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
```

20.13.4 Descripciones del modelo de contenido

Los modelos de contenido se describen mediante tuplas anidadas. Cada tupla contiene cuatro valores: el tipo, el cuantificador, el nombre y una tupla de niños. Los niños son simplemente descripciones adicionales del modelo de contenido.

Los valores de los dos primeros campos son constantes definidas en el módulo `xml.parsers.expat.model`. Estas constantes se pueden recopilar en dos grupos: el grupo de tipo de modelo y el grupo de cuantificador.

Las constantes en el grupo de tipos de modelo son:

`xml.parsers.expat.model.XML_CTYPE_ANY`

Se declaró que el elemento nombrado por el nombre del modelo tiene un modelo de contenido de ANY.

`xml.parsers.expat.model.XML_CTYPE_CHOICE`

El elemento nombrado permite elegir entre varias opciones; se utiliza para modelos de contenido como (A | B | C).

`xml.parsers.expat.model.XML_CTYPE_EMPTY`

Los elementos que se declaran EMPTY tienen este tipo de modelo.

`xml.parsers.expat.model.XML_CTYPE_MIXED`

`xml.parsers.expat.model.XML_CTYPE_NAME`

`xml.parsers.expat.model.XML_CTYPE_SEQ`

Los modelos que representan una serie de modelos que siguen uno tras otro se indican con este tipo de modelo. Se utiliza para modelos como (A, B, C).

Las constantes en el grupo cuantificador son:

`xml.parsers.expat.model.XML_CQUANT_NONE`

No se proporciona ningún modificador, por lo que puede aparecer exactamente una vez, como para A.

`xml.parsers.expat.model.XML_CQUANT_OPT`

El modelo es opcional: puede aparecer una vez o no aparecer, como para A?

`xml.parsers.expat.model.XML_CQUANT_PLUS`

El modelo debe aparecer una o más veces (como A+).

`xml.parsers.expat.model.XML_CQUANT_REP`

El modelo debe aparecer cero o más veces, como en A*.

20.13.5 Constantes de error de expansión

Las siguientes constantes se proporcionan en el módulo `xml.parsers.expat.errors`. Estas constantes son útiles para interpretar algunos de los atributos de los objetos de excepción `ExpatError` que se lanzaran cuando se produce un error. Dado que, por razones de compatibilidad con versiones anteriores, el valor de las constantes es el `message` de error y no el `code` de error numérico, puede hacer esto comparando su atributo `code` con `errors.codes[errors.XML_ERROR_CONSTANT_NAME]`.

El módulo `errors` tiene los siguientes atributos:

`xml.parsers.expat.errors.codes`

Un diccionario que asigna descripciones de cadenas a sus códigos de error.

Nuevo en la versión 3.2.

`xml.parsers.expat.errors.messages`

Un diccionario que asigna códigos de error numéricos a sus descripciones de cadenas.

Nuevo en la versión 3.2.

`xml.parsers.expat.errors.XML_ERROR_ASYNC_ENTITY`

`xml.parsers.expat.errors.XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF`

Una referencia de entidad en un valor de atributo se refiere a una entidad externa en lugar de una entidad interna.

`xml.parsers.expat.errors.XML_ERROR_BAD_CHAR_REF`

Una referencia de carácter se refiere a un carácter que es ilegal en XML (por ejemplo, carácter 0, o “�”).

`xml.parsers.expat.errors.XML_ERROR_BINARY_ENTITY_REF`

Una referencia de entidad se refería a una entidad que se declaró con una notación, por lo que no se puede analizar.

`xml.parsers.expat.errors.XML_ERROR_DUPLICATE_ATTRIBUTE`

Un atributo se utilizó más de una vez en una etiqueta de inicio.

`xml.parsers.expat.errors.XML_ERROR_INCORRECT_ENCODING`

`xml.parsers.expat.errors.XML_ERROR_INVALID_TOKEN`

Se lanza cuando un byte de entrada no se puede asignar correctamente a un carácter; por ejemplo, un byte NUL (valor 0) en un flujo de entrada UTF-8.

`xml.parsers.expat.errors.XML_ERROR_JUNK_AFTER_DOC_ELEMENT`

Se produjo algo diferente a los espacios en blanco después del elemento del documento.

`xml.parsers.expat.errors.XML_ERROR_MISPLACED_XML_PI`

Se encontró una declaración XML en algún lugar que no sea el comienzo de los datos de entrada.

`xml.parsers.expat.errors.XML_ERROR_NO_ELEMENTS`
El documento no contiene elementos (XML requiere que todos los documentos contengan exactamente un elemento de nivel superior)..

`xml.parsers.expat.errors.XML_ERROR_NO_MEMORY`
Expat no pudo asignar memoria internamente.

`xml.parsers.expat.errors.XML_ERROR_PARAM_ENTITY_REF`
Se encontró una referencia de entidad de parámetro donde no estaba permitida.

`xml.parsers.expat.errors.XML_ERROR_PARTIAL_CHAR`
Se encontró un carácter incompleto en la entrada.

`xml.parsers.expat.errors.XML_ERROR_RECURSIVE_ENTITY_REF`
Una referencia de entidad contenía otra referencia a la misma entidad; posiblemente a través de un nombre diferente, y posiblemente indirectamente.

`xml.parsers.expat.errors.XML_ERROR_SYNTAX`
Se encontró algún error de sintaxis no especificado.

`xml.parsers.expat.errors.XML_ERROR_TAG_MISMATCH`
Una etiqueta final no coincidía con la etiqueta inicial abierta más interna.

`xml.parsers.expat.errors.XML_ERROR_UNCLOSED_TOKEN`
Algún token (como una etiqueta de inicio) no se cerró antes del final de la transmisión o se encontró el siguiente token.

`xml.parsers.expat.errors.XML_ERROR_UNDEFINED_ENTITY`
Se hizo referencia a una entidad que no estaba definida.

`xml.parsers.expat.errors.XML_ERROR_UNKNOWN_ENCODING`
La codificación del documento no es compatible con Expat.

`xml.parsers.expat.errors.XML_ERROR_UNCLOSED_CDATA_SECTION`
No se cerró una sección marcada con CDATA.

`xml.parsers.expat.errors.XML_ERROR_EXTERNAL_ENTITY_HANDLING`

`xml.parsers.expat.errors.XML_ERROR_NOT_STANDALONE`
El analizador determinó que el documento no era «independiente» aunque se declaró en la declaración XML, y el `NotStandaloneHandler` se estableció y devolvió 0.

`xml.parsers.expat.errors.XML_ERROR_UNEXPECTED_STATE`

`xml.parsers.expat.errors.XML_ERROR_ENTITY_DECLARED_IN_PE`

`xml.parsers.expat.errors.XML_ERROR_FEATURE_REQUIRES_XML_DTD`
Se solicitó una operación que requiere que se compile el soporte DTD, pero Expat se configuró sin soporte DTD. Esto nunca debería ser informado por una compilación estándar del módulo `xml.parsers.expat`.

`xml.parsers.expat.errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING`
Se solicitó un cambio de comportamiento después de que comenzó el análisis que solo se puede cambiar antes de que haya comenzado el análisis. Esto (actualmente) solo lanzado por `UseForeignDTD()`.

`xml.parsers.expat.errors.XML_ERROR_UNBOUND_PREFIX`
Se encontró un prefijo no declarado cuando se habilitó el procesamiento del espacio de nombres.

`xml.parsers.expat.errors.XML_ERROR_UNDECLARING_PREFIX`
El documento intentó eliminar la declaración de espacio de nombres asociada con un prefijo.

`xml.parsers.expat.errors.XML_ERROR_INCOMPLETE_PE`
El documento no contenía ningún elemento de documento.

`xml.parsers.expat.errors.XML_ERROR_XML_DECL`
El documento no contenía ningún elemento de documento.

`xml.parsers.expat.errors.XML_ERROR_TEXT_DECL`
Se produjo un error al analizar una declaración de texto en una entidad externa.

`xml.parsers.expat.errors.XML_ERROR_PUBLICID`

Se encontraron caracteres en la identificación pública que no están permitidos.

`xml.parsers.expat.errors.XML_ERROR_SUSPENDED`

La operación solicitada se realizó en un analizador suspendido, pero no está permitida. Esto incluye intentos de proporcionar información adicional o detener el analizador.

`xml.parsers.expat.errors.XML_ERROR_NOT_SUSPENDED`

Se realizó un intento de reanudar el analizador cuando no se había suspendido.

`xml.parsers.expat.errors.XML_ERROR_ABORTED`

Esto no se debe informar a las aplicaciones Python.

`xml.parsers.expat.errors.XML_ERROR_FINISHED`

La operación solicitada se realizó en un analizador que terminó de analizar la entrada, pero no está permitido. Esto incluye intentos de proporcionar información adicional o detener el analizador.

`xml.parsers.expat.errors.XML_ERROR_SUSPEND_PE`

Notas al pie

Protocolos y soporte de Internet

Los módulos descritos en este capítulo implementan protocolos de Internet y soporte para la tecnología relacionada. Todos ellos se implementan en Python. La mayoría de estos módulos requieren la presencia del módulo dependiente del sistema *socket*, que actualmente es compatible con las plataformas más populares. Aquí hay una visión general:

21.1 *webbrowser* — Cómodo controlador de navegador web

Código fuente: [Lib/webbrowser.py](#)

El módulo *webbrowser* provee una interfaz de alto nivel que permite desplegar documentos basados en la web a los usuarios. Bajo la mayoría de circunstancias, simplemente invocando la función *open()* desde este módulo se realizará la acción correcta.

En Unix, son privilegiados los navegadores gráficos bajo X11, pero serán usados navegadores en modo texto si los navegadores gráficos no están disponibles o un *display* X11 no está disponible. Si se usan navegadores en modo texto, el proceso invocador será bloqueado hasta que el usuario salga del navegador.

Si existe la variable de entorno *BROWSER*, esta es interpretada como la lista de navegadores a probar antes de los predeterminados de la plataforma, separados por *os.pathsep*. Cuando el valor de una parte de la lista contiene la cadena *%s*, este es interpretado como una línea de comando literal de un navegador a usar con el argumento URL substituido por *%s*; si la parte no contiene *%s*, esta se interpreta simplemente como el nombre del navegador a lanzar.¹

En plataformas no Unix o cuando está disponible un navegador remoto en Unix, el proceso de control no esperará a que el usuario finalice el navegador, sino que permitirá al navegador remoto mantener su propia ventana en la pantalla. Si no están disponibles navegadores remotos en Unix, el proceso de control lanzará un nuevo navegador y esperará.

El script ***webbrowser*** puede ser usado como una interfaz de línea de comandos para el módulo. Acepta una URL como argumento. Acepta los siguientes parámetros opcionales: *-n* abre la URL en una nueva ventana del navegador, si es posible; *-t* abre la URL en una nueva página del navegador («pestaña»). Las opciones son, naturalmente, mutuamente exclusivas. Ejemplo de uso:

```
python -m webbrowser -t "http://www.python.org"
```

La siguiente excepción es definida:

¹ Los ejecutables nombrados aquí sin una ruta completa serán buscados en los directorios dados en la variable de entorno *PATH*.

exception `webbrowser.Error`

Excepción generada cuando ocurre un error de control de navegador.

Las siguientes funciones son definidas:

`webbrowser.open(url, new=0, autoraise=True)`

Muestra *url* usando el navegador por defecto. Si *new* es 0, se abre la *url* en la misma ventana del navegador si es posible. Si *new* es 1, se abre una nueva ventana del navegador si es posible. Si *new* es 2, se abre una nueva página del navegador («pestaña») si es posible. Si *autoraise* es `True`, la ventana es lanzada si es posible (ten en cuenta que bajo muchos gestores de ventana esto ocurrirá independientemente de la configuración de esta variable).

Ten en cuenta que en algunas plataformas, tratar de abrir un nombre de archivo usando esta función puede funcionar e iniciar el programa asociado del sistema operativo. Sin embargo, esto no es soportado ni portable.

Lanza un *evento de auditoría* `webbrowser.open` con el argumento *url*.

`webbrowser.open_new(url)`

Abre *url* en una nueva ventana del navegador por defecto, si es posible, si no, abre *url* en la única ventana del navegador.

`webbrowser.open_new_tab(url)`

Abre *url* en una nueva página («pestaña») del navegador por defecto, si es posible, si no equivale a `open_new()`.

`webbrowser.get(using=None)`

Retorna un objeto de controlador para el tipo de navegador *using*. Si *using* es `None`, retorna un controlador de un navegador por defecto apropiado para el entorno del invocador.

`webbrowser.register(name, constructor, instance=None, *, preferred=False)`

Registra el tipo de navegador *name*. Una vez que el tipo de navegador es registrado, la función `get()` puede retornar un controlador para ese tipo de navegador. Si no se provee *instance* o es `None`, *constructor* será invocado sin parámetros al crear una instancia cuando sea necesario. Si se provee *instance*, *constructor* no será nunca invocado y puede ser `None`.

Definir *preferred* a `True` hace de este navegador un resultado preferido para una invocación `get()` sin argumento. De otra manera, este punto de entrada sólo es útil si planeas definir la variable `BROWSER` o invocar `get()` con un argumento no vacío correspondiendo con el nombre de un manejador que declares.

Distinto en la versión 3.7: fue añadido el parámetro sólo de palabra clave *preferred*.

Un número de tipos de navegador son predefinidos. Esta tabla muestra los nombres de los tipos que se pueden pasar a la función `get()` y las instancias correspondientes para las clases de los controladores, todas definidas en este módulo.

Nombre de tipo	Nombre de clase	Notas
'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')	
'chrome'	Chrome('chrome')	
'chromium'	Chromium('chromium')	
'chromium-browser'	Chromium('chromium-browser')	

Notas:

- (1) «Konqueror» es el manejador de archivos para el entorno de escritorio KDE para Unix y sólo tiene sentido usarlo si KDE está en ejecución. Alguna forma de detectar de manera confiable KDE sería genial; la variable `KDEDIR` no es suficiente. Ten en cuenta también que el nombre «kfm» es usado incluso utilizando el comando **konqueror** con KDE 2 — la implementación selecciona la mejor estrategia para ejecutar Konqueror.
- (2) Sólo en plataformas Windows.
- (3) Sólo en la plataforma Mac OS X.

Nuevo en la versión 3.3: Ha sido añadido soporte para Chrome/Chromium.

Aquí están algunos ejemplos simples:

```
url = 'http://docs.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url)

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

21.1.1 Objetos controladores de navegador

Los controladores de navegador proveen aquellos métodos que son paralelos a tres de las funciones de conveniencia a nivel del módulo:

`controller.open(url, new=0, autoraise=True)`

Despliega *url* usando el navegador manejado por este controlador. Si *new* es 1, se abre una nueva ventana del navegador si es posible. Si *new* es 2, se abre una nueva página del navegador («pestaña») si es posible.

`controller.open_new(url)`

Abre *url* en una nueva ventana del navegador manejado por este controlador, si es posible, si no, abre *url* en la única ventana del navegador. Alias `open_new()`.

`controller.open_new_tab(url)`

Abre *url* en una nueva página («pestaña») del navegador manejado por este controlador, si es posible, si no equivale a `open_new()`.

Notas al pie

21.2 cgi — Soporte de Interfaz de Entrada Común (CGI)

Código fuente: [Lib/cgi.py](#)

Módulo de soporte para scripts de la Interfaz de Entrada Común (CGI)

Este módulo define una serie de utilidades para el uso de scripts CGI escritos en Python.

21.2.1 Introducción

Un script de CGI es invocado por un servidor HTTP, generalmente para procesar entradas de usuario entregadas mediante un elemento HTML `<FORM>` o `<ISINDEX>`.

Muy a menudo, los scripts CGI viven en el directorio especial `cgi-bin` del servidor. El servidor HTTP coloca todo tipo de información sobre la solicitud (como el nombre de host del cliente, la dirección URL solicitada, la cadena de búsqueda (query string) y muchas otras consultas) en el entorno de shell del script, ejecuta el script y envía la salida del script al cliente.

La entrada del script también está conectada al cliente, y a veces los datos del formulario se leen de esta manera; en otras ocasiones los datos del formulario se pasan a través de la parte «cadena de caracteres de búsqueda (query string)» de la dirección URL. Este módulo está diseñado para ocuparse de los diferentes casos y proporcionar una interfaz más simple al script de Python. También proporciona una serie de utilidades que ayudan en la depuración de scripts, y la última adición es la compatibilidad con cargas de archivos desde un formulario (si el navegador lo admite).

La salida de un script CGI debe constar de dos secciones, separadas por una línea en blanco. La primera sección contiene una serie de encabezados, indicando al cliente qué tipo de datos sigue. El código de Python para generar una sección de encabezado mínima tiene este aspecto:

```
print("Content-Type: text/html")    # HTML is following
print()                            # blank line, end of headers
```

La segunda sección suele ser HTML, lo que permite al software cliente mostrar texto bien formateado con encabezado, imágenes en línea, etc. Aquí está el código Python que imprime una simple pieza de HTML:

```
print("<TITLE>CGI script output</TITLE>")
print("<H1>This is my first CGI script</H1>")
print("Hello, world!")
```


21.2.2 Usando el módulo CGI

Empieza escribiendo `import cgi`.

Cuando escribas un nuevo script, considera añadir estas líneas:

```
import cgitb
cgitb.enable()
```

Esto activa un manejador de excepciones especial que mostrará informes detallados en el explorador Web si se produce algún error. Si prefiere no mostrar en detalle su programa a los usuarios de su script, puede tener los informes guardados en archivos en su lugar, con código como este:

```
import cgitb
cgitb.enable(display=0, logdir="/path/to/logdir")
```

Es muy útil usar esta característica durante el desarrollo de scripts. Los informes producidos por `cgitb` proporcionan información que puede ahorrarle mucho tiempo en el seguimiento de errores. Siempre puede eliminar la línea `cgitb` más adelante cuando haya probado su script y esté seguro de que funciona correctamente.

Para obtener los datos del formulario enviado, utilice la clase `FieldStorage`. Si el formulario contiene caracteres no ASCII, utilice el parámetro de palabra clave `encoding` establecido en el valor de la codificación definida para el documento. Normalmente se encuentra en la etiqueta META en la sección HEAD del documento HTML o en el encabezado `Content-Type`. Esto lee el contenido del formulario de la entrada estándar o del entorno (dependiendo del valor de varias variables de entorno establecidas de acuerdo con el estándar CGI). Dado que puede consumir entrada estándar, se debe crear la instancia solo una vez.

La instancia `FieldStorage` puede ser indexada como un diccionario Python. Permite las pruebas de afiliación con el operador `in`, y también es compatible con el método de diccionario estándar `keys()` y la función incorporada `len()`. Los campos de formulario que contienen cadenas de caracteres vacías son ignoradas y no aparecen en el diccionario; para mantener estos valores, proporciona un valor verdadero para el parámetro de palabra clave opcional `keep_blank_values` al crear la instancia `FieldStorage`.

Por ejemplo, el código siguiente (que supone que el encabezado `Content-Type` y la línea en blanco ya se han impreso) comprueba que los campos `name` y `addr` son ambos establecidos a una cadena de caracteres no vacía:

```
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...
```

Aquí los campos, a los que se accede a través de `form[key]`, son por sí mismos las instancias de `FieldStorage` (o `MiniFieldStorage`, dependiendo de la codificación del formulario). El atributo `value` de la instancia produce el valor de cadena de caracteres del campo. El método `getvalue()` retorna este valor de cadena de caracteres directamente; también acepta un segundo argumento opcional como valor predeterminado para retornar si la clave solicitada no está presente.

Si los datos del formulario enviados contienen más de un campo con el mismo nombre, el objeto recuperado por `form[key]` no es una instancia `FieldStorage` o `MiniFieldStorage`, sino una lista de dichas instancias. De forma similar, en esta situación, `form.getvalue(key)` retornaría una lista de cadenas de caracteres. Si espera esta posibilidad (cuando su formulario HTML contiene múltiples campos con el mismo nombre), utilice el método `getlist()`, que siempre retorna una lista de valores (para que no sea necesario poner en mayúsculas y minúsculas en el caso de un solo elemento). Por ejemplo, este código concatena cualquier número de campos de nombre de usuario, separados por comas:

```
value = form.getlist("username")
usernames = ",".join(value)
```

Si un campo representa un archivo cargado, el acceso al valor a través del atributo `value` o el método `getvalue()` lee todo el archivo en memoria como bytes. Puede que esto no sea lo que quiera que ocurra. Puede probar un archivo cargado probando el atributo `filename` o el atributo `file`. Después, puede leer los datos del atributo `file` antes de que se cierre automáticamente como parte de la recolección de elementos no utilizados de la instancia `FieldStorage` (los métodos `read()` y `readline()` retornarán bytes):

```
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while True:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

Los objetos `FieldStorage` también permiten ser usados en una sentencia `with`, lo que automáticamente los cerrará cuando termine la sentencia.

Si un error es encontrado al obtener el contenido de un archivo cargado (por ejemplo, cuando el usuario interrumpe el envío del formulario haciendo clic en un botón *Atrás* o *Cancelar*), el atributo `done` del objeto para el campo se establecerá en el valor `-1`.

El borrador de archivo de carga estándar presenta la posibilidad de cargar varios archivos desde un campo (utilizando una codificación recursiva *multipart/**). Cuando esto ocurre, el elemento será un elemento similar a un diccionario `FieldStorage`. Esto se puede determinar probando su atributo `type`, que debe ser *multipart/form-data* (o tal vez otro tipo MIME que coincida *multipart/**). En este caso, se puede iterar recursivamente al igual que el objeto de formulario de nivel superior.

Cuando se envía un formulario en el formato «antiguo» (como la cadena de caracteres de consulta (query string) o como una sola parte de datos de tipo *application/x-www-form-urlencoded*), los elementos serán realmente instancias de la clase `MiniFieldStorage`. En este caso, los atributos `list`, `file` y `filename` siempre son `None`.

Un formulario enviado a través de POST que también tiene una cadena de caracteres de consulta (query string) contendrá los elementos `FieldStorage` y `MiniFieldStorage`.

Distinto en la versión 3.4: El atributo `file` se cierra automáticamente con el recolector de basura de la instancia creada `FieldStorage`.

Distinto en la versión 3.5: Agregado soporte para el protocolo de administrador de contexto a la clase `FieldStorage`.

21.2.3 Interfaz de Nivel Superior

La sección anterior explica cómo leer datos de un formulario CGI usando la clase `FieldStorage`. Esta sección describe un nivel de interfaz superior que se añadió a esta clase para permitir que uno lo haga de una manera más legible e intuitiva. La interfaz no hace que las técnicas descritas en las secciones anteriores estén obsoletas – por ejemplo, siguen siendo útiles para procesar la carga de archivos de manera eficiente.

La interfaz consiste en dos métodos simples. Usando los métodos puedes procesar datos de formulario de una manera genérica, sin la necesidad de preocuparte si solo se publicaron uno o más valores con un solo nombre.

En la sección anterior, aprendiste a escribir el siguiente código cada vez que esperabas que un usuario publicara más de un valor con un nombre:

```
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

Esta situación es común, por ejemplo, cuando un formulario contiene un grupo de múltiples casillas de verificación (checkboxes) con el mismo nombre:

```
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

En la mayoría de las situaciones, sin embargo, solo hay un control de formulario con un nombre determinado en un formulario y así, espera y solo necesita un valor asociado con este nombre. Así que escribe un script que contiene, por ejemplo, este código:

```
user = form.getvalue("user").upper()
```

El problema con el código es que nunca debe esperar que un cliente proporcione una entrada válida a los scripts. Por ejemplo, si un usuario curioso anexa otro par `user=foo` a la cadena de caracteres de consulta (query string), el script se bloquearía, porque en esta situación la llamada al método `form.getvalue("user")` retorna una lista en lugar de una cadena de caracteres. Llamar al método `upper()` en una lista no es válido (ya que las listas no tienen un método con este nombre) y se produce una excepción `AttributeError`.

Por lo tanto, la forma adecuada de leer los valores de datos de formulario era usar siempre el código que comprueba si el valor obtenido es un valor único o una lista de valores. Eso es molesto y conduce a scripts menos legibles.

Un enfoque más conveniente es utilizar los métodos `getfirst()` y `getlist()` proporcionados por esta interfaz de nivel superior.

`FieldStorage.getfirst(name, default=None)`

Este método siempre retorna solo un valor asociado con el campo de formulario *name*. El método retorna solo el primer valor en caso de que se registraran más valores con dicho nombre. Tenga en cuenta que el orden en que se reciben los valores puede variar de un navegador a otro y no debe darse por sentado.¹ Si no existe ningún campo o valor de formulario, el método retorna el valor especificado por el parámetro opcional *default*. Este parámetro tiene como valor predeterminado `None` si no se especifica.

`FieldStorage.getlist(name)`

Este método siempre retorna una lista de valores asociados con el campo de formulario *name*. El método retorna una lista vacía si no existe tal campo o valor de formulario para *name*. Retorna una lista que consta de un elemento si solo existe un valor de este tipo.

Usando estos métodos puede escribir código compacto agradable:

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()    # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

21.2.4 Funciones

Estas son útiles si desea más control, o si desea emplear algunos de los algoritmos implementados en este módulo en otras circunstancias.

`cgi.parse(fp=None, environ=os.environ, keep_blank_values=False, strict_parsing=False, separator="&")`

Parse a query in the environment or from a file (the file defaults to `sys.stdin`). The *keep_blank_values*, *strict_parsing* and *separator* parameters are passed to `urllib.parse.parse_qs()` unchanged.

Distinto en la versión 3.8.8: Added the *separator* parameter.

`cgi.parse_multipart(fp, pdict, encoding="utf-8", errors="replace", separator="&")`

Analiza la entrada de tipo *multipart/form-data* (para cargas de archivos). Los argumentos son *fp* para el archivo de entrada, *pdict* para un diccionario que contiene otros parámetros en el encabezado *Content-Type* y *encoding*, la codificación de la solicitud.

Retorna un diccionario al igual que `urllib.parse.parse_qs()`: las claves son los nombres de campo, cada valor es una lista de valores para ese campo. Para los campos que no son de archivo, el valor es una lista de cadenas de caracteres.

¹ Tenga en cuenta que algunas versiones recientes de las especificaciones de HTML establecen en qué orden se deben suministrar los valores de campo, pero saber si se recibió una solicitud de un navegador adaptado, o incluso desde un navegador siquiera, es tedioso y propenso a errores.

Esto es fácil de usar pero no muy bueno si espera que se carguen megabytes — en ese caso, utilice la clase `FieldStorage` en su lugar, que es mucho más flexible.

Distinto en la versión 3.7: Se agregaron los parámetros *encoding* y *errors*. Para los campos que no son de archivo, el valor es ahora una lista de cadenas de caracteres, no bytes.

Distinto en la versión 3.8.8: Added the *separator* parameter.

`cgi.parse_header(string)`

Analiza un encabezado MIME (como *Content-Type*) en un valor principal y un diccionario de parámetros.

`cgi.test()`

Script CGI de prueba robusto, usable como programa principal. Escribe encabezados HTTP mínimos y formatea toda la información proporcionada al script en formato HTML.

`cgi.print_environ()`

Da formato al entorno del shell en HTML.

`cgi.print_form(form)`

Da formato a un formulario en HTML.

`cgi.print_directory()`

Da formato al directorio actual en HTML.

`cgi.print_environ_usage()`

Imprime una lista de variables de entorno útiles (utilizadas por CGI) en HTML.

21.2.5 Preocuparse por la seguridad

Hay una regla importante: si invoca un programa externo (a través de las funciones `os.system()` o `os.popen()` u otras con una funcionalidad similar), asegúrese de no pasar cadenas de caracteres arbitrarias recibidas del cliente al shell. Este es una brecha de seguridad muy conocida por la que los hackers inteligentes en cualquier lugar de la Web pueden explotar un inocente script CGI para invocar comandos de shell arbitrarios. Incluso partes de la URL o nombres de campo pueden no ser confiables, ya que la solicitud no tiene que venir de su formulario!

Para estar en el lado seguro, si debe pasar una cadena de caracteres de un formulario a un comando de shell, debe asegurarse de que la cadena de caracteres contiene solo caracteres alfanuméricos, guiones, guiones bajos y puntos.

21.2.6 Instalando su script de CGI en un sistema Unix

Lea la documentación del servidor HTTP y consulte con el administrador del sistema local para encontrar el directorio donde se deben instalar los scripts CGI; por lo general, esto se encuentra en un directorio `cgi-bin` en el árbol del servidor.

Asegúrese de que el script es legible y ejecutable por «otros»; el modo de archivo Unix debe ser octal `00755` (utilice `chmod 0755 filename`). Asegúrese de que la primera línea del script contiene `#!` a partir de la columna 1 seguida del nombre de ruta del intérprete de Python, por ejemplo:

```
#!/usr/local/bin/python
```

Asegúrese que el intérprete de Python exista y sea ejecutable por «otros».

Asegúrese de que cualquier archivo que su script necesite leer o escribir sea legible o tenga permiso de escritura, respectivamente, por «otros» — su modo debe ser `00644` para legible y `00666` para escribir. Esto se debe a que, por razones de seguridad, el servidor HTTP ejecuta el script como usuario «nadie», sin ningún privilegio especial. Sólo puede leer (escribir, ejecutar) archivos que todo el mundo puede leer (escribir, ejecutar). El directorio actual en tiempo de ejecución también es diferente (normalmente es el directorio `cgi-bin` del servidor) y el conjunto de variables de entorno también es diferente de lo que se obtiene al iniciar sesión. En particular, no cuente con la ruta de búsqueda del shell para ejecutables (`PATH`) o la ruta de búsqueda del módulo Python (`PYTHONPATH`) que se establecerá en cualquier cosa interesante.

Si necesita cargar módulos desde un directorio el cual no está en la ruta de búsqueda de módulos predeterminada de Python, puede cambiar la ruta en su script, antes de importar otros módulos. Por ejemplo:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(¡De esta manera, el directorio insertado de último será buscado primero!)

Las instrucciones para sistemas que no son Unix pueden variar; consulte la documentación de su servidor HTTP (usualmente tendrá una sección sobre scripts CGI).

21.2.7 Probando su script de CGI

Desafortunadamente, un script CGI generalmente no se ejecutará cuando lo pruebe desde la línea de comandos, y un script que funcione perfectamente desde la línea de comandos puede fallar misteriosamente cuando se ejecuta desde el servidor. Hay una razón por la que debe probar el script desde la línea de comandos: si contiene un error de sintaxis, el intérprete de Python no lo ejecutará en absoluto y lo más probable es que el servidor HTTP envíe un error críptico al cliente.

Asumiendo que su script no tiene errores de sintaxis, pero este no funciona, no tiene más opción que leer la siguiente sección.

21.2.8 Depurando scripts de CGI

En primer lugar, compruebe si hay errores de instalación triviales — leer la sección anterior sobre la instalación cuidadosa de su script CGI puede ahorrarle mucho tiempo. Si se pregunta si ha entendido correctamente el procedimiento de instalación, intente instalar una copia de este archivo de módulo (`cgi.py`) como un script CGI. Cuando se invoca como un script, el archivo volcará su entorno y el contenido del formulario en formato HTML. Dele el modo correcto, etc., y envíe una solicitud. Si está instalado en el directorio estándar `cgi-bin`, debería ser posible enviarle una solicitud introduciendo una URL en su navegador de la forma:

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

Si esto da un error de tipo 404, el servidor no puede encontrar el script — quizás necesite instalarlo en un directorio diferente. Si este da otro error, hay un problema con la instalación que debería intentar solucionar antes de continuar. Si obtiene una lista bien formateada del entorno y el contenido del formulario (en este ejemplo, los campos deberían estar listados como «addr» con el valor «At Home» y «name» con el valor «Joe Blow»), el script `cgi.py` ha sido instalado correctamente. Si sigue el mismo procedimiento para su propio script, ya debería poder depurarlo.

El siguiente paso podría ser llamar a la función `test()` del módulo `cgi` de su script: reemplace su código principal con la declaración única

```
cgi.test()
```

Esto debería producir los mismos resultados que los obtenidos al instalar el archivo `cgi.py` en sí.

Cuando un script de Python normal lanza una excepción no controlada (por cualquier razón: de un error tipográfico en un nombre de módulo, un archivo que no se puede abrir, etc.), el intérprete de Python imprime un traceback sutil y sale. Aunque el intérprete de Python seguirá haciendo esto cuando el script CGI lance una excepción, lo más probable es que el traceback termine en uno de los archivos de registro del servidor HTTP o se descarte por completo.

Afortunadamente, una vez que haya logrado que su script ejecute *algún* código, puede enviar fácilmente tracebacks al navegador web utilizando el módulo `cgitb`. Si aún no lo ha hecho, solo añada las líneas:

```
import cgitb
cgitb.enable()
```

al principio de su script. Luego intente ejecutarlo de nuevo; cuando un problema ocurra, debería ver un informe detallado que probablemente muestre la causa del error.

Si sospecha que puede haber un problema al importar el módulo `cgitb`, puede usar un enfoque aún más robusto (que solo usa módulos integrados):

```
import sys
sys.stderr = sys.stdout
print("Content-Type: text/plain")
print()
...your code here...
```

Esto se basa en el intérprete de Python para imprimir el traceback. El tipo de contenido de la salida se establece en texto plano, lo que deshabilita todo el procesamiento HTML. Si el script funciona, el cliente mostrará el código HTML sin formato. Si lanza una excepción, lo más probable es que después que se hayan impreso las dos primeras líneas, se mostrará un traceback. Dado que no se está realizando ninguna interpretación HTML, el traceback será legible.

21.2.9 Problemas comunes y soluciones

- La mayoría de servidores HTTP almacenan en búfer la salida de los scripts CGI hasta que el script esté completado. Esto significa que no es posible mostrar un reporte del progreso en la parte del cliente mientras que el script se esté ejecutando.
- Compruebe las instrucciones de instalación anteriores.
- Verifique los archivos de registro (logs) del servidor HTTP. (¡Usar `tail -f logfile` en una ventana separada puede ser útil!)
- Siempre verifique un script para encontrar errores de sintaxis primero, haciendo algo como `python script.py`.
- Si su script no tiene ningún error sintáctico, pruebe añadiendo `import cgitb; cgitb.enable()` en la parte superior del script.
- Cuando se invoquen programas externos, asegúrese de que pueden ser encontrados. Generalmente esto significa usar nombres de ruta absolutos — `PATH` generalmente no se establece en un valor útil en un script CGI.
- Al leer o escribir archivos externos, asegúrese de que puedan ser leídas o escritas por el `userid` por el que su script CGI se va a ejecutar: es típico que esto sea el `userid` bajo el que el servidor web se está ejecutando, o algún `userid` especificado explícitamente por la función `suexec` del servidor web.
- No intente darle un modo `set-uid` a un script CGI. Esto no funciona en la mayoría de sistemas, además de ser un riesgo de seguridad.

Notas al pie

21.3 `cgitb` — Administrador *traceback* para scripts CGI.

Código fuente: [Lib/cgitb.py](#)

El módulo `cgitb` proporciona un manejador de excepciones especial para script de Python. (Su nombre es un poco engañoso. Fue diseñado originalmente para mostrar una amplia información de *traceback* en HTML para los scripts CGI). Más tarde se generalizó para mostrar también esta información en texto plano). Después de activar este módulo, si se produce una excepción no capturada, se mostrará un informe detallado y formateado. El informe incluye un *traceback* que muestra extractos del código fuente para cada nivel, así como los valores de los argumentos y las variables locales de las funciones que se están ejecutando actualmente, para ayudar a depurar el problema. Opcionalmente, puede guardar esta información en un archivo en lugar de enviarla al navegador.

Para activar esta función, simplemente añada lo siguiente a la parte superior de su script CGI:

```
import cgitb
cgitb.enable()
```

Las opciones de la función `enable()` controlan si el informe se muestra en el explorador y si este se registra en un archivo para su posterior análisis.

```
cgitb.enable(display=1, logdir=None, context=5, format="html")
```

Esta función hace que el módulo `cgitb` se encargue del control predeterminado del intérprete para las excepciones estableciendo el valor de `sys.excepthook`.

El argumento opcional `display` tiene como valor predeterminado 1 y se puede establecer en 0 para suprimir el envío de la traza al navegador. Si el argumento `logdir` está presente, los informes de `traceback` se escriben en los archivos. El valor de `logdir` debe ser un directorio donde se estos archivos serán colocados. El argumento opcional `context` es el número de líneas de contexto que se mostrarán alrededor de la línea actual del código fuente en el `traceback`; esto tiene como valor predeterminado 5. Si el argumento opcional `format` es "html", la salida se formatea como HTML. Cualquier otro valor fuerza la salida de texto sin formato. El valor predeterminado es "html".

```
cgitb.text(info, context=5)
```

Esta función controla la excepción descrita por `info` (una tupla de 3 que contiene el resultado de `sys.exc_info()`), dando formato a su `traceback` como texto y retornando el resultado como una cadena de caracteres. El argumento opcional `context` es el número de líneas de contexto que se mostrarán alrededor de la línea actual de código fuente en el `traceback`; esto tiene como valor predeterminado 5.

```
cgitb.html(info, context=5)
```

Esta función controla la excepción descrita por `info` (una tupla de 3 que contiene el resultado de `sys.exc_info()`), dando formato a su `traceback` como HTML y retornando el resultado como una cadena de caracteres. El argumento opcional `context` es el número de líneas de contexto que se mostrarán alrededor de la línea actual de código fuente en el `traceback`; esto tiene como valor predeterminado 5.

```
cgitb.handler(info=None)
```

Esta función maneja una excepción utilizando la configuración predeterminada (es decir, mostrar un informe en el navegador, pero no registrar en un archivo). Esto puede ser usado cuando has capturado una excepción y quieres reportarla usando `cgitb`. El argumento opcional `info` debería ser una tupla de 3 que contenga un tipo de excepción, un valor de excepción y un objeto de `traceback`, exactamente como la tupla retornada por `sys.exc_info()`. Si no se proporciona el argumento `info`, la excepción actual se obtiene de `sys.exc_info()`.

21.4 wsgiref — Utilidades WSGI e implementación de referencia

La Interfaz de Pasarela del Servidor Web, o *Web Server Gateway Interface* en inglés (WSGI), es una interfaz estándar entre el servidor web y aplicaciones web escritas en Python. Con una interfaz estándar es más sencillo usar una aplicación que soporte WSGI con diferentes servidores web.

Sólo los autores de servidores y frameworks web necesitan conocer cada detalle y caso límite del diseño WSGI. No es necesario conocer cada detalle de WSGI sólo para instalar o escribir una aplicación web usando un *framework* existente.

`wsgiref` es una implementación de referencia de la especificación WSGI que se puede usar para añadir soporte WSGI a un servidor o *framework* web. Este módulo provee utilidades para manipular las variables de entorno WSGI y las cabeceras de respuesta, clases base para implementar servidores WSGI, un servidor HTTP de demostración que sirve aplicaciones WSGI, y una herramienta de validación que comprueba la compatibilidad de servidores y aplicaciones WSGI en base a la especificación [PEP 3333](#).

Vea wsgi.readthedocs.io para más información sobre WSGI, así como enlaces a tutoriales y otros recursos.

21.4.1 `wsgiref.util` – Utilidades de entorno WSGI

Este módulo ofrece una variedad de funciones útiles para trabajar con entornos WSGI. Un entorno WSGI es un diccionario que contiene variables de la petición HTTP, descrito en [PEP 3333](#). Todas las funciones que aceptan un parámetro *environ* esperan un diccionario compatible con WSGI. Por favor, consulte la especificación detallada en [PEP 3333](#).

`wsgiref.util.guess_scheme(environ)`

Retorna una deducción del valor para `wsgi.url_scheme` que debería ser «http» o «https», buscando la variable de entorno HTTPS en el diccionario *environ*. El valor de retorno es una cadena.

Esta función es útil al crear un *gateway* que envuelve CGI o un protocolo similar como FastCGI. Habitualmente, los servidores que ofrecen estos protocolos incluyen una variable HTTPS con el valor «1», «yes», o «on» cuando reciben una petición vía SSL. Así, esta función retorna «https» si encuentra ese valor, o «http», en caso contrario.

`wsgiref.util.request_uri(environ, include_query=True)`

Retorna la URI completa de la petición, opcionalmente la cadena de consulta, usando el algoritmo encontrado en la sección «URL Reconstruction» de [PEP 3333](#). Si *include_query* es falso, la cadena de consulta no se incluye en la URI resultante.

`wsgiref.util.application_uri(environ)`

Similar a `request_uri()` excepto que se ignoran las variables `PATH_INFO` y `QUERY_STRING`. El resultado es la URI base del objeto de aplicación indicado en la petición.

`wsgiref.util.shift_path_info(environ)`

Desplaza un solo nombre de `PATH_INFO` a `SCRIPT_NAME` y retorna el nombre. Se modifica el diccionario *environ*, por lo que se deberá usar una copia si se quiere mantener `PATH_INFO` o `SCRIPT_NAME` intactos.

Si no quedan segmentos en `PATH_INFO`, retornará `None`.

Habitualmente, esta rutina se usa para procesar cada porción de la ruta del URI de la petición, por ejemplo, para usar la ruta como una serie de claves en un diccionario. Esta rutina modifica el entorno pasado para permitir invocar otra aplicación WSGI que esté localizada en la URI objetivo. Por ejemplo, si hay una aplicación WSGI en `/foo`, y la ruta es `/foo/bar/baz`, y la aplicación WSGI en `/foo` llama a `shift_path_info()`, recibirá la cadena «bar», y se actualizará el entorno para pasarlo a una aplicación WSGI en `/foo/bar`. Es decir, `SCRIPT_NAME` cambiará de `/foo` a `/foo/bar` y `PATH_INFO` cambiará de `/bar/baz` a `/baz`.

Cuando `PATH_INFO` es únicamente «/», esta rutina retornará una cadena vacía y añadirá una barra final a `SCRIPT_NAME`, incluso cuando normalmente los segmentos de ruta vacíos se ignoran y `SCRIPT_NAME` no acaba con una barra. Este comportamiento es intencional, para asegurar que una aplicación puede diferenciar entre URIs que acaban en `/x` de las que acaban en `/x/` cuando usan esta rutina para atravesar objetos.

`wsgiref.util.setup_testing_defaults(environ)`

Actualiza *environ* con valores por defecto triviales con propósito de prueba.

Esta rutina añade varios parámetros requeridos por WSGI, incluyendo `HTTP_POST`, `SERVER_NAME`, `SERVER_PORT`, `REQUEST_METHOD`, `SCRIPT_NAME`, `PATH_INFO`, y todas las variables `wsgi.*` definidas en [PEP 3333](#). Sólo ofrece valores por defecto y no reemplaza ningún valor existente en esas variables.

Esta rutina pretende facilitar la preparación de entornos ficticios para pruebas unitarias de servidores y aplicaciones WSGI. NO debería usarse en servidores y aplicaciones WSGI reales, ya que los valores son ficticios!

Ejemplo de uso:

```
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'
```

(continúe en la próxima página)

(proviene de la página anterior)

```

headers = [('Content-type', 'text/plain; charset=utf-8')]

start_response(status, headers)

ret = [("%s: %s\n" % (key, value)).encode("utf-8")
        for key, value in environ.items()]
return ret

with make_server(' ', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()

```

Además de las funciones de entorno previas, el módulo `wsgiref.util` también ofrece las siguientes utilidades varias:

`wsgiref.util.is_hop_by_hop(header_name)`

Retorna True si “header_name” es una cabecera «Hop-by-Hop» HTTP/1.1, como se define en [RFC 2616](#).

class `wsgiref.util.FileWrapper(filelike, blksize=8192)`

Un *wrapper* para convertir un objeto archivo en un *iterator*. Los objetos resultantes soportan los estilos de iteración `__getitem__()` y `__iter__()`, por compatibilidad con Python 2.1 y Jython. A medida que se itera sobre el objeto, el parámetro opcional `blksize` se pasará repetidamente al método `read()` del objeto archivo para obtener cadenas de bytes para entregar. Cuando `read()` retorna una cadena de bytes vacía, la iteración finalizará y no se podrá reiniciar.

Si *filelike* tiene un método `close()`, el objeto retornado también tendrá un método `close()` que llamará al método `close()` del objeto archivo subyacente.

Ejemplo de uso:

```

from io 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)

for chunk in wrapper:
    print(chunk)

```

Obsoleto desde la versión 3.8: El soporte de `sequence` protocol es obsoleto.

21.4.2 `wsgiref.headers` – Herramientas para cabeceras de respuesta WSGI

Este módulo ofrece una sola clase, `Headers`, para la manipulación de cabeceras de respuesta WSGI usando un interfaz de mapa.

class `wsgiref.headers.Headers([headers])`

Crea un objeto con interfaz de mapa envolviendo *headers*, que debe ser una lista de tuplas nombre/valor de las cabeceras, como se describe en [PEP 3333](#). El valor por defecto de *headers* es una lista vacía.

Los objetos `Headers` soportan las operaciones de mapa habituales incluyendo `__getitem__()`, `get()`, `__setitem__()`, `setdefault()`, `__delitem__()` y `__contains__()`. Para cada uno de esos métodos, la clave es el nombre de la cabecera, sin distinción entre mayúsculas y minúsculas, y el valor es el primer valor asociado con el nombre de la cabecera. Establecer una cabecera borra cualquier valor previamente existente y añade un nuevo valor al final de la lista de cabeceras. En general, el orden de las cabeceras existentes se mantiene, con las nuevas cabeceras añadidas al final de la lista de cabeceras.

A diferencia de un diccionario, los objetos `Headers` no lanzan un error cuando se intenta obtener o eliminar una clave que no está en la lista de cabeceras subyacente. Al intentar obtener una clave inexistente simplemente

se retornará `None`, mientras que el intento de eliminación de una cabecera inexistente simplemente no tendrá ningún efecto.

Los objetos `Headers` también soportan los métodos `keys()`, `values()`, y `items()`. Las listas retornadas por `keys()` y `items()` pueden incluir la misma clave más de una vez si se trata de una cabecera de valor múltiple. La `len()` de un objeto `Headers` es la misma que la longitud de sus `items()`, que es la misma que la longitud de la lista de cabeceras envuelta. De hecho, el método `items()` simplemente retorna una copia de la lista de cabeceras.

La llamada `bytes()` sobre un objeto `Headers` retorna una cadena de bytes formateada y lista para su transmisión como cabeceras de respuesta HTTP. Cada cabecera se ubica en una línea con su valor separado por dos puntos y un espacio. Cada línea finaliza con un retorno de carro y un salto de línea, y la cadena de bytes finaliza con una línea en blanco.

Además de la interfaz de mapa y las funcionalidades de formateado, los objetos `Headers` también ofrecen los siguientes métodos para consultar y añadir cabeceras con múltiples valores, y para añadir cabeceras con parámetros MIME:

`get_all(name)`

Retorna una lista de todos los valores para la cabecera indicada.

La lista retornada tendrá los valores ordenados según la lista de cabeceras original o según se hayan añadido a esta instancia, y podrá contener duplicados. Cualquier campo eliminado y añadido de nuevo estará al final de la lista. Si no existen campos con el nombre indicado, retornará una lista vacía.

`add_header(name, value, *_params)`

Añade una cabecera, posiblemente de valor múltiple, con los parámetros MIME opcionales especificados vía argumentos por palabra clave.

`name` es el nombre de la cabecera a añadir. Se pueden usar argumentos por palabra clave para establecer parámetros MIME para la cabecera. Cada parámetro debe ser una cadena o `None`. Todos los guiones bajos en los nombres de parámetros se convierten en guiones, dado que los guiones son inválidos en identificadores Python y muchos parámetros MIME incluyen guiones. Si el valor del parámetro es una cadena, se añade a los parámetros del valor de la cabecera con la forma `nombre=valor`. Si es `None`, sólo se añade el nombre del parámetro, para reflejar parámetros MIME sin valor. Ejemplo de uso:

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

El código anterior añadirá una cabecera como la siguiente:

```
Content-Disposition: attachment; filename="bud.gif"
```

Distinto en la versión 3.5: El parámetro `headers` es opcional.

21.4.3 `wsgiref.simple_server`– Un servidor HTTP WSGI simple

Este módulo implementa un servidor HTTP simple, basado en `http.server`, que sirve aplicaciones WSGI. Cada instancia del servidor sirve una aplicación WSGI simple en una máquina y puerto dados. Si se quiere servir múltiples aplicaciones en una misma máquina y puerto, se deberá crear una aplicación WSGI que analiza `PATH_INFO` para seleccionar qué aplicación invocar para cada petición. Por ejemplo, usando la función `shift_path_info()` de `wsgiref.util`.

`wsgiref.simple_server.make_server(host, port, app, server_class=WSGIServer, handler_class=WSGIRequestHandler)`

Crea un nuevo servidor WSGI que sirve en `host` y `port`, aceptando conexiones para `app`. El valor de retorno es una instancia de `server_class` y procesará peticiones usando `handler_class`. `app` debe ser un objeto aplicación WSGI, como se define en [PEP 3333](#).

Ejemplo de uso:

```

from wsgiref.simple_server import make_server, demo_app

with make_server(' ', 8000, demo_app) as httpd:
    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*)

Esta función es una pequeña pero completa aplicación WSGI que retorna una página de texto con el mensaje «Hello world!» y una lista de pares clave/valor obtenida del parámetro *environ*. Es útil para verificar que un servidor WSGI, como `wsgiref.simple_server`, es capaz de ejecutar una aplicación WSGI simple correctamente.

class `wsgiref.simple_server.WSGIServer` (*server_address*, *RequestHandlerClass*)

Crea una instancia de `WSGIServer`. *server_address* debe ser una tupla (máquina,puerto) y *RequestHandlerClass* debe ser la subclase de `http.server.BaseHTTPRequestHandler` que se usará para procesar peticiones.

Normalmente, no es necesario invocar este constructor, ya que la función `make_server()` puede gestionar todos los detalles.

`WSGIServer` es una subclase de `http.server.HTTPServer`, por lo que todos sus métodos, como `serve_forever()` y `handle_request()`, están disponibles. `WSGIServer` también ofrece los siguientes métodos específicos de WSGI:

set_app (*application*)

Establece el invocable *application* como la aplicación WSGI que recibirá las peticiones.

get_app ()

Retorna la aplicación invocable actual.

Habitualmente, sin embargo, no es necesario usar estos métodos adicionales, ya que `set_app()` se llama desde `make_server()` y `get_app()` existe sobretodo para el beneficio de instancias del gestor de peticiones.

class `wsgiref.simple_server.WSGIRequestHandler` (*request*, *client_address*, *server*)

Crea un gestor HTTP para la *request* indicada (es decir un socket), *client_address* (una tupla "(máquina,puerto)"), y **server* (una instancia `WSGIServer`).

No es necesario crear instancias de esta clase directamente. Se crean automáticamente bajo demanda por objetos `WSGIServer`. Sin embargo, se pueden crear subclases de esta clase y proveerlas como *handler_class* a la función `make_server()`. Algunos métodos posiblemente relevantes para sobrescribir en estas subclases:

get_environ ()

Retorna un diccionario con el entorno WSGI para una petición. La implementación por defecto copia el contenido del diccionario atributo `base_environ` del objeto `WSGIServer` y añade varias cabeceras derivadas de la petición HTTP. Cada llamada a este método debe retornar un nuevo diccionario con todas las variables de entorno CGI relevante especificadas en [PEP 3333](#).

get_stderr ()

Retorna el objeto que debe usarse como el flujo de `wsgi.errors`. La implementación por defecto retorna simplemente `sys.stderr`.

handle ()

Procesa la petición HTTP. La implementación por defecto crea una instancia gestora usando una clase `wsgiref.handlers` para implementar la interfaz de aplicación WSGI real.

21.4.4 `wsgiref.validate` — Verificador de compatibilidad WSGI

Al crear nuevos objetos aplicación WSGI, *frameworks*, servidores, o *middleware*, puede ser útil validar la compatibilidad del nuevo código usando `wsgiref.validate`. Este módulo ofrece una función que crea objetos de aplicación WSGI que validan las comunicaciones entre un servidor o *gateway* WSGI y un objeto de aplicación WSGI, para comprobar la compatibilidad del protocolo en ambos lados.

Hay que observar que esta utilidad no garantiza compatibilidad completa con [PEP 3333](#). La ausencia de errores usando este módulo no implica que no existan errores. Sin embargo, si este módulo produce errores, implica que o el servidor o la aplicación no son 100% compatibles.

Este módulo se basa en el módulo `paste.lint` de la librería *Python Paste* de *Ian Bicking*.

`wsgiref.validate.validator(application)`

Envuelve *application* y retorna un nuevo objeto de aplicación WSGI. La aplicación retornada reenviará todas las peticiones a la *application* original y comprobará que tanto la *application* como el servidor que la llama son compatibles con la especificación WSGI y con [RFC 2616](#).

Cualquier incompatibilidad detectada provocará el lanzamiento de un `AssertionError`. Nótese que, sin embargo, cómo se gestionan estos errores depende del servidor. Por ejemplo, `wsgiref.simple_server` y otros servidores basados en `wsgiref.handlers`, que no sobrescriben los métodos de gestión de errores para hacer otras cosas, simplemente escribirán un mensaje de que el error ha ocurrido y volcarán la traza de error en `sys.stderr` o algún otro flujo de errores.

Este *wrapper* puede también generar salidas usando el módulo `warnings` para señalar comportamientos que son cuestionables pero que no están realmente prohibidos por [PEP 3333](#). A no ser que se suprima esta salida usando opciones de línea de comandos de Python o la API de `warnings`, cualquier aviso se escribirá en `sys.stderr`, en lugar de en `wsgi.errors`, aunque sean el mismo objeto.

Ejemplo de uso:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# 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 b"Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000...")
    httpd.serve_forever()
```

21.4.5 wsgiref.handlers - Clases base servidor/gateway

Este módulo ofrece clases gestoras base para implementar servidores y *gateways* WSGI. Estas clases base gestionan la mayoría del trabajo de comunicarse con una aplicación WSGI, siempre que se les dé un entorno CGI, junto con una entrada, una salida, y un flujo de errores.

class wsgiref.handlers.CGIHandler

Invocación basada en CGI vía `sys.stdin`, `sys.stdout`, `sys.stderr` y `os.environ`. Esto es útil cuando se quiere ejecutar una aplicación WSGI como un script CGI. Simplemente es necesario invocar `CGIHandler().run(app)`, donde `app` es el objeto de aplicación WSGI que se quiere invocar.

Esta clase es una subclase de `BaseCGIHandler` que establece `wsgi.run_once` a cierto, `wsgi.multithread` a falso, y `wsgi.multiprocess` a cierto, y siempre usa `sys` y `os` para obtener los flujos y entorno CGI necesarios.

class wsgiref.handlers.IISCGIHandler

Una alternativa especializada a `CGIHandler`, para usar al desplegar en servidores web Microsoft IIS, sin establecer la opción de configuración `allowPathInfo` (IIS>=7) o la meta-base `allowPathInfoForScriptMappings` (IIS<7).

Por defecto, IIS entrega `PATH_INFO` que duplica `SCRIPT_NAME` al principio, causando problemas con aplicaciones WSGI que implementan enrutamiento. Este gestor elimina las partes duplicadas de la ruta.

IIS can be configured to pass the correct `PATH_INFO`, but this causes another bug where `PATH_TRANSLATED` is wrong. Luckily this variable is rarely used and is not guaranteed by WSGI. On IIS<7, though, the setting can only be made on a vhost level, affecting all other script mappings, many of which break when exposed to the `PATH_TRANSLATED` bug. For this reason IIS<7 is almost never deployed with the fix (Even IIS7 rarely uses it because there is still no UI for it.).

No hay forma de que el código CGI detecte cuándo la opción está activada, por lo que se ofrece una clase gestora separada. Se usa de la misma forma que `CGIHandler`, p.e. llamando a `IISCGIHandler().run(app)`, donde `app` es el objeto aplicación WSGI que se desea invocar.

Nuevo en la versión 3.2.

class wsgiref.handlers.BaseCGIHandler (*stdin, stdout, stderr, environ, multithread=True, multiprocess=False*)

Similar a `CGIHandler`, pero en lugar de usar los módulos `sys` y `os`, el entorno CGI y los flujos de E/S se especifican explícitamente. Los valores `multithread` y `multiprocess` se pasan a las aplicaciones ejecutadas por la instancia de gestión como `wsgi.multithread` y `wsgi.multiprocess`.

Esta clase es una subclase de `SimpleHandler` para usarse con servidores HTTP que no reciben peticiones directas de Internet, *HTTP origin servers*. Al escribir una implementación de un protocolo de pasarela, como CGI, FastCGI, SCGI, etcétera, que use una cabecera `Status:` para enviar un estado HTTP, probablemente sea adecuado heredar de esta clase, en lugar de `SimpleHandler`.

class wsgiref.handlers.SimpleHandler (*stdin, stdout, stderr, environ, multithread=True, multiprocess=False*)

Similar a `BaseCGIHandler` pero diseñada para usarse con servidores que reciban peticiones directamente de Internet, o *HTTP origin servers*. Al escribir una implementación de un servidor HTTP, probablemente prefiera heredar de esta clase en lugar de `BaseCGIHandler`.

Esta clase es una subclase de `BaseHandler`. Sobreescribe los métodos `__init__()`, `get_stdin()`, `get_stderr()`, `add_cgi_vars()`, `_write()` y `_flush()` para soportar el uso explícito del entorno y los flujos a partir del constructor. El entorno y los flujos especificados se almacenan en los atributos `stdin`, `stdout`, `stderr` y `environ`.

El método `write()` de `stdout` podría escribir cada fragmento de información completamente, como `io.BufferedReader`.

class wsgiref.handlers.BaseHandler

Esta es una clase base abstracta para ejecutar aplicaciones WSGI. Cada instancia gestionará una sola petición HTTP, aunque en principio se podría crear una subclase reutilizable para múltiples peticiones.

Las instancias de `BaseHandler` tiene un sólo método para uso externo:

run (*app*)

Ejecuta la aplicación WSGI *app* indicada.

Todos los otros métodos de *BaseHandler* se invocan por este método en el proceso de ejecutar la aplicación, es decir, existen principalmente para permitir personalizar el proceso.

Los métodos siguientes DEBEN sobrescribirse en una subclase:

_write (*data*)

Enviar los bytes *data* para la transmisión al cliente. Es correcto que este método realmente transmita la información. La clase *BaseHandler* simplemente separa las operaciones de escritura y liberación para una mayor eficiencia cuando el sistema subyacente realmente hace esa distinción.

_flush ()

Forzar la transmisión de los datos en el búfer al cliente. Es correcto si este método no es operativo, p.e., si *_write* () realmente envía los datos.

get_stdin ()

Retorna un objeto de flujo de entrada disponible para usar como *wsgi.input* de la petición en proceso actualmente.

get_stderr ()

Retorna un objeto de flujo de salida disponible para usar como *wsgi.errors* de la petición en proceso actualmente.

add_cgi_vars ()

Inserta las variables CGI para la petición actual en el atributo *environ*.

A continuación se describen algunos métodos y atributos más que podría ser interesante sobrescribir. Esta lista es sólo un sumario, sin embargo, y no incluye cada método que puede ser sobrescrito. Conviene consultar las docstrings y el código fuente para obtener información adicional antes de intentar crear una subclase de *BaseHandler* personalizada.

Atributos y métodos para personalizar el entorno WSGI:

wsgi_multithread

El valor a usar en la variable de entorno *wsgi.multithread*. Por defecto es cierto en *BaseHandler*, pero puede tener un valor por defecto distinto (o establecerse en el constructor) en otras subclases.

wsgi_multiprocess

El valor a usar en la variable de entorno *wsgi.multiprocess*. Por defecto es cierto en *BaseHandler*, pero puede tener un valor por defecto distinto (o establecerse en el constructor) en otras subclases.

wsgi_run_once

El valor a usar en la variable de entorno *wsgi.run_once*. Por defecto es falso en *BaseHandler*, pero en *CGIHandler* es cierto por defecto.

os_environ

Las variables de entorno por defecto que se incluirán en el entorno WSGI de cada petición. Por defecto, es una copia de *os.environ* cuando se importa *wsgiref.handlers*, pero otras subclases pueden crear las suyas propias a nivel de clase o de instancia. Se debe tener en cuenta que el diccionario se debe considerar como de sólo lectura, ya que el valor por defecto se comparte entre múltiples clases e instancias.

server_software

Si el atributo *origin_server* tiene valor, éste se usa para establecer el valor por defecto de la variable de entorno WSGI *SERVER_SOFTWARE* y un encabezado *Server*: por defecto en las respuestas HTTP. Las clases gestoras que no son *HTTP origin servers*, como *BaseCGIHandler* y *CGIHandler*, ignoran este atributo.

Distinto en la versión 3.3: El término «Python» se reemplaza con el término específico correspondiente a la implementación del intérprete, como «CPython», «Jython», etc.

get_scheme()

Retorna el esquema usado en la URL para la petición actual. La implementación por defecto utiliza la función `guess_scheme()` de `wsgiref.util` para adivinar si el esquema debería ser «http» o «https», basándose en las variables de `environ` de la petición actual.

setup_environ()

Establece el atributo `environ` a un entorno WSGI completo. La implementación por defecto utiliza todos los métodos y atributos anteriormente mencionados, más los métodos `get_stdin()`, `get_stderr()` y `add_cgi_vars()`, y el atributo `wsgi_file_wrapper`. También incluye una clave `SERVER_SOFTWARE` si no existe, siempre y cuando el atributo `origin_server` tiene un valor válido y el atributo `server_software` está establecido.

Métodos y atributos para personalizar el manejo de excepciones:

log_exception(exc_info)

Envía la tupla `exc_info` al registro del servidor. `exc_info` es un tupla (`type`, `value`, `traceback`). La implementación por defecto simplemente escribe el seguimiento de la pila en el flujo `wsgi.errors` de la petición y lo vacía. Las subclases pueden sobrescribir éste método para cambiar el formato o redirigir la salida, enviar mensajes de correo con el seguimiento de pila a un administrador o cualquier otra acción que se considere adecuada.

traceback_limit

El máximo número de marcos a incluir en la salida de seguimientos de pilas por el método por defecto `log_exception()`. Si vale `None`, se incluyen todos los marcos.

error_output(environ, start_response)

Este método es una aplicación WSGI para generar una página de error para el usuario. Sólo se invoca si un error ocurre antes de enviar las cabeceras al cliente.

Este método puede acceder a la información de error actual usando `sys.exc_info()` y debería pasar esa información a `start_response` cuando es llamada, tal y como se describe en la sección «Error Handling» de [PEP 3333](#).

La implementación por defecto sólo utiliza los atributos `error_status`, `error_headers` y `error_body` para generar una página de salida. Las subclases pueden sobrescribir éste para producir una salida de error dinámica mejor.

Hay que tener en cuenta, sin embargo, que no se recomienda, desde una perspectiva de seguridad, mostrar información de diagnóstico a cualquier usuario antiguo. Idealmente, se debería hacer algo especial para activar la salida de información de diagnóstico, motivo por el que la implementación por defecto no incluye ninguna.

error_status

El estado HTTP utilizado para las respuestas de error. Se debería utilizar una de las cadenas de estado definidas en [PEP 3333](#). Por defecto es un código 500 y un mensaje.

error_headers

Las cabeceras HTTP utilizadas por las respuestas de error. Debería tratarse de una lista de cabeceras de respuesta WSGI (tuplas (`name`, `value`)), tal y como se describe en [PEP 3333](#). La lista por defecto simplemente establece el tipo de contenido a `text/plain`.

error_body

El cuerpo de la respuesta de error. Debería ser una cadena de bytes con el cuerpo de la respuesta HTTP. Por defecto contiene el texto plano *A server error occurred. Please contact the administrator.*

Métodos y atributos para la funcionalidad «Optional Platform-Specific File Handling» de [PEP 3333](#):

wsgi_file_wrapper

Una factoría `wsgi.file_wrapper`, o `None`. El valor por defecto de este atributo es la clase `wsgiref.util.FileWrapper`.

sendfile()

Sobrescribir para implementar la transmisión de ficheros específica para la plataforma. Este método se llama sólo si el valor de retorno de la aplicación es una instancia de la clase especificada por el atributo `wsgi_file_wrapper`. Debería retornar un valor cierto si fue capaz de transmitir correctamente el

fichero, de modo que el código por defecto de transmisión no será ejecutado. La implementación por defecto de este método simplemente retorna un valor falso.

Métodos y atributos varios:

origin_server

Este atributo debería establecerse a cierto si los métodos `_write()` y `_flush()` están siendo usados para comunicar directamente al cliente, en lugar de usar un protocolo de pasarela CGI que requiere el estado HTTP en una cabecera `Status:` especial.

El valor por defecto de este atributo es cierto en `BaseHandler`, pero en `BaseCGIHandler` y `CGIHandler` es falso.

http_version

Si `origin_server` es cierto, este atributo de tipo cadena se usa para establecer la versión HTTP de la respuesta enviada al cliente. Por defecto es `"1.0"`.

`wsgiref.handlers.read_environ()`

Transcodifica las variables CGI de `os.environ` a cadenas «bytes in unicode» definidas en [PEP 3333](#), retornando un nuevo diccionario. Esta función se usa en `CGIHandler` y `IISCGIHandler` en lugar de usar directamente `os.environ`, lo que no es necesariamente compatible con EWGI en todas las plataformas y servidores web usando Python 3 – específicamente, aquellas en las que el entorno actual del sistema operativo es Unicode, p.e. Windows, o en las que el entorno está en bytes, pero la codificación del sistema usada por Python para descodificar lo es cualquier otro que ISO-8859-1, por ejemplo, sistemas UNIX que usen UTF-8.

Cuando se está implementando un gestor basado en CGI propio, probablemente se requiera usar esta rutina en lugar de sólo copiar directamente los valores de `os.environ`.

Nuevo en la versión 3.2.

21.4.6 Ejemplos

Ésta es una aplicación WSGI «Hello World» que funciona:

```
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.
def hello_world_app(environ, start_response):
    status = '200 OK' # HTTP Status
    headers = [('Content-type', 'text/plain; charset=utf-8')] # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return [b"Hello World"]

with make_server('', 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # Serve until process is killed
    httpd.serve_forever()
```

Ejemplo de una aplicación WSGI que sirve el directorio actual, acepta un directorio opcional y un número de puerto (default: 8000) en la línea de comandos:

```
#!/usr/bin/env python3
'''
Small wsgiref based web server. Takes a path to serve from and an
```

(continué en la próxima página)

(proviene de la página anterior)

```

optional port number (defaults to 8000), then tries to serve files.
Mime types are guessed from the file names, 404 errors are raised
if the file is not found. Used for the make serve target in Doc.
'''
import sys
import os
import mimetypes
from wsgiref import simple_server, util

def app(environ, respond):

    fn = os.path.join(path, environ['PATH_INFO'][1:])
    if '.' not in fn.split(os.path.sep)[-1]:
        fn = os.path.join(fn, 'index.html')
    type = mimetypes.guess_type(fn)[0]

    if os.path.exists(fn):
        respond('200 OK', [('Content-Type', type)])
        return util.FileWrapper(open(fn, "rb"))
    else:
        respond('404 Not Found', [('Content-Type', 'text/plain')])
        return [b'not found']

if __name__ == '__main__':
    path = sys.argv[1] if len(sys.argv) > 1 else os.getcwd()
    port = int(sys.argv[2]) if len(sys.argv) > 2 else 8000
    httpd = simple_server.make_server('', port, app)
    print("Serving {} on port {}, control-C to stop".format(path, port))
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        print("Shutting down.")
        httpd.server_close()

```

21.5 urllib — URL módulos de manipulación

Código fuente: [Lib/urllib/](#)

`urllib` es un paquete que reúne varios módulos para trabajar con URLs:

- `urllib.request` para abrir y leer URLs
- `urllib.error` contiene las excepciones propuestas por `urllib.request`
- `urllib.parse` para parsear URLs
- `urllib.robotparser` para parsear “robots.txt” archivos

21.6 urllib.request — Biblioteca extensible para abrir URLs

Código fuente: [Lib/urllib/request.py](#)

El módulo `urllib.request` define funciones y clases que ayudan en la apertura de URLs (la mayoría HTTP) en un mundo complejo — autenticación básica y digest, redirecciones, cookies y más.

Ver también:

Se recomienda el paquete [Requests](#) para una interfaz de cliente HTTP de mayor nivel.

El módulo `urllib.request` define las siguientes funciones:

`urllib.request.urlopen(url, data=None[, timeout], *, cafile=None, capath=None, cadefault=False, context=None)`

Abre la URL `url`, la cual puede ser una cadena de caracteres o un objeto [Request](#).

`data` debe ser un objeto que especifique datos adicionales a ser enviados al servidor o `None` si no se necesitan tales datos. Vea [Request](#) para más detalles.

El módulo `urllib.request` usa HTTP/1.1 e incluye el encabezado `Connection:close` en sus peticiones HTTP.

El parámetro opcional `timeout` especifica un tiempo de expiración en segundos para operaciones bloqueantes como el intento de conexión (si no se especifica, será usado el tiempo de expiración global predeterminado). Esto actualmente sólo funciona para conexiones HTTP, HTTPS y FTP.

Si se especifica `context`, debe ser una instancia [ssl.SSLContext](#) describiendo las diferentes opciones SSL. Vea [HTTPSConnection](#) para más detalles.

Los parámetros opcionales `cafile` y `capath` especifican un conjunto de certificados CA de confianza para peticiones HTTPS. `cafile` debe apuntar a un único archivo que contenga un paquete de certificados CA, mientras `capath` debe apuntar a un directorio de archivos de certificado hash. Se puede encontrar más información en [ssl.SSLContext.load_verify_locations\(\)](#).

Se ignora el parámetro `cadefault`.

Esta función siempre retorna un objeto que puede funcionar como un [context manager](#) y tiene métodos como

- `geturl()` — retorna la URL del recurso obtenido, comúnmente usado para determinar si se ha seguido una redirección
- `info()` — retorna la meta información de la página, como los encabezados, en la forma de una instancia [email.message_from_string\(\)](#) (vea [Quick Reference to HTTP Headers](#))
- `getcode()` — retorna el código de estado HTTP de la respuesta.

Para URLs HTTP y HTTPS, esta función retorna un objeto [http.client.HTTPResponse](#) ligeramente modificado. Adicionalmente a los tres nuevos métodos anteriores, el atributo `msg` contiene la misma información que el atributo `reason` — la frase de motivo devuelta por el servidor — en lugar de los encabezados de la respuesta como se especifica en la documentación para [HTTPResponse](#).

Para URLs FTP, de archivo y de datos y para peticiones manejadas explícitamente por las clases heredadas [URLopener](#) y [FancyURLopener](#), esta función retorna un objeto `urllib.response.addinfourl`.

Genera [URLError](#) en errores de protocolo.

Tenga en cuenta que `None` puede ser retornado si ningún manejador gestiona la petición (aunque el [OpenerDirector](#) global instalado de manera predeterminada usa [UnknownHandler](#) para asegurar que esto nunca suceda).

Adicionalmente, si se detectan configuraciones de proxy (por ejemplo, cuando se establece una variable de entorno `*_proxy` como `http_proxy`), [ProxyHandler](#) está instalada de forma predeterminada y se asegura que las peticiones son gestionadas a través del proxy.

La función heredada de Python 2.6 y anteriores `urllib.urlopen` ha sido descontinuada, [urllib.request.urlopen\(\)](#) corresponde a la antigua `urllib2.urlopen`. La gestión de proxy, la cual

se hacía pasando un parámetro diccionario a `urllib.urlopen`, puede ser obtenida usando objetos *ProxyHandler*.

Genera un *evento de auditoría* `urllib.Request` con los argumentos `fullurl`, `data`, `headers`, `method`.

Distinto en la versión 3.2: *cafile* y *capath* fueron añadidos.

Distinto en la versión 3.2: Los hosts virtuales HTTPS ahora están soportados si es posible (esto es, si *ssl.SSLContext* es verdadero).

Nuevo en la versión 3.2: *data* puede ser un objeto iterable.

Distinto en la versión 3.3: *cadefault* fue añadido.

Distinto en la versión 3.4.3: *context* fue añadido.

Obsoleto desde la versión 3.6: *cafile*, *capath* y *cadefault* están obsoletos en favor de *context*. Por favor, use *ssl.SSLContext.load_cert_chain()* en su lugar o deja a *ssl.create_default_context()* seleccionar el certificado de confianza CA del sistema por ti.

`urllib.request.install_opener(opener)`

Instala una instancia *OpenerDirector* como el abridor global predeterminado. Instalar un abridor sólo es necesario si quieres que `urlopen` use ese abridor; si no, simplemente invoca *OpenerDirector.open()* en lugar de *urlopen()*. El código no comprueba por un *OpenerDirector* real y cualquier clase con la interfaz apropiada funcionará.

`urllib.request.build_opener([handler, ...])`

Retorna una instancia *OpenerDirector*, la cual encadena los manejadores en el orden dado. *handlers* pueden ser tanto instancias de *BaseHandler* o subclases de *BaseHandler* (en cuyo caso debe ser posible invocar el constructor sin ningún parámetro). Instancias de las siguientes clases estarán delante del *handlers*, a no ser que el *handlers* las contenga, instancias o subclases de ellas: *ProxyHandler* (si son detectadas configuraciones de proxy), *UnknownHandler*, *HTTPHandler*, *HTTPDefaultErrorHandler*, *HTTPRedirectHandler*, *FTPHandler*, *FileHandler*, *HTTPErrorProcessor*.

Si la instalación de Python tiene soporte SSL (ej. si se puede importar el módulo *ssl*), también será añadida *HTTPSHandler*.

Una subclase de *BaseHandler* puede cambiar también su atributo `handler_order` para modificar su posición en la lista de manejadores.

`urllib.request.pathname2url(path)`

Convierte el nombre de ruta *path* desde la sintaxis local para una ruta a la forma usada en el componente ruta de una URL. Esto no produce una URL completa. El valor retornado ya estará entrecomillado usando la función *quote()*.

`urllib.request.url2pathname(path)`

Convierte el componente ruta *path* desde una URL codificada con porcentajes a la sintaxis local para una ruta. No acepta una URL completa. Esta función usa *unquote()* para decodificar *path*.

`urllib.request.getproxies()`

Esta función auxiliar devuelve un diccionario de esquema para las asignaciones de URL del servidor proxy. Escanea el entorno en busca de variables denominadas `<scheme>_proxy`, tomando en cuenta diferencia entre mayúsculas y minúsculas, para todos los sistemas operativos primero, y cuando no pueden encontrarla, buscan información de proxy desde la Configuración del Sistema Mac OSX para Mac OS X y desde Registros del Sistema para Windows. Si existen variables de entorno tanto en mayúsculas como en minúsculas (y no concuerdan), las minúsculas son preferidas.

Nota: Si la variable del entorno `REQUEST_METHOD` está definida, lo cual usualmente indica que tu script está ejecutándose en un entorno CGI, la variable de entorno `HTTP_PROXY` (mayúsculas `_PROXY`) será ignorada. Esto es porque esa variable puede ser inyectada por un cliente usando el encabezado HTTP «Proxy:». Si necesitas usar un proxy HTTP en un entorno CGI, usa *ProxyHandler* explícitamente o asegúrate de que el nombre de la variable está en minúsculas (o al menos el sufijo `_proxy`).

Se proveen las siguientes clases:

```
class urllib.request.Request (url, data=None, headers={}, origin_req_host=None, unverifiable=False, method=None)
```

Esta clase es un abstracción de una petición URL.

url debe ser una cadena de caracteres conteniendo una URL válida.

data debe ser un objeto que especifique datos adicionales a enviar al servidor o `None` si no se necesitan tales datos. Actualmente las peticiones HTTP son las únicas que usan *data*. Los tipos de objetos soportados incluyen bytes, objetos como archivos e iterables de objetos como bytes. Si no se ha provisto el campo de encabezado `Content-Length` ni `Transfer-Encoding`, `HTTPHandler` establecerá estos encabezados de acuerdo al tipo de *data*. `Content-Length` será usado para enviar objetos de bytes, mientras `Transfer-Encoding: chunked` como se especifica en [RFC 7230](#), Sección 3.3.1 será usado para enviar archivos y otros iterables.

Para un método de una petición HTTP POST, *data* debe ser un buffer en el formato estándar `application/x-www-form-urlencoded`. La función `urllib.parse.urlencode()` toma un mapeo o una secuencia de tuplas de dos valores y retorna una cadena de caracteres ASCII en este formato. Debe ser codificada a bytes antes de ser usada como el parámetro *data*.

headers debe ser un diccionario, y será tratado como si `add_header()` fuera invocado con cada clave y valor como argumentos. Esto es usado frecuentemente para «parodiar» el valor de encabezado `User-Agent`, el cual es usado por un navegador para identificarse a sí mismo – algunos servidores HTTP sólo permiten peticiones que vienen de navegadores comunes a diferencia de los scripts. Por ejemplo, Mozilla Firefox puede identificarse a sí mismo como "Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11", mientras el agente de usuario predeterminado de `urllib` es "Python-urllib/2.6" (en Python 2.6).

Un encabezado apropiado `Content-Type` debe ser incluido si el argumento *data* está presente. Si este encabezado no ha sido provisto y *data* no es `None`, será añadido `Content-Type: application/x-www-form-urlencoded` de forma predeterminada.

Los siguientes dos argumentos sólo tienen interés para la gestión correcta de cookies HTTP de terceros:

origin_req_host debe ser el host de la petición de la transacción origen, como define [RFC 2965](#). Por defecto es `http.cookiejar.request_host(self)`. Este es el nombre de host o la dirección IP de la petición original que fue iniciada por el usuario. Por ejemplo, si la petición es para una imagen en un documento HTML, debe ser el host de la petición para la página que contiene la imagen.

unverifiable debe indicar si la petición no es verificable, como define [RFC 2965](#). Por defecto es `False`. Una petición no verificable es una cuya URL el usuario no tuvo opción de aprobar. Por ejemplo, si la petición es por una imagen en un documento HTML y el usuario no tuvo opción de aprobar la obtención automática de la imagen, este debe ser verdadero.

method debe ser una cadena que indica el método de la petición HTTP que será usado (ej. `'HEAD'`). Si se provee, su valor es almacenado en el atributo `method` y usado por `get_method()`. Por defecto es `'GET'` si *data* es `None`, o `'POST'` si no. Las subclases pueden indicar un método predeterminado diferente estableciendo el atributo `method` es la clase misma.

Nota: La petición no funcionará como se espera si el objeto de datos es incapaz de entregar su contenido más de una vez (ej. un archivo o un iterable que puede producir el contenido sólo una vez) y la petición se reintentará para redirecciones HTTP o autenticación. El *data* es enviado al servidor HTTP directamente después de los encabezados. No hay soporte para una expectativa de funcionamiento 100% continuo en la biblioteca.

Distinto en la versión 3.3: El argumento `Request.method` es añadido a la clase `Request`.

Distinto en la versión 3.4: El atributo predeterminado `Request.method` puede ser indicado a nivel de clase.

Distinto en la versión 3.6: No se genera un error si el `Content-Length` no ha sido provisto y *data* no es `None` ni un objeto de bytes. En su lugar recurre a la codificación de transferencia fragmentada.

class urllib.request.OpenerDirector

La clase *OpenerDirector* abre URLs mediante la encadenación conjunta de *BaseHandler*. Este maneja el encadenamiento de manejadores y la recuperación de errores.

class urllib.request.BaseHandler

Esta es la clase base para todos los manejadores registrados — y manejan sólo las mecánicas simples del registro.

class urllib.request.HTTPDefaultErrorHandler

Una clase la cual define un manejador predeterminado para los errores de respuesta HTTP; todas las respuestas son convertidas en excepciones *HTTPError*.

class urllib.request.HTTPRedirectHandler

Una clase para manejar redirecciones.

class urllib.request.HTTPCookieProcessor (*cookiejar=None*)

Una clase para manejar Cookies HTTP.

class urllib.request.ProxyHandler (*proxies=None*)

Causa que las peticiones vayan a través de un proxy. Si se provee *proxies*, debe ser un diccionario mapeando nombres de protocolos a URLs de proxies. Por defecto lee la lista de proxies de las variables de entorno `<protocol>_proxy`. Si no se establecen variables de entorno de proxy, entonces se obtienen las configuraciones de proxy en un entorno Windows desde la sección del registro de Configuraciones de Internet y en un entorno Mac OS X se obtiene la información de proxy desde el Framework de Configuración del Sistema de OS X.

Para deshabilitar la detección automática de proxy pasa un diccionario vacío.

La variable de entorno `no_proxy` puede ser usada para especificar hosts los cuales no deben ser alcanzados mediante proxy; si se establece, debe ser una lista separada por comas de sufijos de nombres de host, con `:port` añadidos opcionalmente, por ejemplo `cern.ch,ncsa.uiuc.edu,some.host:8080`.

Nota: HTTP_PROXY será ignorado si se establece una variable REQUEST_METHOD; vea la documentación de *getproxies()*.

class urllib.request.HTTPPasswordMgr

Mantiene una base de datos de mapeos (*realm, uri*) -> (*user, password*).

class urllib.request.HTTPPasswordMgrWithDefaultRealm

Mantiene una base de datos de mapeos (*realm, uri*) -> (*user, password*). Un reino de None se considera un reino caza todo, el cual es buscado si ningún otro reino encaja.

class urllib.request.HTTPPasswordMgrWithPriorAuth

Una variante de *HTTPPasswordMgrWithDefaultRealm* que también tiene una base de datos de mapeos *uri* -> *is_authenticated*. Puede ser usada por un manejador BasicAuth para determinar cuando enviar credenciales de autenticación inmediatamente en lugar de esperar primero a una respuesta 401.

Nuevo en la versión 3.5.

class urllib.request.AbstractBasicAuthHandler (*password_mgr=None*)

Esta es una clase mixin que ayuda con la autenticación HTTP, tanto al host remoto y a un proxy. Si se proporciona *password_mgr*, debe ser algo compatible con *HTTPPasswordMgr*; refiera a la sección *Objetos HTTPPasswordMgr* para información sobre la interfaz que debe ser soportada. Si *password_mgr* proporciona también métodos *is_authenticated* y *update_authenticated* (vea *Objetos HTTPPasswordMgrWithPriorAuth*), entonces el manejador usará el *is_authenticated* resultado para una URI dada para determinar el envío o no de credenciales de autenticación con la petición. Si *is_authenticated* retorna True para la URI, las peticiones subsecuentes a la URI o cualquiera de las super URIs incluirán automáticamente los credenciales de autenticación.

Nuevo en la versión 3.5: Añadido soporte *is_authenticated*.

class urllib.request.HTTPBasicAuthHandler (*password_mgr=None*)

Administra autenticación con el host remoto. Si se proporciona *password_mgr*, debe ser compatible con *HTTPPasswordMgr*; refiera a la sección *Objetos HTTPPasswordMgr* para información sobre la interfaz que

debe ser soportada. HTTPBasicAuthHandler generará un *ValueError* cuando se presente con un esquema de Autenticación incorrecto.

class urllib.request.ProxyBasicAuthHandler (password_mgr=None)

Administra autenticación con el proxy. Si se proporciona *password_mgr* debe ser compatible con *HTTPPasswordMgr*; refiera a la sección *Objetos HTTPPasswordMgr* para información sobre la interfaz que debe ser soportada.

class urllib.request.AbstractDigestAuthHandler (password_mgr=None)

Esto es una clase mixin que ayuda con la autenticación HTTP, tanto al host remoto como a un proxy. Si se proporciona *password_mgr* debe ser compatible con *HTTPPasswordMgr*; refiera a la sección *Objetos HTTPPasswordMgr* para información sobre la interfaz que debe ser soportada.

class urllib.request.HTTPDigestAuthHandler (password_mgr=None)

Maneja autenticación con el host remoto. Si se proporciona *password_mgr* debe ser compatible con *HTTPPasswordMgr*; refiera a la sección *Objetos HTTPPasswordMgr* para información sobre la interfaz que debe ser soportada. Cuando se añaden tanto el Manejador de Autenticación Digest (*Digest Authentication Handler*) como el Manejador de Autenticación Básico (*Basic Authentication Handler*) la Autenticación Digest siempre se intenta primero. Si la Autenticación Digest retorna una respuesta 40x de nuevo, se envía al controlador de Autenticación Básica para Manejar. Este método Handler generará un *ValueError* cuando sea presentado con un esquema de autenticación diferente a Digest o Básico.

Distinto en la versión 3.3: Genera *ValueError* en Esquema de Autenticación no soportado.

class urllib.request.ProxyDigestAuthHandler (password_mgr=None)

Administra autenticación con el proxy. Si se proporciona *password_mgr* debe ser compatible con *HTTPPasswordMgr*; refiera a la sección *Objetos HTTPPasswordMgr* para información sobre la interfaz que debe ser soportada.

class urllib.request.HTTPHandler

Una clase para gestionar apertura de URLs HTTP.

class urllib.request.HTTPSHandler (debuglevel=0, context=None, check_hostname=None)

Una clase para gestionar apertura de URLs HTTPS. *context* y *check_hostname* tienen el mismo significado que en *http.client.HTTPSConnection*.

Distinto en la versión 3.2: *context* y *check_hostname* fueron añadidos.

class urllib.request.FileHandler

Abre archivos locales.

class urllib.request.DataHandler

Abre URLs de datos.

Nuevo en la versión 3.4.

class urllib.request.FTPHandler

Abre URLs FTP.

class urllib.request.CacheFTPHandler

Abre URLs FTP, manteniendo una caché de conexiones FTP abiertas para minimizar retrasos.

class urllib.request.UnknownHandler

Una clase caza todo para gestionar URLs desconocidas.

class urllib.request.HTTPErrorProcessor

Procesa errores de respuestas HTTP.

21.6.1 Objetos Request

Los siguientes métodos describen la interfaz pública de *Request* por lo que pueden ser sobrescritos en subclases. También define varios atributos públicos que pueden ser usado por clientes para inspeccionar la respuesta analizada.

`Request.full_url`

La URL original pasada al constructor.

Distinto en la versión 3.4.

`Request.full_url` es una propiedad con setter, getter y deleter. Obtener *full_url* retorna la petición URL original con el fragmento, si este estaba presente.

`Request.type`

El esquema de URI.

`Request.host`

La autoridad de URI, típicamente un host, pero también puede contener un puerto separado por un caracter de doble punto.

`Request.origin_req_host`

El host original de la petición, sin puerto.

`Request.selector`

La ruta de URI. Si *Request* usa un proxy, entonces selector será la URL completa que se pasa al proxy.

`Request.data`

El cuerpo de la entidad para la solicitud o *None* si no es especificado.

Distinto en la versión 3.4: Cambiar el valor de *Request.data* elimina ahora el encabezado «Content-Length» si fue establecido o calculado previamente.

`Request.unverifiable`

booleano, indica si la petición no es verificable como se define por [RFC 2965](#).

`Request.method`

El método de petición HTTP a usar. Por defecto su valor es *None*, lo que significa que *get_method()* realizará su cálculo normal del método a usar. Su valor puede ser definido (sobrescribiendo así el cálculo predeterminado en *get_method()*) tanto proporcionando un valor por defecto estableciéndolo a nivel de clase en una subclase de *Request* o pasando un valor al constructor de *Request* por medio del argumento *method*.

Nuevo en la versión 3.3.

Distinto en la versión 3.4: Un valor predeterminado puede ser establecido ahora en subclases; previamente sólo podía ser definido mediante el argumento del constructor.

`Request.get_method()`

Retorna una cadena indicando el método de petición HTTP. Si *Request.method* no es *None*, retorna su valor, de otra forma retorna 'GET' si *Request.data* es *None* o 'POST' si no lo es. Esto sólo es significativo para peticiones HTTP.

Distinto en la versión 3.3: *get_method* ahora mira el valor de *Request.method*.

`Request.add_header(key, val)`

Añade otro encabezado a la petición. Los encabezados actualmente son ignorados por todos los manejadores excepto los manejadores HTTP, donde son añadidos a la lista de encabezados enviados al servidor. Tenga en cuenta que no puede haber más de un encabezado con el mismo nombre y las invocaciones posteriores sobrescribirán las invocaciones previas en caso de que *key* colisione. Actualmente, esto no es una pérdida de funcionalidad HTTP, ya que todos los encabezados que tienen sentido cuando son usados más de una vez tienen una manera (específica de encabezado) de ganar la misma funcionalidad usando sólo un encabezado.

`Request.add_unredirected_header(key, header)`

Añade un encabezado que no será añadido a una petición redireccionada.

`Request.has_header(header)`

Retorna si la instancia tiene el encabezado nombrado (comprueba tanto regular como no redirigido).

`Request.remove_header(header)`

Elimina el encabezado nombrado de la instancia de la petición (desde encabezados regulares y no redireccionados).

Nuevo en la versión 3.4.

`Request.get_full_url()`

Retorna la URL dada en el constructor.

Distinto en la versión 3.4.

Retorna `Request.full_url`

`Request.set_proxy(host, type)`

Prepara la petición conectando a un servidor proxy. Los *host* y *type* reemplazarán aquellos de la instancia y el selector de la instancia será la URL original dada en el constructor.

`Request.get_header(header_name, default=None)`

Retorna el valor del encabezado dado.

`Request.header_items()`

Retorna una lista de tuplas (*header_name*, *header_value*) de los encabezados de la Petición.

Distinto en la versión 3.4: Los métodos de petición `add_data`, `has_data`, `get_data`, `get_type`, `get_host`, `get_selector`, `get_origin_req_host` y `is_unverifiable` que quedaron obsoletos desde 3.3 han sido eliminados.

21.6.2 Objetos `OpenerDirector`

Las instancias de `OpenerDirector` tienen los siguientes métodos:

`OpenerDirector.add_handler(handler)`

handler debe ser una instancia de `BaseHandler`. Los siguientes métodos son buscados y añadidos a las cadenas posibles (tenga en cuenta que los errores HTTP son un caso especial). Tenga en cuenta que, en los siguientes, *protocol* debe ser reemplazado con el protocolo actual a manejar, por ejemplo `http_response()` sería el protocolo HTTP del manejador de respuesta. También *type* debe ser reemplazado con el código HTTP actual, por ejemplo `http_error_404()` manejaría errores HTTP 404.

- `<protocol>_open()` — señala que el manejador sabe como abrir URLs *protocol*.
Vea `BaseHandler.<protocol>_open()` para más información.
- `http_error_<type>()` — señala que el manejador sabe como manejar errores HTTP con el código de error *type*.
Vea `BaseHandler.http_error_<nnn>()` para más información.
- `<protocol>_error()` — señala que el manejador sabe como manejar errores de (no http) *protocol*.
- `<protocol>_request()` — señala que el manejador sabe como preprocesar peticiones *protocol*.
Vea `BaseHandler.<protocol>_request()` para más información.
- `<protocol>_response()` — señala que el manejador sabe como postprocesar respuestas *protocol*.
Vea `BaseHandler.<protocol>_response()` para más información.

`OpenerDirector.open(url, data=None[, timeout])`

Abre la *url* dada (la cual puede ser un objeto de petición o una cadena de caracteres), pasando opcionalmente el *data* dado. Los argumentos, los valores de retorno y las excepciones generadas son las mismas que aquellas de `urlopen()` (las cuales simplemente invocan el método `open()` en el `OpenerDirector` instalado global). El parámetro opcional *timeout* especifica un tiempo de expiración en segundos para operaciones bloqueantes como el intento de conexión (si no se especifica, el tiempo de expiración global será usado). La característica de tiempo de expiración actualmente funciona sólo para conexiones HTTP, HTTPS y FTP.

`OpenerDirector.error(proto, *args)`

Maneja un error del protocolo dado. Esto invocará los manejadores de error registrados para el protocolo dado con los argumentos dados (los cuales son específicos del protocolo). El protocolo HTTP es un caso especial el cual usa el código de respuesta HTTP para determinar el manejador de error específico; refiera a los métodos `http_error_<type>()` de las clases del manejador.

Retorna si los valores y excepciones generadas son las mismas que aquellas de `urlopen()`.

Los objetos `OpenerDirector` abren URLs en tres etapas:

El orden en el cual esos métodos son invocados dentro de cada etapa es determinado ordenando las instancias manejadoras.

1. Cada manejador con un método nombrado como `<protocol>_request()` tiene ese método invocador para preprocesar la petición.
2. Los manejadores con un método nombrado como `<protocol>_open()` son invocados para manejar la petición. Esta etapa termina cuando un manejador retorna un valor no `None` (ej. una respuesta) o genera una excepción (generalmente `URLError`). Se permite que las excepciones propaguen.

De hecho, el algoritmo anterior se intenta primero para métodos nombrados `default_open()`. Si todos esos métodos retornan `None`, el algoritmo se repite para métodos nombrados como `<protocol>_open()`. Si todos esos métodos retornan `None`, el algoritmo se repite para métodos nombrados como `unknown_open()`.

Tenga en cuenta que la implementación de esos métodos puede involucrar invocaciones de los métodos `open()` y `error()` de la instancia `OpenerDirector` padre.

3. Cada manejador con un método nombrado como `<protocol>_response()` tiene ese método invocado para postprocesar la respuesta.

21.6.3 Objetos BaseHandler

Los objetos `BaseHandler` proporcionan un par de métodos que son útiles directamente y otros que están destinados a ser utilizados por clases derivadas. Estos están pensados para uso directo:

`BaseHandler.add_parent(director)`

Añade un director como padre.

`BaseHandler.close()`

Elimina cualquier padre.

El siguiente atributo y los siguientes métodos sólo deben ser usados por clases derivadas de `BaseHandler`.

Nota: Se ha adoptado la convención de que las subclases que definen los métodos `<protocol>_request()` o `<protocol>_response()` son nombradas `*Processor`; todas las otras son nombradas `*Handler`.

`BaseHandler.parent`

Un `OpenerDirector` válido, el cual puede ser utilizado para abrir usando un protocolo diferente, o para manejar errores.

`BaseHandler.default_open(req)`

Este método no es definido en `BaseHandler`, pero las subclases deben definirlo si quieren cazar todas las URLs.

Este método, si se implementa, será invocado por el `OpenerDirector` padre. Debe retornar un archivo como objeto tal y como se describe en el valor retornado por `open()` de `OpenerDirector` o `None`. Debe generar `URLError` a no ser que algo verdaderamente excepcional ocurra (por ejemplo, `MemoryError` no debe ser mapeado a `URLError`).

Este método será invocado antes de cualquier método de apertura específico de protocolo.

BaseHandler.<protocol>_open(req)

Este método no está definido en *BaseHandler*, pero las subclases deben definirlo si quieren manejar URLs con el protocolo dado.

Este método, si está definido, será invocado por el *OpenerDirector* padre. Los valores retornados deben ser los mismos que para *default_open()*.

BaseHandler.unknown_open(req)

Este método *no* está definido en *BaseHandler*, pero las subclases deben definirlo si quieren cazar todas las URLs sin manejador registrado para abrirlo.

Este método, si está implementado, será invocado por el *parent* de *OpenerDirector*. Los valores retornados deben ser los mismos que para *default_open()*.

BaseHandler.http_error_default(req, fp, code, msg, hdrs)

Este método *no* está definido en *BaseHandler*, pero las subclases deben sobreescribirlo si pretenden proporcionar una solución general para los errores HTTP que de otro modo no se manejarían. Sería invocado automáticamente por el *OpenerDirector* obteniendo el error y no debe ser invocado normalmente en otras circunstancias.

req será un objeto *Request*, *fp* será un objeto como archivo con el cuerpo de error HTTP, *code* será el código de error de tres dígitos, *msg* será la explicación visible para el usuario del código y *hdrs* será un objeto de mapeo con los encabezados del error.

Los valores de retorno y las excepciones generadas deben ser los mismos que aquellos de *urlopen()*.

BaseHandler.http_error_nnn(req, fp, code, msg, hdrs)

nnn debe ser un código de error HTTP de tres dígitos. Este método tampoco está definido en *BaseHandler*, pero será invocado, si existe, en una instancia de una subclase, cuando ocurra un error HTTP con código *nnn*.

Las subclases deben sobrescribir este método para manejar errores HTTP específicos.

Los argumentos, valores de retorno y las excepciones generadas deben ser las mismas que para *http_error_default()*.

BaseHandler.<protocol>_request(req)

Este método *no* está definido en *BaseHandler*, pero las subclases deben definirlo si pretenden preprocesar peticiones del protocolo dado.

Este método, si está definido, será invocado por el *OpenerDirector* padre. *req* será un objeto *Request*. El valor retornado debe ser un objeto *Request*.

BaseHandler.<protocol>_response(req, response)

Este método *no* está definido en *BaseHandler*, pero las subclases deben definirlo si quieren postprocesar respuestas del protocolo dado.

Este método, si está definido, será invocado por el *OpenerDirector* padre. *req* será un objeto *Request*. *response* será un objeto que implementa la misma interfaz que el valor retornado de *urlopen()*. El valor retornado debe implementar la misma interfaz que el valor retornado de *urlopen()*.

21.6.4 Objetos HTTPRedirectHandler

Nota: Algunas redirecciones HTTP requieren acción desde el código del módulo del cliente. Si este es el caso, se genera *HTTPError*. Vea **RFC 2616** para más detalles de los significados precisos de los diferentes códigos de redirección.

Una excepción *HTTPError* generada como consideración de seguridad si el *HTTPRedirectHandler* se presenta con una URL redirigida la cual no es una URL HTTP, HTTPS o FTP.

HTTPRedirectHandler.redirect_request(req, fp, code, msg, hdrs, newurl)

Retorna un *Request* o *None* en respuesta a una redirección. Esto es invocado por las implementaciones pre-determinadas de los métodos *http_error_30*()* cuando se recibe una redirección del servidor. Si puede tomar lugar una redirección, retorna un nuevo *Request* para permitir a *http_error_30*()* realizar la

redirección a *newurl*. De otra forma, genera `HTTPError` si ningún otro manejador debe intentar manejar esta URL, o retorna `None` si no tú pero otro manejador puede.

Nota: La implementación predeterminada de este método no sigue estrictamente [RFC 2616](#), la cual dice que las respuestas 301 y 302 a peticiones POST no deben ser redirigidas automáticamente sin confirmación por el usuario. En realidad, los navegadores permiten redirección automática de esas respuestas, cambiando el POST a un GET y la implementación predeterminada reproduce este comportamiento.

`HTTPRedirectHandler.http_error_301` (*req, fp, code, msg, hdrs*)

Redirecciona a la URL `Location:` o `URI:`. Este método es invocado por el `OpenerDirector` padre al obtener una respuesta HTTP “moved permanently”.

`HTTPRedirectHandler.http_error_302` (*req, fp, code, msg, hdrs*)

Lo mismo que `http_error_301()`, pero invocado para la respuesta “found”.

`HTTPRedirectHandler.http_error_303` (*req, fp, code, msg, hdrs*)

Lo mismo que `http_error_301()`, pero invocado para la respuesta “see other”.

`HTTPRedirectHandler.http_error_307` (*req, fp, code, msg, hdrs*)

Lo mismo que `http_error_301()`, pero invocado para la respuesta “temporary redirect”.

21.6.5 Objetos HTTPCookieProcessor

Las instancias `HTTPCookieProcessor` tienen un atributo:

`HTTPCookieProcessor.cookiejar`

El `http.cookiejar.CookieJar` en el cual las cookies están almacenadas.

21.6.6 Objetos ProxyHandler

`ProxyHandler.<protocol>_open(request)`

El `ProxyHandler` tendrá un método `<protocol>_open()` para cada *protocol* el cual tiene un proxy en el diccionario `proxies` dado en el constructor. El método modificará peticiones para ir a través del proxy, invocando `request.set_proxy()`, e invoca el siguiente manejador en la cadena que ejecuta actualmente el protocolo.

21.6.7 Objetos HTTPPasswordMgr

Estos métodos están disponibles en los objetos `HTTPPasswordMgr` y `HTTPPasswordMgrWithDefaultRealm`.

`HTTPPasswordMgr.add_password` (*realm, uri, user, passwd*)

uri puede ser una única URI o una secuencia de URIs. *realm*, *user* y *passwd* deben ser cadenas. Esto causa que (*user*, *passwd*) se utilice como tokens de autenticación cuando la autenticación para *realm* y para una super URI de ninguna de las URIs dadas es provista.

`HTTPPasswordMgr.find_user_password` (*realm, authuri*)

Obtener usuario/contraseña para el reino y URI dados, si alguno ha sido dado. Este método retornará (`None`, `None`) si no hay usuario/contraseña concordante.

Para objetos `HTTPPasswordMgrWithDefaultRealm`, el reino `None` será buscado si el *realm* dado no tiene usuario/contraseña concordante.

21.6.8 Objetos HTTPPasswordMgrWithPriorAuth

Esta manejador de contraseña extiende `HTTPPasswordMgrWithDefaultRealm` para soportar el seguimiento de URIs para las cuales deben ser enviadas siempre credenciales de autenticación.

`HTTPPasswordMgrWithPriorAuth.add_password(realm, uri, user, passwd, is_authenticated=False)`
realm, uri, user, passwd son como para `HTTPPasswordMgr.add_password()`. *is_authenticated* establece el valor inicial del indicador *is_authenticated* para la URI o lista de URIs dadas. Si se especifica *is_authenticated* como `True`, *realm* se ignora.

`HTTPPasswordMgrWithPriorAuth.find_user_password(realm, authuri)`
Lo mismo que para objetos `HTTPPasswordMgrWithDefaultRealm`

`HTTPPasswordMgrWithPriorAuth.update_authenticated(self, uri, is_authenticated=False)`
Actualiza el indicador *is_authenticated* para la *uri* o lista de URIs dadas.

`HTTPPasswordMgrWithPriorAuth.is_authenticated(self, authuri)`
Retorna el estado actual del indicador *is_authenticated* para la URI dada.

21.6.9 Objetos AbstractBasicAuthHandler

`AbstractBasicAuthHandler.http_error_auth_reged(authreq, host, req, headers)`
Maneja una autenticación de petición obteniendo un par usuario/contraseña y reintentando la petición. *authreq* debe ser el nombre del encabezado donde la información sobre el reino se incluye en la petición, *host* especifica la URL y ruta para la cual autenticar, *req* debe ser el objeto `Request` (fallido) y *headers* deben ser los encabezados de error.

host es una autoridad (ej. "python.org") o una URL conteniendo un componente de autoridad (ej. "http://python.org/"). En cualquier caso, la autoridad no debe contener un componente userinfo (por lo que "python.org" y "python.org:80" están bien, "joe:password@python.org" no).

21.6.10 Objetos HTTPBasicAuthHandler

`HTTPBasicAuthHandler.http_error_401(req, fp, code, msg, hdrs)`
Reintenta la petición con la información de autenticación, si está disponible.

21.6.11 Objetos ProxyBasicAuthHandler

`ProxyBasicAuthHandler.http_error_407(req, fp, code, msg, hdrs)`
Reintenta la petición con la información de autenticación, si está disponible.

21.6.12 Objetos AbstractDigestAuthHandler

`AbstractDigestAuthHandler.http_error_auth_reged(authreq, host, req, headers)`
authreq debe ser el nombre del encabezado donde la información sobre el reino está incluida en la petición, *host* debe ser el host al que autenticar, *req* debe ser el objeto `Request` (fallido) y *headers* deben ser los encabezados de error.

21.6.13 Objetos HTTPDigestAuthHandler

`HTTPDigestAuthHandler.http_error_401` (*req, fp, code, msg, hdrs*)
Reintenta la petición con la información de autenticación, si está disponible.

21.6.14 Objetos ProxyDigestAuthHandler

`ProxyDigestAuthHandler.http_error_407` (*req, fp, code, msg, hdrs*)
Reintenta la petición con la información de autenticación, si está disponible.

21.6.15 Objetos HTTPHandler

`HTTPHandler.http_open` (*req*)
Envía una petición HTTP, que puede ser GET o POST, dependiendo de `req.has_data()`.

21.6.16 Objetos HTTPSHandler

`HTTPSHandler.https_open` (*req*)
Envía una petición HTTPS, que puede ser GET o POST, dependiendo de `req.has_data()`.

21.6.17 Objetos FileHandler

`FileHandler.file_open` (*req*)
Abre el archivo localmente, si no hay nombre de host, o el nombre de host es `'localhost'`.
Distinto en la versión 3.2: Este método es aplicable sólo para nombres de host locales. Cuando un nombre de host remoto es dado, se genera una excepción `URLError`.

21.6.18 Objetos DataHandler

`DataHandler.data_open` (*req*)
Lee una URL de datos. Este tipo de URL contiene el contenido codificado en la URL misma. La sintaxis de la URL de datos se especifica en [RFC 2397](#). Esta implementación ignora los espacios en blanco en datos codificados como base64 así que la URL puede ser envuelta en cualquier archivo fuente del que proviene. Pero a pesar de que a algunos navegadores no les importa si falta relleno al final de una URL codificada como base64, esta implementación generará un `ValueError` en este caso.

21.6.19 Objetos FTPHandler

`FTPHandler.ftp_open` (*req*)
Abre el archivo FTP indicado por *req*. El inicio de sesión siempre se realiza con un usuario y contraseña vacíos.

21.6.20 Objetos CacheFTPHandler

Los objetos *CacheFTPHandler* son objetos *FTPHandler* con los siguientes métodos adicionales:

```
CacheFTPHandler.setTimeout(t)
```

Establece el tiempo de expiración de conexiones a t segundos.

```
CacheFTPHandler.setMaxConns (m)
```

Establece el número máximo de conexiones cacheadas a m .

21.6.21 Objetos UnknownHandler

```
UnknownHandler.unknown open()
```

Genera una excepción `URL_Error`.

21.6.22 Objetos HTTPErrorProcessor

```
HTTPErrorProcessor.http_response(request, response)
```

Procesa errores de respuestas HTTP.

Para códigos de error que no están en el rango de los 200, el objeto de respuesta es retornado inmediatamente.

Para códigos de error que no están en el rango de los 200, esto simplemente pasa el trabajo a los métodos del manejador `http_error_<type>()`, mediante `OpenerDirector.error()`. Eventualmente, `HTTPDefaultErrorHandler` generará un `HTTPError` si ningún otro manejador maneja el error.

```
HTTPErrorProcessor.https response(request, response)
```

Procesa los errores HTTPS de las respuestas.

Este comportamiento es el mismo que `http_response()`.

21.6.23 Ejemplos

Adicionalmente a los ejemplos siguientes, se dan más ejemplos en `urllib-howto`.

Este ejemplo obtiene la página principal `python.org` y despliega los primeros 300 bytes de ella.

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n
<meta http-equiv="content-type" content="text/html; charset=utf-8" />\n
<title>Python Programming '
```

Tenga en cuenta que `urlopen` retorna un objeto de bytes. Esto es porque no hay forma para `urlopen` de determinar automáticamente la codificación del flujo de bytes que recibe del servidor HTTP. En general, un programa decodificará el objeto de bytes retornado a cadena de caracteres una vez que determine o adivine la codificación apropiada.

El siguiente documento W3C, <https://www.w3.org/International/O-charset>, lista las diferentes formas en la cual un documento (X)HTML o XML podría haber especificado su información de codificación.

Ya que el sitio web `python.org` usa codificación *utf-8* tal y como se especifica en su etiqueta meta, usaremos la misma para decodificar el objeto de bytes.

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
... 
```

(continué en la próxima página)

(proviene de la página anterior)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml
```

Es posible conseguir el mismo resultado sin usar la aproximación *context manager*.

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> print(f.read(100).decode('utf-8'))
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml
```

En el siguiente ejemplo, estamos enviando un flujo de datos a la entrada estándar de un CGI y leyendo los datos que nos retorna. Tenga en cuenta que este ejemplo sólo funcionará cuando la instalación de Python soporte SSL.

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi',
...                             data=b'This data is passed to stdin of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

El código para el CGI de muestra usado en el ejemplo anterior es:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

Aquí hay un ejemplo de realizar una petición PUT usando *Request*:

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA, method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

Uso de Autenticación HTTP Básica:

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
urllib.request.urlopen('http://www.example.com/login.html')
```

build_opener() proporciona muchos manejadores por defecto, incluyendo un *ProxyHandler*. De forma predeterminada, *ProxyHandler* usa las variables de entorno llamadas `<scheme>_proxy`, donde `<scheme>` es el esquema URL involucrado. Por ejemplo, se lee la variable de entorno `http_proxy` para obtener la URL del proxy HTTP.

Este ejemplo reemplaza el *ProxyHandler* predeterminado por uno que usa URLs de proxy suministradas mediante programación y añade soporte de autorización de proxy con *ProxyBasicAuthHandler*.


```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/
↪'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.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')
```

Añadiendo encabezados HTTP:

Usa el argumento *headers* en el constructor de *Request*, o:

```
import urllib.request
req = urllib.request.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 = urllib.request.urlopen(req)
```

OpenerDirector añade automáticamente un encabezado *User-Agent* a cada *Request*. Para cambiar esto:

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

También, recuerda que algunos encabezados estándar (*Content-Length*, *Content-Type* y *Host*) son añadidos cuando se pasa *Request* a *urlopen()* (o *OpenerDirector.open()*).

Aquí hay un ejemplo de sesión que usa el método GET para obtener una URL que contiene los parámetros:

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
... 
```

El siguiente ejemplo usa el método POST en su lugar. Tenga en cuenta que la salida de parámetros desde *urlencode* es codificada a bytes antes de ser enviados a *urlopen* como datos:

```
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr", data) as f:
...     print(f.read().decode('utf-8'))
... 
```

El siguiente ejemplo usa un proxy HTTP especificado, sobrescribiendo las configuraciones de entorno:

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.FancyURLopener(proxies)
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
... 
```

El siguiente ejemplo no usa proxies, sobrescribiendo las variables de entorno:


```
>>> import urllib.request
>>> opener = urllib.request.FancyURLopener({})
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
... 
```

21.6.24 Interfaz heredada

Las siguientes funciones y clases están portadas desde el módulo `urllib` de Python 2 (en oposición a `urllib2`). Ellas pueden estar obsoletas en algún punto del futuro.

`urllib.request.urlretrieve(url, filename=None, reporthook=None, data=None)`

Copia un objeto de red denotado por una URL a un archivo local. Si la URL apunta a un archivo local, el objeto no será copiado a no ser que sea suministrado un nombre de archivo. Retorna una tupla (`filename`, `headers`) donde `filename` es el nombre de archivo local bajo el cual el objeto puede ser encontrado y `headers` es lo que retorna el método `info()` retornado por `urlopen()` (para un objeto remoto). Las excepciones son las mismas que para `urlopen()`.

El segundo argumento, si está presente, especifica la localización a la que será copiada el objeto (si está ausente, la localización será un archivo temporal con un nombre generado). El tercer argumento, si está presente, es un objeto invocable que será invocado una vez que se establezca la conexión de red y después de eso una vez después de cada lectura de bloque. Al invocable se le pasarán tres argumentos; una cuenta de los bloques transferidos hasta el momento, un tamaño de bloque en bytes y el tamaño total del archivo. El tercer argumento puede ser `-1` en servidores FTP antiguos los cuales no retornan un tamaño de archivo en respuesta a una solicitud de recuperación.

El siguiente ejemplo ilustra el escenario de uso más común:

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

Si la `url` usa el esquema de identificador `http:`, el argumento opcional `data` puede ser dado para especificar una petición POST (normalmente el tipo de petición es GET). El argumento `data` debe ser un objeto de bytes en formato `application/x-www-form-urlencoded`; vea la función `urllib.parse.urlencode()`.

`urlretrieve()` generará `ContentTooShortError` cuando detecte que la cantidad de datos disponibles sea menor que la cantidad esperada (la cual es el tamaño reportado por un encabezado `Content-Length`). Esto puede ocurrir, por ejemplo, cuando se interrumpe la descarga.

El `Content-Length` es tratado como un límite inferior: si no hay más datos a leer, `urlretrieve` lee más datos, pero si están disponibles menos datos, se genera la excepción.

Puedes seguir obteniendo los datos descargados en este caso, son almacenados en el atributo `content` de la instancia de la excepción.

Si no fue proporcionado el encabezado `Content-Length`, `urlretrieve` no puede comprobar el tamaño de los datos que han sido descargados, y sólo los retorna. En este caso sólo tienes que asumir que la descarga fue exitosa.

`urllib.request.urlcleanup()`

Limpiar archivos temporales que pueden haber quedado tras llamadas anteriores a `urlretrieve()`.

class `urllib.request.URLopener` (`proxies=None`, `**x509`)

Obsoleto desde la versión 3.3.

Clase base para apertura y lectura de URLs. A no ser que necesites soporte de apertura de objetos usando esquemas diferentes a `http:`, `ftp:` o `file:`, probablemente quieras usar `FancyURLopener`.

Por defecto, la clase `URLopener` envía un encabezado `User-Agent` de `urllib/VVV`, donde VVV es el número de versión `urllib`. Las aplicaciones pueden definir su propio encabezado `User-Agent` heredando

de `URLOpener` o `FancyURLOpener` y estableciendo el atributo de clase `version` a un valor de cadena de caracteres apropiado en la definición de la subclase.

El parámetro opcional `proxies` debe ser un diccionario mapeando nombres de esquemas a URLs de proxy, donde un diccionario vacío apaga los proxies completamente. Su valor predeterminado es `None`, en cuyo caso las configuraciones de proxy del entorno serán usadas si están presentes, como ha sido discutido en la definición de `urlopen()`, arriba.

Parámetros adicionales de palabra clave, recogidos en *x509*, pueden ser usados por autenticación del cliente cuando usan el esquema `https:`. Las palabras claves `key_file` y `cert_file` están soportadas para proveer una clave y certificado SSL; ambos son necesarias para soportar autenticación de cliente.

Los objetos `URLOpener` generarán una excepción `OSError` si el servidor retorna un código de error.

open (*fullurl*, *data=None*)

Abre *fullurl* usando el protocolo apropiado. Este método configura la información de caché e información de proxy, entonces invoca el método apropiado con sus argumentos de entrada. Si el esquema no está reconocido, se invoca `open_unknown()`. El argumento *data* tiene el mismo significado que el argumento *data* de `urlopen()`.

Este método siempre entrecomilla *fullurl* usando `quote()`.

open_unknown (*fullurl*, *data=None*)

Interfaz sobrescribible para abrir tipos de URL desconocidos.

retrieve (*url*, *filename=None*, *reporthook=None*, *data=None*)

Obtiene el contenido de *url* y lo coloca en *filename*. El valor retornado es una tupla que consiste de un nombre de archivo local y un objeto `email.message.Message` conteniendo los encabezados de respuesta (para URLs remotas) o `None` (para URLs locales). El invocador debe entonces abrir y leer los contenidos de *filename*. Si *filename* no está dado y la URL refiere a un archivo local, se retorna el nombre de archivo de entrada. Si la URL no es local y no se da *filename*, el nombre de archivo es la salida de la función `tempfile.mktemp()` con un sufijo que concuerda con el sufijo del último componente de la ruta de la URL de entrada. Si se da *reporthook*, debe ser una función que acepte tres parámetros numéricos: Un número de fragmento, se leen los fragmentos de tamaño máximo y el tamaño total de la descarga (-1 si es desconocida). Será invocada una vez al comienzo y después de que cada fragmento de datos sea leído de la red. *reporthook* es ignorado para URLs locales.

Si la *url* usa el identificador de esquema `http:`, el argumento opcional *data* puede ser dado para especificar una petición POST (normalmente el tipo de petición es GET). El argumento *data* debe estar en formato estándar `application/x-www-form-urlencoded`; vea la función `urllib.parse.urlencode()`.

version

Variable que especifica el agente de usuario del objeto abridor. Para obtener `urllib` para decir a los servidores que es un agente de usuario particular, establece esto en una subclase como una variable de clase o en el constructor antes de invocar el constructor base.

class `urllib.request.FancyURLOpener` (...)

Obsoleto desde la versión 3.3.

`FancyURLOpener` hereda de `URLOpener` proveyendo manejo predeterminado para los siguientes códigos de respuesta HTTP: 301, 302, 303, 307 y 401. Para los códigos de respuesta 30x listados anteriormente, se usa el encabezado `Location` para obtener la URL actual. Para códigos de respuesta 401 (autenticación requerida), se realiza autenticación HTTP. Para los códigos de respuesta 30x, la recursión está limitada por el valor del atributo `maxentries`, el cual por defecto es 10.

Para todos los demás códigos de respuesta, se invoca al método `http_error_default()`, que puede sobrescribir en subclases para manejar el error de manera adecuada.

Nota: De acuerdo a la carta de **RFC 2616**, las respuestas a las peticiones POST 301 y 302 no debe ser redirigidas automáticamente sin confirmación por el usuario. En realidad, los navegadores permiten redirección automática de esas respuestas, cambiando de POST a GET, y `urllib` reproduce este comportamiento.

Los parámetros del constructor son el mismo que aquellos para `URLopener`.

Nota: Cuando se realiza autenticación básica, una instancia `FancyURLopener` invoca a su método `prompt_user_passwd()`. La implementación predeterminada pregunta a los usuarios la información requerida en la terminal de control. Una subclase puede sobrescribir este método para soportar un comportamiento más apropiado si se necesita.

La clase `FancyURLopener` ofrece un método adicional que debe ser sobrecargado para proveer el comportamiento apropiado:

prompt_user_passwd (*host, realm*)

Retorna la información necesaria para autenticar el usuario en el host dado en el reino de seguridad especificado. El valor retornado debe ser una tupla (`user, password`), la cual puede ser usada para autenticación básica.

La implementación solicita esta información en el terminal; una aplicación debe sobrescribir este método para usar un modelo de interacción apropiado en el entorno local.

21.6.25 Restricciones `urllib.request`

- Actualmente, sólo uno de los siguientes protocolo están soportados: HTTP (versiones 0.9 y 1.0), FTP, archivos locales y URLs de datos.

Distinto en la versión 3.4: Añadido soporte para URLs de datos.

- La característica de caché de `urlretrieve()` ha sido deshabilitada hasta que alguien encuentre el tiempo para hackear el procesamiento adecuado de los encabezados de tiempo de Expiración.
- Debería haber una función para consultar si una URL en particular está en la caché.
- Para compatibilidad con versiones anteriores, si una URL parece apuntar a un archivo local pero el archivo no puede ser abierto, la URL es reinterpretada usando el protocolo FTP. Esto a veces puede causar mensajes de error confusos.
- Las funciones `urlopen()` y `urlretrieve()` pueden causar retrasos arbitrariamente largos mientras esperan a que se configure una conexión de red. Esto significa que es difícil construir un cliente Web interactivo usando estas funciones sin utilizar hilos.
- Los datos retornados por `urlopen()` o `urlretrieve()` son los datos en crudo retornados por el servidor. Estos pueden ser datos binarios (como una imagen), texto plano o (por ejemplo) HTML. El protocolo HTTP provee información de tipo en el encabezado de respuesta, el cual puede ser inspeccionado mirando el encabezado `Content-Type`. Si los datos retornados son HTML, puedes usar el módulo `html.parser` para analizarlos.
- El código que maneja el protocolo FTP no puede diferenciar entre un archivo y un directorio. Esto puede llevar a un comportamiento inesperado cuando se intenta leer una URL que apunta a un archivo que no es accesible. Si la URL termina en un `/`, se asume que se refiere a un directorio y será manejada acordeamente. Pero si un intento de leer un archivo lleva a un error 550 (lo que significa que la URL no puede ser encontrada o no es accesible, a menudo por razones de permisos), entonces se trata la ruta como un directorio para manejar el caso cuando un directorio es especificado por una URL pero el `/` trasero ha sido dejado fuera. Esto puede causar resultados erróneos cuando intenta obtener un archivo cuyos permisos de lectura lo hacen inaccesible; el código FTP intentará leerlo, fallará con un error 550 y entonces realizará un listado de directorio para el archivo ilegible. Si se necesita un control más detallado, considere usar el módulo `ftplib`, heredando `FancyURLopener` o cambiando `_urloper` para ajustarlo a tus necesidades.

21.7 `urllib.response` — Clases de respuesta usadas por `urllib`

El módulo `urllib.response` define funciones y clases las cuales definen una interfaz mínima como objeto de archivo, incluyendo `read()` y `readline()`. El objeto de respuesta típico es una instancia `addinfourl`, la cual define un método `info()` y este retorna encabezados y un método `geturl()` que retorna la url. Las funciones definidas por este módulo son usadas internamente por el módulo `urllib.request`.

21.8 `urllib.parse` — Analiza URL en componentes

Código fuente: [Lib/urllib/parse.py](#)

Este modulo define una interfaz estándar para separar cadenas de texto del Localizador de recursos uniforme (más conocido por las siglas URL, del inglés *Uniform Resource Locator*) en componentes (esquema de dirección, ubicación de red, ruta de acceso, etc.), para poder combinar los componentes nuevamente en una cadena de texto URL, y convertir una «URL relativa» a una URL absoluta a partir de una «URL base».

Este módulo ha sido diseñado para coincidir con el estándar de Internet RFC de los Localizadores de recursos uniformes relativos. Este modulo soporta los siguientes esquemas URL: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `mailto`, `mms`, `news`, `nnntp`, `prospero`, `rsync`, `rtsp`, `rtspu`, `sftp`, `shhttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`, `ws`, `wss`.

El módulo `urllib.parse` define funciones que se dividen en dos categorías amplias: análisis de URL y cita de URL. Estos se tratan en detalle en las secciones siguientes.

21.8.1 Análisis de URL

Las funciones de análisis de url se centran en dividir una cadena de dirección URL en sus componentes o en combinar componentes de dirección URL en una cadena de dirección URL.

`urllib.parse.urlparse(urlstring, scheme="", allow_fragments=True)`

Analice una dirección URL en seis componentes y retorne una tupla de 6 elementos *named tuple*. Esto corresponde a la estructura general de una URL: `scheme://netloc/path;parameters?query#fragment`. Cada elemento de la tupla es una cadena, posiblemente vacía. Los componentes no se dividen en piezas más pequeñas (por ejemplo, la ubicación de red es una sola cadena) y los caracteres % de escapes no se expanden. Los delimitadores como se muestra anteriormente no forman parte del resultado, excepto una barra diagonal inicial en el componente *path*, que se conserva si está presente. Por ejemplo:

```
>>> from urllib.parse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o
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'
```

Siguiendo las especificaciones de sintaxis en [RFC 1808](#), `urlparse` reconoce un `netloc` sólo si es introducido correctamente por `"/"`. De lo contrario, se supone que la entrada es una dirección URL relativa y, por lo tanto, comienza con un componente de ruta de acceso.

```
>>> from urllib.parse 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')
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='')
```

El argumento *scheme* proporciona el esquema de direccionamiento predeterminado, que solo se utilizará si la dirección URL no especifica uno. Debe ser del mismo tipo (texto o bytes) que *urlstring*, excepto que el valor predeterminado `''` siempre está permitido, y se convierte automáticamente a `b''` si aplica.

Si el argumento *allow_fragments* es falso, los identificadores de fragmento no son reconocidos. Sino que, son analizados como parte de la ruta, parámetros o componentes de la consulta y el *fragment* es configurado como una cadena vacía en el valor retornado.

El valor retornado es un *named tuple*, lo que significa que se puede tener acceso a sus elementos por índice o como atributos con nombre, que son:

Atributo	Índice	Valor	Valor si no está presente
<code>scheme</code>	0	Especificador de esquema de URL	parámetro <i>scheme</i>
<code>netloc</code>	1	Parte de ubicación de red	cadena vacía
<code>path</code>	2	Hierarchical path	cadena vacía
<code>params</code>	3	Parámetros para el último elemento de ruta de acceso	cadena vacía
<code>query</code>	4	Componente de consulta	cadena vacía
<code>fragment</code>	5	Identificador de fragmento	cadena vacía
<code>username</code>		Nombre de usuario	<i>None</i>
<code>password</code>		Contraseña	<i>None</i>
<code>hostname</code>		Nombre de host (minúsculas)	<i>None</i>
<code>port</code>		Número de puerto como entero, si está presente	<i>None</i>

La lectura del atributo `port` generará un *ValueError* si se especifica un puerto no válido en la dirección URL. Consulte la sección *Resultados del análisis estructurado* para obtener más información sobre el objeto de resultado.

Los corchetes no coincidentes en el atributo `netloc` generarán un *ValueError*.

Los caracteres del atributo `netloc` que se descomponen en la normalización de NFKC (según lo utilizado por la codificación IDNA) en cualquiera de `/`, `?`, `#`, `@` o `:` lanzará un *ValueError*. Si la dirección URL se descompone antes del análisis, no se producirá ningún error.

Como es el caso con todas las tuplas con nombre, la subclase tiene algunos métodos y atributos adicionales que son particularmente útiles. Uno de estos métodos es `_replace()`. El método `_replace()` retornará un nuevo objeto *ParseResult* reemplazando los campos especificados por nuevos valores.

```
>>> from urllib.parse import urlparse
>>> u = urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
>>> u
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> u._replace(scheme='http')
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            ↪ params='', query='', fragment='')
```

Advertencia: `urlparse()` does not perform validation. See [URL parsing security](#) for details.

Distinto en la versión 3.2: Se han añadido capacidades de análisis de URL IPv6.

Distinto en la versión 3.3: El fragmento ahora se analiza para todos los esquemas de URL (a menos que `allow_fragment` sea `false`), de acuerdo con [RFC 3986](#). Anteriormente, existía una lista blanca de esquemas que admiten fragmentos.

Distinto en la versión 3.6: Los números de puerto fuera de rango ahora generan `ValueError`, en lugar de retornar `None`.

Distinto en la versión 3.8: Los caracteres que afectan al análisis de netloc en la normalización de NFKC ahora generarán `ValueError`.

```
urllib.parse.parse_qs(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8',
                      errors='replace', max_num_fields=None, separator='&')
```

Analizar una cadena de consulta proporcionada como argumento de cadena (datos de tipo `application/x-www-form-urlencoded`). Los datos se retornan como un diccionario. Las claves de diccionario son los nombres de variable de consulta únicos y los valores son listas de valores para cada nombre.

El argumento opcional `keep_blank_values` es un indicador que indica si los valores en blanco de las consultas codificadas por porcentaje deben tratarse como cadenas en blanco. Un valor `true` indica que los espacios en blanco deben conservarse como cadenas en blanco. El valor `false` predeterminado indica que los valores en blanco deben omitirse y tratarse como si no se hubieran incluido.

El argumento opcional `strict_parsing` es una marca que indica qué hacer con los errores de análisis. Si es `false` (valor predeterminado), los errores se omiten silenciosamente. Si es `true`, los errores generan una excepción `ValueError`.

Los parámetros opcionales `encoding` y `errors` especifican cómo decodificar secuencias codificadas porcentualmente en caracteres Unicode, tal como lo acepta el método `bytes.decode()`.

El argumento opcional `max_num_fields` es el número máximo de campos que se van a leer. Si se establece, se produce un `ValueError` si hay más de `max_num_fields` campos leídos.

The optional argument `separator` is the symbol to use for separating the query arguments. It defaults to `&`.

Utilice la función `urllib.parse.urlencode()` (con el parámetro `doseq` establecido en `True`) para convertir dichos diccionarios en cadenas de consulta.

Distinto en la versión 3.2: Agregue los parámetros `encoding` y `errors`.

Distinto en la versión 3.8: Añadido parámetro `max_num_fields`.

Distinto en la versión 3.8.8: Added `separator` parameter with the default value of `&`. Python versions earlier than Python 3.8.8 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

```
urllib.parse.parse_qsl(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8',
                      errors='replace', max_num_fields=None, separator='&')
```

Analizar una cadena de consulta proporcionada como argumento de cadena (datos de tipo `application/x-www-form-urlencoded`). Los datos se retornan como una lista de pares de nombre y valor.

El argumento opcional `keep_blank_values` es un indicador que indica si los valores en blanco de las consultas codificadas por porcentaje deben tratarse como cadenas en blanco. Un valor `true` indica que los espacios en blanco deben conservarse como cadenas en blanco. El valor `false` predeterminado indica que los valores en blanco deben omitirse y tratarse como si no se hubieran incluido.

El argumento opcional `strict_parsing` es una marca que indica qué hacer con los errores de análisis. Si es `false` (valor predeterminado), los errores se omiten silenciosamente. Si es `true`, los errores generan una excepción `ValueError`.

Los parámetros opcionales `encoding` y `errors` especifican cómo decodificar secuencias codificadas porcentualmente en caracteres Unicode, tal como lo acepta el método `bytes.decode()`.

El argumento opcional `max_num_fields` es el número máximo de campos que se van a leer. Si se establece, se produce un `ValueError` si hay más de `max_num_fields` campos leídos.

The optional argument `separator` is the symbol to use for separating the query arguments. It defaults to `&`.

Utilice la función `urllib.parse.urlencode()` para convertir dichas listas de pares en cadenas de consulta.

Distinto en la versión 3.2: Agregue los parámetros `encoding` y `errors`.

Distinto en la versión 3.8: Añadido parámetro `max_num_fields`.

Distinto en la versión 3.8.8: Added `separator` parameter with the default value of `&`. Python versions earlier than Python 3.8.8 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

`urllib.parse.urlunparse(parts)`

Construir una URL a partir de una tupla retornada por `urlparse()`. El argumento `parts` puede ser iterable de seis elementos. Esto puede dar lugar a una dirección URL ligeramente diferente, pero equivalente, si la dirección URL que se analizó originalmente tenía delimitadores innecesarios (por ejemplo, un `?` con una consulta vacía; la RFC indica que son equivalentes).

`urllib.parse.urlsplit(urlstring, scheme="", allow_fragments=True)`

Esto es similar a `urlparse()`, pero no divide los parámetros de la dirección URL. Esto se debe utilizar generalmente en lugar de `urlparse()` si se desea la sintaxis de URL más reciente que permite aplicar parámetros a cada segmento de la parte `path` de la dirección URL (consulte [RFC 2396](#)). Se necesita una función independiente para separar los segmentos y parámetros de ruta. Esta función retorna un 5-item *named tuple*:

```
(addressing scheme, network location, path, query, fragment identifier).
```

El valor retornado es un *named tuple*, se puede acceder a sus elementos por índice o como atributos con nombre:

Atributo	Índice	Valor	Valor si no está presente
<code>scheme</code>	0	Especificador de esquema de URL	parámetro <code>scheme</code>
<code>netloc</code>	1	Parte de ubicación de red	cadena vacía
<code>path</code>	2	Hierarchical path	cadena vacía
<code>query</code>	3	Componente de consulta	cadena vacía
<code>fragment</code>	4	Identificador de fragmento	cadena vacía
<code>username</code>		Nombre de usuario	<code>None</code>
<code>password</code>		Contraseña	<code>None</code>
<code>hostname</code>		Nombre de host (minúsculas)	<code>None</code>
<code>port</code>		Número de puerto como entero, si está presente	<code>None</code>

La lectura del atributo `port` generará un `ValueError` si se especifica un puerto no válido en la dirección URL. Consulte la sección [Resultados del análisis estructurado](#) para obtener más información sobre el objeto de resultado.

Los corchetes no coincidentes en el atributo `netloc` generarán un `ValueError`.

Los caracteres del atributo `netloc` que se descomponen en la normalización de NFKC (según lo utilizado por la codificación IDNA) en cualquiera de `/`, `?`, `#`, `@` o `:` lanzará un `ValueError`. Si la dirección URL se descompone antes del análisis, no se producirá ningún error.

Following some of the [WHATWG spec](#) that updates RFC 3986, leading C0 control and space characters are stripped from the URL. `\n`, `\r` and tab `\t` characters are removed from the URL at any position.

Advertencia: `urlsplit()` does not perform validation. See [URL parsing security](#) for details.

Distinto en la versión 3.6: Los números de puerto fuera de rango ahora generan `ValueError`, en lugar de retornar `None`.

Distinto en la versión 3.8: Los caracteres que afectan al análisis de netloc en la normalización de NFKC ahora generarán `ValueError`.

Distinto en la versión 3.8.10: ASCII newline and tab characters are stripped from the URL.

Distinto en la versión 3.8.17: Leading WHATWG C0 control and space characters are stripped from the URL.

`urllib.parse.urlunsplit(parts)`

Combine los elementos de una tupla retornados por `urlsplit()` en una URL completa como una cadena. El argumento `parts` puede ser iterable de cinco elementos. Esto puede dar lugar a una dirección URL ligeramente diferente, pero equivalente, si la dirección URL que se analizó originalmente tenía delimitadores innecesarios (por ejemplo, un `?` con una consulta vacía; la RFC indica que son equivalentes).

`urllib.parse.urljoin(base, url, allow_fragments=True)`

Construya una URL completa («absoluta») combinando una «URL base» (`base`) con otra URL (`url`). Informalmente, esto utiliza componentes de la dirección URL base, en particular el esquema de direccionamiento, la ubicación de red y (parte de) la ruta de acceso, para proporcionar los componentes que faltan en la dirección URL relativa. Por ejemplo:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

El argumento `allow_fragments` tiene el mismo significado y el valor predeterminado que para `urlparse()`.

Nota: Si `url` es una URL absoluta (es decir, a partir de `//` o `scheme://`), el nombre de host y/o esquema de `url` estará presente en el resultado. Por ejemplo:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         'http://www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

Si no desea ese comportamiento, preprocesa las partes `url` con `urlsplit()` y `urlunsplit()`, eliminando posibles partes `esquema` y `netloc`.

Distinto en la versión 3.5: Comportamiento actualizado para que coincida con la semántica definida en **RFC 3986**.

`urllib.parse.urldefrag(url)`

Si `url` contiene un identificador de fragmento, retorne una versión modificada de `url` sin identificador de fragmento y el identificador de fragmento como una cadena independiente. Si no hay ningún identificador de fragmento en `url`, retorne `url` sin modificar y una cadena vacía.

El valor retornado es un *named tuple*, se puede acceder a sus elementos por índice o como atributos con nombre:

Atributo	Índice	Valor	Valor si no está presente
<code>url</code>	0	URL sin fragmento	cadena vacía
<code>fragment</code>	1	Identificador de fragmento	cadena vacía

Consulte la sección *Resultados del análisis estructurado* para obtener más información sobre el objeto de resultado.

Distinto en la versión 3.2: El resultado es un objeto estructurado en lugar de una simple tupla de 2.

`urllib.parse.unwrap(url)`

Extract the url from a wrapped URL (that is, a string formatted as `<URL:scheme://host/path>`, `<scheme://host/path>`, `URL:scheme://host/path` or `scheme://host/path`). If `url` is not a wrapped URL, it is returned without changes.

21.8.2 URL parsing security

The `urlsplit()` and `urlparse()` APIs do not perform **validation** of inputs. They may not raise errors on inputs that other applications consider invalid. They may also succeed on some inputs that might not be considered URLs elsewhere. Their purpose is for practical functionality rather than purity.

Instead of raising an exception on unusual input, they may instead return some component parts as empty strings. Or components may contain more than perhaps they should.

We recommend that users of these APIs where the values may be used anywhere with security implications code defensively. Do some verification within your code before trusting a returned component part. Does that `scheme` make sense? Is that a sensible `path`? Is there anything strange about that `hostname`? etc.

21.8.3 Análisis de bytes codificados ASCII

Las funciones de análisis de URL se diseñaron originalmente para funcionar solo en cadenas de caracteres. En la práctica, es útil poder manipular las direcciones URL correctamente citadas y codificadas como secuencias de bytes ASCII. En consecuencia, las funciones de análisis de URL de este módulo funcionan en los objetos `bytes` y `bytearray`, además de los objetos `str`.

Si se pasan datos `str`, el resultado también contendrá solo los datos `str`. Si se pasan datos `bytes` o `bytearray`, el resultado contendrá solo los datos `bytes`.

Si intenta mezclar datos `str` con `bytes` o `bytearray` en una sola llamada de función, se producirá un `TypeError`, mientras que al intentar pasar valores de bytes no ASCII se desencadenará `UnicodeDecodeError`.

Para admitir una conversión más sencilla de objetos de resultado entre `str` y `bytes`, todos los valores retornados de las funciones de análisis de URL proporcionan un método `encode()` (cuando el resultado contiene `str` data) o un método `decode()` (cuando el resultado contiene datos `bytes`). Las firmas de estos métodos coinciden con las de los métodos correspondientes `str` y `bytes` (excepto que la codificación predeterminada es `'ascii'` en lugar de `'utf-8'`). Cada uno produce un valor de un tipo correspondiente que contiene datos `bytes` (para los métodos `encode()`) o `str` (para `decode()` métodos).

Las aplicaciones que necesitan operar en direcciones URL potencialmente citadas incorrectamente que pueden contener datos no ASCII tendrán que realizar su propia descodificación de bytes a caracteres antes de invocar los métodos de análisis de URL.

El comportamiento descrito en esta sección solo se aplica a las funciones de análisis de URL. Las funciones de citación de URL utilizan sus propias reglas al producir o consumir secuencias de bytes como se detalla en la documentación de las funciones de citación de URL individuales.

Distinto en la versión 3.2: Las funciones de análisis de URL ahora aceptan secuencias de bytes codificadas en ASCII

21.8.4 Resultados del análisis estructurado

Los objetos resultantes de las funciones `urlparse()`, `urlsplit()` y `urldefrag()` son subclases del tipo `tuple`. Estas subclases agregan los atributos enumerados en la documentación para esas funciones, el soporte de codificación y decodificación descrito en la sección anterior, así como un método adicional:

`urllib.parse.SplitResult.geturl()`

Retorna la versión re-combinada de la dirección URL original como una cadena. Esto puede diferir de la dirección URL original en que el esquema se puede normalizar a minúsculas y los componentes vacíos pueden descartarse. En concreto, se quitarán los parámetros vacíos, las consultas y los identificadores de fragmento.

Para los resultados `urldefrag()`, solo se eliminarán los identificadores de fragmento vacíos. Para los resultados `urlsplit()` y `urlparse()`, todos los cambios observados se realizarán en la URL retornada por este método.

El resultado de este método permanece inalterado si se pasa de nuevo a través de la función de análisis original:

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

Las clases siguientes proporcionan las implementaciones de los resultados del análisis estructurado cuando se opera en objetos *str*:

class `urllib.parse.DefragResult` (*url, fragment*)

Clase concreta para los resultados de `urldefrag()` que contienen datos *str*. El método `encode()` retorna una instancia *DefragResultBytes*.

Nuevo en la versión 3.2.

class `urllib.parse.ParseResult` (*scheme, netloc, path, params, query, fragment*)

Clase concreta para los resultados de `urlparse()` que contiene *str* data. El método `encode()` retorna una instancia *ParseResultBytes*.

class `urllib.parse.SplitResult` (*scheme, netloc, path, query, fragment*)

Clase concreta para los resultados de `urlsplit()` que contiene *str* data. El método `encode()` retorna una instancia *SplitResultBytes*.

Las clases siguientes proporcionan las implementaciones de los resultados del análisis cuando se opera en objetos *bytes* o *bytearray*:

class `urllib.parse.DefragResultBytes` (*url, fragment*)

Clase concreta para los resultados de `urldefrag()` que contienen datos *bytes*. El método `decode()` retorna una instancia *DefragResult*.

Nuevo en la versión 3.2.

class `urllib.parse.ParseResultBytes` (*scheme, netloc, path, params, query, fragment*)

Clase concreta para los resultados `urlparse()` que contienen datos *bytes*. El método `decode()` retorna una instancia *ParseResult*.

Nuevo en la versión 3.2.

class `urllib.parse.SplitResultBytes` (*scheme, netloc, path, query, fragment*)

Clase concreta para los resultados de `urlsplit()` que contienen datos *bytes*. El método `decode()` retorna una instancia *SplitResult*.

Nuevo en la versión 3.2.

21.8.5 Cita de URL

Las funciones de citación de URL se centran en tomar datos del programa y hacerlos seguros para su uso como componentes URL citando caracteres especiales y codificando adecuadamente texto no ASCII. También admiten la inversión de estas operaciones para volver a crear los datos originales a partir del contenido de un componente de dirección URL si esa tarea aún no está cubierta por las funciones de análisis de URL anteriores.

`urllib.parse.quote` (*string, safe='/', encoding=None, errors=None*)

Reemplace caracteres especiales en *string* con el escape `%xx`. Las letras, los dígitos y los caracteres `'_ . - ~'` nunca se citan. De forma predeterminada, esta función está pensada para citar la sección de ruta de acceso de la dirección URL. El parámetro opcional *safe* especifica caracteres ASCII adicionales que no se deben citar — su valor predeterminado es `'/'`.

string puede ser *str* o *bytes*.

Distinto en la versión 3.7: Moved from **RFC 2396** to **RFC 3986** for quoting URL strings. `<~>` is now included in the set of unreserved characters.

Los parámetros opcionales *encoding* y *errors* especifican cómo tratar con caracteres no ASCII, tal como lo acepta el método `str.encode()`. *encoding* por defecto es `'utf-8'`. *errors* tiene como valor predeterminado `'strict'`, lo que significa que los caracteres no admitidos generan un `UnicodeEncodeError`. *encoding* y *errors* no se deben proporcionar si *string* es `bytes` o se genera `TypeError`.

Tenga en cuenta que `quote(string, safe, encoding, errors)` es equivalente a `quote_from_bytes(string.encode(encoding, errors), safe)`.

Ejemplo: `quote('/El Niño/')` produce `'/El%20Ni%C3%B1o/'`.

`urllib.parse.quote_plus(string, safe="", encoding=None, errors=None)`

Como `quote()`, pero también reemplaza los espacios por signos más, según sea necesario para citar valores de formulario HTML al crear una cadena de consulta para ir a una dirección URL. Además, los signos de la cadena original se escapan a menos que se incluyan en *safe*. Tampoco tiene *safe* predeterminado para `'/'`.

Ejemplo: `quote_plus('/El Niño/')` produce `'%2FE1+Ni%C3%B1o%2F'`.

`urllib.parse.quote_from_bytes(bytes, safe='')`

Como `quote()`, pero acepta un objeto `bytes` en lugar de un `str`, y no realiza la codificación de cadena a bytes.

Ejemplo: `quote_from_bytes(b'a&\xef')` produce `'a%26%EF'`.

`urllib.parse.unquote(string, encoding='utf-8', errors='replace')`

Reemplace los escapes `%xx` por su equivalente de un solo carácter. Los parámetros opcionales *encoding* y *errors* especifican cómo decodificar secuencias codificadas porcientualmente en caracteres Unicode, tal como lo acepta el método `bytes.decode()`.

string debe ser `str`.

encoding por defecto es `'utf-8'`. *errors* por defecto es `'replace'`, lo que significa que las secuencias no válidas se reemplazan por un carácter de marcador de posición.

Ejemplo: `unquote('/El%20Ni%C3%B1o/')` produce `'/El Niño/'`.

`urllib.parse.unquote_plus(string, encoding='utf-8', errors='replace')`

Como `unquote()`, pero también reemplaza los signos más por espacios, según sea necesario para los valores de formulario HTML sin comillas.

string debe ser `str`.

Ejemplo: `unquote_plus('/El+Ni%C3%B1o/')` produce `'/El Niño/'`.

`urllib.parse.unquote_to_bytes(string)`

Reemplace los escapes `%xx` por su equivalente de un solo octeto y retorne un objeto `bytes`.

string puede ser `str` o `bytes`.

Si es un `str`, los caracteres no ASCII sin escapar en *string* se codifican en bytes UTF-8.

Ejemplo: `unquote_to_bytes('a%26%EF')` produce `b'a&\xef'`.

`urllib.parse.urlencode(query, doseq=False, safe="", encoding=None, errors=None, quote_via=quote_plus)`

Convierta un objeto de asignación o una secuencia de tuplas de dos elementos, que pueden contener objetos `str` o `bytes`, en una cadena de texto ASCII codificada porcientualmente. Si la cadena resultante se va a utilizar como una operación *data* para post con la función `urllib.request.urlopen()`, entonces debe codificarse en bytes, de lo contrario resultaría en un `TypeError`.

La cadena resultante es una serie de pares *key=value* separados por caracteres `'&'`, donde tanto *key* como *value* se citan mediante la función *quote_via*. De forma predeterminada, `quote_plus()` se utiliza para citar los valores, lo que significa que los espacios se citan como un carácter `'+'` y `'/'` los caracteres se codifican como `%2F`, que sigue el estándar para las solicitudes GET (`application/x-www-form-urlencoded`). Una función alternativa que se puede pasar como *quote_via* es `quote()`, que codificará espacios como `%20` y no codificará caracteres `'/'`. Para obtener el máximo control de lo que se cita, utilice `quote` y especifique un valor para *safe*.

Cuando se utiliza una secuencia de tuplas de dos elementos como argumento *query*, el primer elemento de cada tupla es una clave y el segundo es un valor. El elemento *value* en sí mismo puede ser una secuencia y, en ese caso, si el parámetro opcional *doseq* se evalúa como `True`, se generan pares individuales *key=value* separados por `'&'` para cada elemento de la secuencia de valores de la clave. El orden de los parámetros de la cadena codificada coincidirá con el orden de las tuplas de parámetros de la secuencia.

Los parámetros *safe*, *encoding* y *errors* se pasan a *quote_via* (los parámetros *encoding* y *errors* solo se pasan cuando un elemento de consulta es *str*).

Para revertir este proceso de codificación, en este módulo se proporcionan *parse_qs()* y *parse_qsl()* para analizar cadenas de consulta en estructuras de datos de Python.

Consulte [urllib examples](#) para averiguar cómo se puede utilizar el método `urlencode` para generar una cadena de consulta para una dirección URL o datos para POST.</urllib-examples>

Distinto en la versión 3.2: El parámetro *Query* admite bytes y objetos de cadena.

Nuevo en la versión 3.5: *quote_via*.

Ver también:

WHATWG - URL Living standard Working Group for the URL Standard that defines URLs, domains, IP addresses, the application/x-www-form-urlencoded format, and their API.

RFC 3986 - Identificadores uniformes de recursos Este es el estándar actual (STD66). Cualquier cambio en el módulo `urllib.parse` debe ajustarse a esto. Se podrían observar ciertas desviaciones, que son principalmente para fines de compatibilidad con versiones anteriores y para ciertos requisitos de análisis de facto como se observa comúnmente en los principales navegadores.

RFC 2732 - Formato de direcciones IPv6 literales en URL. Esto especifica los requisitos de análisis de las direcciones URL IPv6.

RFC 2396 - Identificadores uniformes de recursos (URI): Sintaxis genérica Documento que describe los requisitos sintácticos genéricos para los nombres de recursos uniformes (URL) y los localizadores uniformes de recursos (URL).

RFC 2368 - El esquema mailto URL. Análisis de requisitos para esquemas de URL de correo a correo.

RFC 1808 - Localizadores uniformes de recursos relativos Esta solicitud de comentarios incluye las reglas para unirse a una URL absoluta y relativa, incluyendo un buen número de «Ejemplos anormales» que rigen el tratamiento de los casos fronterizos.

RFC 1738 - Localizadores uniformes de recursos (URL) Esto especifica la sintaxis formal y la semántica de las direcciones URL absolutas.

21.9 urllib.error — Clases de excepción lanzadas por urllib.request

Código fuente: [Lib/urllib/error.py](#)

El módulo `urllib.error` define las clases de excepción para las excepciones lanzadas por `urllib.request`. La clase de excepción base es `URLError`.

Las siguientes excepciones son lanzadas por `urllib.error` según sea apropiado:

exception `urllib.error.URLError`

Los gestores lanzan esta excepción (o excepciones derivadas) cuando encuentran un problema. Es una subclase de `OSError`.

reason

El motivo de este error. Puede ser una cadena de mensaje u otra instancia de una excepción.

Distinto en la versión 3.3: Se ha convertido a `URLError` en una subclase de `OSError` en lugar de `IOError`.

exception `urllib.error.HTTPError`

A pesar de ser una excepción (una subclase de `URLError`), un `HTTPError` también puede funcionar como un valor de retorno no excepcional de tipo archivo (lo mismo que retorna `urlopen()`). Esto es útil para gestionar errores HTTP exóticos, como peticiones de autenticación.

code

Un código de estado HTTP como los definidos en **RFC 2616**. Este valor numérico se corresponde con un valor de un diccionario de códigos como el que hay en `http.server.BaseHTTPRequestHandler.responses`.

reason

Normalmente esto es una cadena de caracteres que explica el motivo de este error.

headers

Las cabeceras de la respuesta HTTP de la petición HTTP que causó el `HTTPError`.

Nuevo en la versión 3.4.

exception `urllib.error.ContentTooShortError` (*msg, content*)

Esta excepción se lanza cuando la función `urlretrieve()` detecta que la cantidad de datos descargados es menor que la esperada (dada por la cabecera `Content-Length`). El atributo `content` almacena los datos descargados (y supuestamente truncados).

21.10 urllib.robotparser — Analizador para robots.txt

Código fuente: [Lib/urllib/robotparser.py](#)

Este módulo proporciona una sola clase, `RobotFileParser`, la cual responde preguntas acerca de si un agente de usuario en particular puede o no obtener una URL en el sitio Web que publico el archivo `robots.txt`. Para más detalles sobre la estructura del archivo `robots.txt`, consulte <http://www.robotstxt.org/orig.html>.

class `urllib.robotparser.RobotFileParser` (*url=""*)

Esta clase proporciona métodos para leer, analizar y responder preguntas acerca de `robots.txt`

set_url (*url*)

Establece la URL que hace referencia a un archivo `robots.txt`.

read ()

Lee la URL `robots.txt` y la envía al analizador.

parse (*lines*)

Analiza el argumento *lines*.

can_fetch (*useragent, url*)

Retorna `True` si el *useragent* tiene permiso para buscar la *url* de acuerdo con las reglas contenidas en el archivo `robots.txt` analizado.

mtime ()

Retorna la hora en que se recuperó por última vez el archivo `robots.txt`. Esto es útil para arañas web de larga duración que necesitan buscar nuevos archivos `robots.txt` periódicamente.

modified ()

Establece la hora a la que se recuperó por última vez el archivo `robots.txt` hasta la hora actual.

crawl_delay (*useragent*)

Retorna el valor del parámetro `Crawl-delay` de `robots.txt` para el *useragent* en cuestión. Si no existe tal parámetro o no se aplica al *useragent* especificado o la entrada `robots.txt` para este parámetro tiene una sintaxis no válida, devuelve `None`.

Nuevo en la versión 3.6.

request_rate (*useragent*)

Retorna el contenido del parámetro `Request-rate` de `robots.txt` como una *tupla nombrada* `RequestRate(requests, seconds)`. Si no existe tal parámetro o no se aplica al *useragent* especificado o la entrada `robots.txt` para este parámetro tiene una sintaxis no válida, devuelve `None`.

Nuevo en la versión 3.6.

site_maps ()

Retorna el contenido del parámetro `Sitemap` de `robots.txt` en forma de *list()*. Si no existe tal parámetro o la entrada `robots.txt` para este parámetro tiene una sintaxis no válida, devuelve `None`.

Nuevo en la versión 3.8.

El siguiente ejemplo demuestra el uso básico de la clase *RobotFileParser*:

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("")
6
>>> rp.can_fetch("", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("", "http://www.musi-cal.com/")
True
```

21.11 http — Módulos HTTP

Código fuente: `Lib/http/__init__.py`

http es un paquete que recopila varios módulos para trabajar con el Protocolo de transferencia de hipertexto:

- *http.client* es un cliente del protocolo HTTP de bajo nivel; para la apertura de URL de alto nivel use *urllib.request*
- *http.server* contiene clases de servidor HTTP básicas basadas en *socketserver*
- *http.cookies* tiene utilidades para implementar la gestión de estados mediante cookies
- *http.cookiejar* provee persistencia de cookies

http es también un módulo que define una serie de códigos de estado HTTP y mensajes asociados a través de la enumeración *http.HTTPStatus*:

class `http.HTTPStatus`

Nuevo en la versión 3.5.

Una subclase de *enum.IntEnum* que define un conjunto de códigos de estado HTTP, frases de motivo y descripciones largas escritas en inglés.

Uso:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
<HTTPStatus.OK: 200>
>>> HTTPStatus.OK == 200
True
```

(continué en la próxima página)

(proviene de la página anterior)

```

>>> HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[<HTTPStatus.CONTINUE: 100>, <HTTPStatus.SWITCHING_PROTOCOLS: 101>, ...]

```

21.11.1 Códigos de estado HTTP

Los códigos de estado [registrados por IANA](#) soportados y disponibles en `http.HTTPStatus` son:

Código	Nombre de la enumeración	Detalle
100	CONTINUE	HTTP/1.1 RFC 7231 , Sección 6.2.1
101	SWITCHING_PROTOCOLS	HTTP/1.1 RFC 7231 , Sección 6.2.2
102	PROCESSING	WebDAV RFC 2518 , Sección 10.1
200	OK	HTTP/1.1 RFC 7231 , Sección 6.3.1
201	CREATED	HTTP/1.1 RFC 7231 , Sección 6.3.2
202	ACCEPTED	HTTP/1.1 RFC 7231 , Sección 6.3.3
203	NON_AUTHORITATIVE_INFORMATION	HTTP/1.1 RFC 7231 , Sección 6.3.4
204	NO_CONTENT	HTTP/1.1 RFC 7231 , Sección 6.3.5
205	RESET_CONTENT	HTTP/1.1 RFC 7231 , Sección 6.3.6
206	PARTIAL_CONTENT	HTTP/1.1 RFC 7233 , Sección 4.1
207	MULTI_STATUS	WebDAV RFC 4918 , Sección 11.1
208	ALREADY_REPORTED	Extensiones de enlace a WebDAV RFC 5842 , Sección 7.1 (Experimental)
226	IM_USED	Codificación delta en HTTP RFC 3229 , Sección 10.4.1
300	MULTIPLE_CHOICES	HTTP/1.1 RFC 7231 , Sección 6.4.1
301	MOVED_PERMANENTLY	HTTP/1.1 RFC 7231 , Sección 6.4.2
302	FOUND	HTTP/1.1 RFC 7231 , Sección 6.4.3
303	SEE_OTHER	HTTP/1.1 RFC 7231 , Sección 6.4.4
304	NOT_MODIFIED	HTTP/1.1 RFC 7232 , Sección 4.1
305	USE_PROXY	HTTP/1.1 RFC 7231 , Sección 6.4.5
307	TEMPORARY_REDIRECT	HTTP/1.1 RFC 7231 , Sección 6.4.7
308	PERMANENT_REDIRECT	Redirección permanente RFC 7238 , Sección 3 (Experimental)
400	BAD_REQUEST	HTTP/1.1 RFC 7231 , Sección 6.5.1
401	UNAUTHORIZED	Autenticación HTTP/1.1 RFC 7235 , Sección 3.1
402	PAYMENT_REQUIRED	HTTP/1.1 RFC 7231 , Sección 6.5.2
403	FORBIDDEN	HTTP/1.1 RFC 7231 , Sección 6.5.3
404	NOT_FOUND	HTTP/1.1 RFC 7231 , Sección 6.5.4
405	METHOD_NOT_ALLOWED	HTTP/1.1 RFC 7231 , Sección 6.5.5
406	NOT_ACCEPTABLE	HTTP/1.1 RFC 7231 , Sección 6.5.6
407	PROXY_AUTHENTICATION_REQUIRED	Autenticación HTTP/1.1 RFC 7235 , Sección 3.2
408	REQUEST_TIMEOUT	HTTP/1.1 RFC 7231 , Sección 6.5.7
409	CONFLICT	HTTP/1.1 RFC 7231 , Sección 6.5.8
410	GONE	HTTP/1.1 RFC 7231 , Sección 6.5.9
411	LENGTH_REQUIRED	HTTP/1.1 RFC 7231 , Sección 6.5.10
412	PRECONDITION_FAILED	HTTP/1.1 RFC 7232 , Sección 4.2
413	REQUEST_ENTITY_TOO_LARGE	HTTP/1.1 RFC 7231 , Sección 6.5.11
414	REQUEST_URI_TOO_LONG	HTTP/1.1 RFC 7231 , Sección 6.5.12
415	UNSUPPORTED_MEDIA_TYPE	HTTP/1.1 RFC 7231 , Sección 6.5.13
416	REQUESTED_RANGE_NOT_SATISFIABLE	Rango de solicitudes HTTP/1.1 RFC 7233 , Sección 4.4
417	EXPECTATION_FAILED	HTTP/1.1 RFC 7231 , Sección 6.5.14

Continúa en la

Tabla 1 – proviene de la página anterior

Código	Nombre de la enumeración	Detalle
421	MISDIRECTED_REQUEST	HTTP/2 RFC 7540 , Sección 9.1.2
422	UNPROCESSABLE_ENTITY	WebDAV RFC 4918 , Sección 11.2
423	LOCKED	WebDAV RFC 4918 , Sección 11.3
424	FAILED_DEPENDENCY	WebDAV RFC 4918 , Sección 11.4
426	UPGRADE_REQUIRED	HTTP/1.1 RFC 7231 , Sección 6.5.15
428	PRECONDITION_REQUIRED	Códigos de estados HTTP adicionales RFC 6585
429	TOO_MANY_REQUESTS	Códigos de estados HTTP adicionales RFC 6585
431	REQUEST_HEADER_FIELDS_TOO_LARGE	Códigos de estados HTTP adicionales RFC 6585
451	UNAVAILABLE_FOR_LEGAL_REASONS	Un código de estado HTTP para reportar obstáculos legales RFC 7725
500	INTERNAL_SERVER_ERROR	HTTP/1.1 RFC 7231 , Sección 6.6.1
501	NOT_IMPLEMENTED	HTTP/1.1 RFC 7231 , Sección 6.6.2
502	BAD_GATEWAY	HTTP/1.1 RFC 7231 , Sección 6.6.3
503	SERVICE_UNAVAILABLE	HTTP/1.1 RFC 7231 , Sección 6.6.4
504	GATEWAY_TIMEOUT	HTTP/1.1 RFC 7231 , Sección 6.6.5
505	HTTP_VERSION_NOT_SUPPORTED	HTTP/1.1 RFC 7231 , Sección 6.6.6
506	VARIANT_ALSO_NEGOTIATES	Negociación transparente de contenido en HTTP RFC 2295 , Sección 6
507	INSUFFICIENT_STORAGE	WebDAV RFC 4918 , Sección 11.5
508	LOOP_DETECTED	Extensiones de unión WebDAV RFC 5842 , Sección 7.2 (Experimental)
510	NOT_EXTENDED	Un framework de extensión HTTP RFC 2774 , Sección 7 (Experimental)
511	NETWORK_AUTHENTICATION_REQUIRED	Códigos de estados HTTP adicionales RFC 6585 , Sección 6

Con el fin de preservar la compatibilidad con versiones anteriores, los valores de la enumeración están también presentes en el módulo `http.client` en forma de constantes. El nombre de la enumeración es el mismo que el nombre de la constante (ej. `http.HTTPStatus.OK` se encuentra también disponible como `http.client.OK`).

Distinto en la versión 3.7: Se agregó el código de estado 421 `MISDIRECTED_REQUEST`.

Nuevo en la versión 3.8: Se agregó el código de estado 451 `UNAVAILABLE_FOR_LEGAL_REASONS`.

21.12 `http.client` — Cliente de protocolo HTTP

Código fuente: [Lib/http/client.py](#)

Este módulo define clases que se implementan del lado del cliente de los protocolos HTTP y HTTPS. Normalmente no se usa directamente — el módulo `urllib.request` lo usa para gestionar URLs que usan HTTP y HTTPS.

Ver también:

El [Paquete de solicitudes](#) se recomienda para una interfaz de cliente HTTP de alto nivel.

Nota: El soporte HTTPS solo está disponible si Python se compiló con soporte SSL (a través del módulo `ssl`).

El módulo proporciona las siguientes clases:

```
class http.client.HTTPConnection (host, port=None[, timeout], source_address=None, blocksize=8192)
```

Una instancia `HTTPConnection` representa una transacción con un servidor HTTP. Se debe instanciar pasándole un host y un número de puerto opcional. Si no se pasa ningún número de puerto, el puerto se extrae de la cadena de host si tiene la forma `host:port`; de lo contrario, se utiliza el puerto HTTP predeterminado (80). Si se proporciona el parámetro opcional `timeout`, las operaciones de bloqueo (como los intentos de conexión) expirarán después de esos segundos (si no se proporciona, se usa la configuración de tiempo de espera global predeterminada). El parámetro opcional `source_address` puede ser una tupla de un (host, puerto) para

usar como la dirección de origen desde la que se realiza la conexión HTTP. El parámetro opcional *blocksize* establece el tamaño del búfer en bytes para enviar un cuerpo de mensaje similar a un archivo.

Por ejemplo, las siguientes llamadas crean instancias que se conectan al servidor en el mismo host y puerto:

```
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=10)
```

Distinto en la versión 3.2: *source_address* fue adicionado.

Distinto en la versión 3.4: Se eliminó el argumento *strict*. Las «Respuestas Simples» de estilo HTTP 0.9 ya no son compatibles.

Distinto en la versión 3.7: argumento *blocksize* fue adicionado.

```
class http.client.HTTPSConnection(host, port=None, key_file=None, cert_file=None[,
                                     timeout], source_address=None, *, context=None,
                                     check_hostname=None, blocksize=8192)
```

Una subclase de *HTTPConnection* que usa SSL para la comunicación con servidores seguros. El puerto predeterminado es 443. Si se especifica *context*, debe ser una instancia de *ssl.SSLContext* que describa las diversas opciones de SSL.

Por favor lea *Security considerations* para obtener más información sobre las mejores prácticas.

Distinto en la versión 3.2: *source_address*, *context* y *check_hostname* fueron adicionados.

Distinto en la versión 3.2: Esta clase ahora soporta *hosts* virtuales HTTPS si es posible (es decir, si *ssl.HAS_SNI* es verdadero).

Distinto en la versión 3.4: Se eliminó el argumento *strict*. Las «Respuestas Simples» de estilo HTTP 0.9 ya no son compatibles.

Distinto en la versión 3.4.3: Esta clase ahora realiza todas las comprobaciones necesarias de certificados y nombres de host de forma predeterminada. Para volver al comportamiento anterior no verificado *ssl._create_unverified_context()* se puede pasar al argumento *context*.

Distinto en la versión 3.8: Esta clase ahora habilita TLS 1.3 *ssl.SSLContext.post_handshake_auth* para el *context* predeterminado o cuando *cert_file* se pasa con un *context* personalizado.

Obsoleto desde la versión 3.6: *key_file* y *cert_file* están discontinuadas en favor de *context*. Por favor use *ssl.SSLContext.load_cert_chain()* en su lugar, o deje que *ssl.create_default_context()* seleccione los certificados de CA de confianza del sistema para usted.

El argumento *check_hostname* también está discontinuado; el atributo *ssl.SSLContext.check_hostname* de *context* debe usarse en su lugar.

```
class http.client.HTTPResponse(sock, debuglevel=0, method=None, url=None)
```

Clase cuyas instancias se retornan tras una conexión exitosa. No son instancias realizadas directamente por el usuario.

Distinto en la versión 3.4: Se eliminó el argumento *strict*. Las «Respuestas Simples» del estilo HTTP 0.9 ya no están soportadas.

Este módulo proporciona la siguiente función:

```
http.client.parse_headers(fp)
```

Analiza los encabezados desde un puntero de archivo *fp* que representa una solicitud/respuesta HTTP. El archivo debe ser un lector *BufferedIOBase* (es decir, no texto) y debe proporcionar un encabezado de estilo válido **RFC 2822**.

Esta función retorna una instancia de *http.client.HTTPMessage* que contiene los campos de encabezado, pero no un *payload* (lo mismo que *HTTPResponse.msg* y *http.server.BaseHTTPRequestHandler.headers*) Después de regresar, el puntero de archivo *fp* está listo para leer el cuerpo HTTP.

Nota: `parse_headers()` no analiza la línea de inicio de un mensaje HTTP; solo analiza las líneas `Name: value`. El archivo tiene que estar listo para leer estas líneas de campo, por lo que la primera línea ya debe consumirse antes de llamar a la función.

Las siguientes excepciones son lanzadas según corresponda:

exception `http.client.HTTPException`

La clase base de las otras excepciones en este módulo. Es una subclase de `Exception`.

exception `http.client.NotConnected`

Una subclase de `HTTPException`.

exception `http.client.InvalidURL`

Una subclase de `HTTPException`, es lanzada si se proporciona un puerto y no es numérico o está vacío.

exception `http.client.UnknownProtocol`

Una subclase de `HTTPException`.

exception `http.client.UnknownTransferEncoding`

Una subclase de `HTTPException`.

exception `http.client.UnimplementedFileMode`

Una subclase de `HTTPException`.

exception `http.client.IncompleteRead`

Una subclase de `HTTPException`.

exception `http.client.ImproperConnectionState`

Una subclase de `HTTPException`.

exception `http.client.CannotSendRequest`

Una subclase de `ImproperConnectionState`.

exception `http.client.CannotSendHeader`

Una subclase de `ImproperConnectionState`.

exception `http.client.ResponseNotReady`

Una subclase de `ImproperConnectionState`.

exception `http.client.BadStatusLine`

Una subclase de `HTTPException`. Es lanzada si un servidor responde con un código de estado HTTP que no entendemos.

exception `http.client.LineTooLong`

Una subclase de `HTTPException`. Es lanzada si se recibe una línea excesivamente larga en el protocolo HTTP del servidor.

exception `http.client.RemoteDisconnected`

Una subclase de `ConnectionResetError` y `BadStatusLine`. Lanzada por `HTTPConnection.getresponse()` cuando el intento de leer la respuesta no produce datos leídos de la conexión, indica que el extremo remoto ha cerrado la conexión.

Nuevo en la versión 3.5: Previamente, `BadStatusLine('')` fue lanzada.

Las constantes definidas en este módulo son:

`http.client.HTTP_PORT`

El puerto predeterminado para el protocolo HTTP (siempre 80).

`http.client.HTTPS_PORT`

El puerto predeterminado para el protocolo HTTPS (siempre 443).

`http.client.responses`

Este diccionario asigna los códigos de estado HTTP 1.1 a los nombres W3C.

Ejemplo: `http.client.responses[http.client.NOT_FOUND]` es `'Not Found'`.

Consulte *Códigos de estado HTTP* para obtener una lista de los códigos de estado HTTP que están disponibles en este módulo como constantes.

21.12.1 Objetos de `HTTPConnection`

Las instancias `HTTPConnection` tienen los siguientes métodos:

`HTTPConnection.request(method, url, body=None, headers={}, *, encode_chunked=False)`

Esto enviará una solicitud al servidor utilizando el método de solicitud HTTP *method* y el selector *url*.

Si se especifica *body*, los datos especificados se envían una vez finalizados los encabezados. Puede ser *str*, un objeto *bytes-like object*, un objeto *file object* abierto, o un iterable de *bytes*. Si *body* es una cadena, se codifica como ISO-8859-1, el valor predeterminado para HTTP. Si es un objeto de tipo bytes, los bytes se envían tal cual. Si es un objeto *file object*, se envía el contenido del archivo; Este objeto de archivo debe soportar al menos el método `read()`. Si el objeto de archivo es una instancia de `io.TextIOBase`, los datos retornados por el método `read()` se codificarán como ISO-8859-1, de lo contrario, los datos retornados por `read()` se envía como está. Si *body* es un iterable, los elementos del iterable se envían tal cual hasta que se agota el iterable.

El argumento *headers* debe ser un mapeo de encabezados HTTP extras para enviar con la solicitud.

Si *headers* no contiene `Content-Length` ni `Transfer-Encoding`, pero hay un cuerpo de solicitud, uno de esos campos de encabezado se agregará automáticamente. Si *body* es `None`, el encabezado `Content-Length` se establece en 0 para los métodos que esperan un cuerpo (`PUT`, `POST` y `PATCH`). Si *body* es una cadena de caracteres o un objeto similar a bytes que no es también un *file*, el encabezado `Content-Length` se establece en su longitud. Cualquier otro tipo de *body* (archivos e iterables en general) se codificará en fragmentos, y el encabezado `Transfer-Encoding` se establecerá automáticamente en lugar de `Content-Length`.

El argumento *encode_chunked* solo es relevante si `Transfer-Encoding` se especifica en *headers*. Si *encode_chunked* es `False`, el objeto `HTTPConnection` supone que toda la codificación es manejada por el código de llamada. Si es `True`, el cuerpo estará codificado en fragmentos.

Nota: La codificación de transferencia fragmentada se ha agregado al protocolo HTTP versión 1.1. A menos que se sepa que el servidor HTTP maneja HTTP 1.1, el llamador debe especificar la longitud del contenido o debe pasar un *str* o un objeto similar a bytes que no sea también un archivo como la representación del cuerpo.

Nuevo en la versión 3.2: *body* ahora puede ser un iterable.

Distinto en la versión 3.6: Si ni `Content-Length` ni `Transfer-Encoding` están configurados en *headers*, el archivo y los objetos iterables *body* ahora están codificados en fragmentos. Se agregó el argumento *encode_chunked*. No se intenta determinar la longitud del contenido para los objetos de archivo.

`HTTPConnection.getresponse()`

Debe llamarse después de enviar una solicitud para obtener la respuesta del servidor. Retorna una instancia de `HTTPResponse`.

Nota: Tenga en cuenta que debe haber leído la respuesta completa antes de poder enviar una nueva solicitud al servidor.

Distinto en la versión 3.5: Si una `ConnectionError` o una subclase fue lanzada, el objeto `HTTPConnection` estará listo para volver a conectarse cuando se envíe una nueva solicitud.

`HTTPConnection.set_debuglevel(level)`

Establecer el nivel de depuración. El nivel de depuración predeterminado es 0, lo que significa que no se imprime ninguna salida de depuración. Cualquier valor mayor que 0 hará que todos los resultados de depuración definidos actualmente se impriman en *stdout*. El `debuglevel` se pasa a cualquier objeto nuevo `HTTPResponse` que se cree.

Nuevo en la versión 3.1.

`HTTPConnection.set_tunnel (host, port=None, headers=None)`

Configure el host y el puerto para el túnel de conexión HTTP. Esto permite ejecutar la conexión a través de un servidor proxy.

Los argumentos de host y puerto especifican el punto final de la conexión realizada por el túnel (es decir, la dirección incluida en la solicitud CONNECT, da *not* la dirección del servidor proxy).

El argumento de los encabezados debe ser un mapeo de encabezados HTTP adicionales para enviar con la solicitud CONNECT.

Por ejemplo, para hacer un túnel a través de un servidor proxy HTTPS que se ejecuta localmente en el puerto 8080, pasaríamos la dirección del proxy al constructor `HTTPSConnection`, y la dirección del host al que finalmente queremos llegar al método `set_tunnel()`:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD", "/index.html")
```

Nuevo en la versión 3.2.

`HTTPConnection.connect ()`

Se conecta al servidor especificado cuando el objeto fue creado. Por defecto, esto se llama automáticamente cuando se realiza una solicitud si el cliente aún no tiene una conexión.

`HTTPConnection.close ()`

Cierre la conexión al servidor.

`HTTPConnection.blocksize`

Tamaño del búfer en bytes para enviar un archivo como cuerpo del mensaje.

Nuevo en la versión 3.7.

Como alternativa al uso del método `request ()` descrito anteriormente, también puede enviar su solicitud paso a paso, utilizando las cuatro funciones a continuación.

`HTTPConnection.putrequest (method, url, skip_host=False, skip_accept_encoding=False)`

Esta debería ser la primera llamada después de que se haya realizado la conexión al servidor. Envía una línea al servidor que consta de la cadena de caracteres *method*, la cadena de caracteres *url* y la versión HTTP (HTTP / 1.1). Para deshabilitar el envío automático de encabezados "Host:" o `Accept-Encoding:` (por ejemplo, para aceptar codificaciones de contenido adicionales), especifique *skip_host* o *skip_accept_encoding* con valores no Falsos.

`HTTPConnection.putheader (header, argument[, ...])`

Envía un encabezado [RFC 822](#)-style al servidor. Este envía una línea al servidor que consta del encabezado, dos puntos y un espacio, y el primer argumento. Si se dan más argumentos, se envían líneas de continuación, cada una de las cuales consta de tabulación y un argumento.

`HTTPConnection.endheaders (message_body=None, *, encode_chunked=False)`

Envía una línea en blanco al servidor, señalando el final de los encabezados. El argumento opcional *message_body* se puede usar para pasar un cuerpo de mensaje asociado a la solicitud.

Si *encode_chunked* es `True`, el resultado de cada iteración de *message_body* se codificará en fragmentos como se especifica en [RFC 7230](#), Sección 3.3.1. La forma en que se codifican los datos depende del tipo de *message_body*. Si *message_body* implementa `buffer interface` la codificación dará como resultado un solo fragmento. Si *message_body* es una `collections.abc.Iterable`, cada iteración de *message_body* dará como resultado un fragmento. Si *message_body* es un objeto *file object*, cada llamada a `.read()` dará como resultado un fragmento. El método señala automáticamente el final de los datos codificados en fragmentos inmediatamente después de *message_body*.

Nota: Debido a la especificación de codificación fragmentada, fragmentos vacíos producidos por un cuerpo iterador será ignorado por el codificador de fragmentos. Esto es para evitar la terminación prematura de la lectura de la solicitud por parte del servidor de destino debido a una codificación con formato incorrecto.

Nuevo en la versión 3.6: Soporte de codificación fragmentada. Se agregó el parámetro *encode_chunked*.

`HTTPConnection.send(data)`

Envía datos al servidor. Esto debe usarse directamente solo después de que se haya llamado al método *endheaders()* y antes de que se llame al método *getresponse()*.

21.12.2 Objetos de `HTTPResponse`

Una instancia de *HTTPResponse* envuelve la respuesta HTTP del servidor. Proporciona acceso a los encabezados de la solicitud y al cuerpo de la entidad. La respuesta es un objeto iterable y puede usarse en una declaración *with*.

Distinto en la versión 3.5: La interfaz *io.BufferedIOBase* ahora está implementada y todas sus operaciones de lectura están soportadas.

`HTTPResponse.read([amt])`

Lee y retorna el cuerpo de respuesta, o hasta los siguientes bytes *amt*.

`HTTPResponse.readinto(b)`

Lee hasta los siguientes bytes *len(b)* del cuerpo de respuesta en el búfer *b*. Retorna el número de bytes leídos.

Nuevo en la versión 3.3.

`HTTPResponse.getheader(name, default=None)`

Retorna el valor del encabezado *name* o *default* si no hay un encabezado que coincida con *name*. Si hay más de un encabezado con el nombre *name*, retorne todos los valores unidos por “,”. Si es “default” es cualquier iterable que no sea una sola cadena de caracteres, sus elementos se retornan de manera similar unidos por comas.

`HTTPResponse.getheaders()`

Retorna una lista de tuplas (encabezado, valor).

`HTTPResponse.fileno()`

Retorna el fileno del socket implícito.

`HTTPResponse.msg`

Una instancia *http.client.HTTPMessage* que contiene los encabezados de respuesta. *http.client.HTTPMessage* es una subclase de *email.message.Message*.

`HTTPResponse.version`

Versión del protocolo HTTP utilizada por el servidor. 10 para HTTP/1.0, 11 para HTTP/1.1.

`HTTPResponse.status`

Código del estado retornado por el servidor.

`HTTPResponse.reason`

Una frase de la razón es retornado por el servidor.

`HTTPResponse.debuglevel`

Un depurador. Si *debuglevel* es mayor que cero, los mensajes se imprimirán en *stdout* a medida que se lee y analiza la respuesta.

`HTTPResponse.closed`

Es *True* si la transmisión está cerrada.

21.12.3 Ejemplos

Aquí hay una sesión de ejemplo que usa el método GET *method*:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read() # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while chunk := r1.read(200):
...     print(repr(chunk))
b'<!doctype html>\n<!--[if"...
...
>>> # Example of an invalid request
>>> conn = http.client.HTTPSConnection("docs.python.org")
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Aquí hay una sesión de ejemplo que usa el método HEAD. Tenga en cuenta que el método HEAD nunca retorna ningún dato.

```
>>> import http.client
>>> conn = http.client.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 == b''
True
```

Aquí hay una sesión de ejemplo que muestra cómo solicitar POST:

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action': 'show'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...           "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/issue12524</a>'
>>> conn.close()
```

Las solicitudes HTTP PUT del lado del cliente son muy similares a las solicitudes POST. La diferencia radica solo en el lado del servidor donde el servidor HTTP permitirá que se creen recursos a través de la solicitud PUT. Cabe señalar que los métodos HTTP personalizados también se manejan en `urllib.request.Request` configurando el

atributo de método apropiado. Aquí hay una sesión de ejemplo que muestra cómo enviar una solicitud `PUT` utilizando `http.client`:

```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/file
...
>>> import http.client
>>> BODY = """filecontents"""
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

21.12.4 Objetos de `HTTPMessage`

Una instancia de `http.client.HTTPMessage` contiene los encabezados de una respuesta HTTP. Se implementa utilizando la clase `email.message.Message`.

21.13 `ftplib` — cliente de protocolo FTP

Código fuente [Lib/ftplib.py](#)

Este módulo define la clase `FTP` y algunos elementos relacionados. La clase `FTP` implementa el lado cliente del protocolo FTP. Puedes usarlo para escribir programas en Python que realizan una variedad de trabajos FTP automatizados, como reflejar otros servidores FTP. También es utilizado por el módulo `urllib.request` para manejar URLs que usan FTP. Para más información sobre FTP (Protocolo de transferencia de archivos), véase Internet [RFC 959](#).

Aquí hay una sesión de ejemplo usando el módulo `ftplib`:

```
>>> 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.'
>>> with open('README', 'wb') as fp:
>>>     ftp.retrbinary('RETR README', fp.write)
'226 Transfer complete.'
>>> ftp.quit()
```

El módulo define los siguientes elementos:

class `ftplib.FTP` (*host*="", *user*="", *passwd*="", *acct*="", *timeout*=None, *source_address*=None)

Retorna una instancia nueva de la clase `FTP`. Cuando se proporciona *host*, se llama al método `connect(host)`. Cuando se proporciona *user*, se llama adicionalmente al método `login(user, passwd, acct)` (donde *passwd* y *acct* tienen como valor predeterminado la cadena vacía cuando no se indica). El parámetro opcional *timeout* especifica un tiempo de espera en segundo para bloquear operaciones

como el intento de conexión (si no está especificado, se usa el tiempo de espera global por defecto). *source_address* es una tupla de 2 (*host*, *port*) para que el *socket* se enlace como su dirección de origen antes de conectarse.

La clase `FTP` admite la instrucción `with`, por ejemplo:

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp          154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp          154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp         4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp          18 Jul 10  2008 Fedora
>>>
```

Distinto en la versión 3.2: Se agregó compatibilidad con la instrucción `with`.

Distinto en la versión 3.3: Se agregó el parámetro *source_address*.

class `ftplib.FTP_TLS` (*host*="", *user*="", *passwd*="", *acct*="", *keyfile*=None, *certfile*=None, *context*=None, *timeout*=None, *source_address*=None)

Una subclase `FTP` que agrega compatibilidad con TLS a FTP como se describe en [RFC 4217](#). Conéctate como de costumbre al puerto 21 asegurando implícitamente la conexión de control antes de autenticar. Proteger la conexión de datos requiere que el usuario la solicite explícitamente llamando al método `prot_p()`. *context* es un objeto `ssl.SSLContext` que permite agrupar opciones de configuración SSL, certificados y claves privadas en una sola estructura (potencialmente de larga duración). Por favor, lee [Security considerations](#) para conocer las mejores prácticas.

keyfile y *certfile* son una alternativa de legado a *context* – pueden apuntar a una clave privada en formato PEM y certificar archivos de cadena (respectivamente) para la conexión SSL.

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Se agregó el parámetro *source_address*.

Distinto en la versión 3.4: La clase ahora admite el chequeo del nombre de *host* con `ssl.SSLContext.check_hostname` y *Server Name Indication* (véase `ssl.HAS_SNI`).

Obsoleto desde la versión 3.6: *keyfile* y *certfile* son rechazados a favor de *context*. Por favor, usa `ssl.SSLContext.load_cert_chain()` en su lugar, o deja que `ssl.create_default_context()` seleccione los certificados CA confiables para ti.

Aquí hay una sesión de ejemplo que usa la clase `FTP_TLS`:

```
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"'
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam', 'clockspeed', 'djbdns-
→jedi', 'docs', 'eaccelerator-jedi', 'favicon.ico', 'francotone', 'fugu',
→'ignore', 'libpuzzle', 'metalog', 'minidentd', 'misc', 'mysql-udf-global-
→user-variables', 'php-jenkins-hash', 'php-skein-hash', 'php-webdav',
→'phpaudit', 'phpbench', 'pincaster', 'ping', 'posto', 'pub', 'public',
→'public_keys', 'pure-ftp', 'qscan', 'qtc', 'sharedance', 'skycache', 'sound
→', 'tmp', 'ucarp']
```

exception `ftplib.error_reply`

Se lanza una excepción cuando una respuesta inesperada se recibe del servidor.

exception `ftplib.error_temp`

Se genera una excepción cuando se recibe un código de error que refiere a un error temporal (códigos de respuesta en el rango 400-499).

exception `ftplib.error_perm`

Se lanza una excepción cuando se recibe un código de error que refiere a un error permanente (códigos de respuesta en el rango 500-599).

exception `ftplib.error_proto`

Se lanza una excepción cuando se recibe una respuesta del servidor que no coincide con las especificaciones de respuesta del protocolo de transferencia de archivos (FTP), es decir, que comienza con un dígito en el rango 1-5.

`ftplib.all_errors`

El conjunto de todas las excepciones (como una tupla) que los métodos de instancias *FTP* pueden lanzar como resultado de problemas con la conexión FTP (a diferencia de los errores de programación hechos por el autor de la llamada). Este conjunto incluye las cuatro excepciones enumeradas anteriormente, como también *OSError* y *EOFError*.

Ver también:

Módulo *netrc* Analizador para el formato de archivo `.netrc`. El archivo `.netrc` suele ser utilizado por clientes FTP para cargar la información de autenticación de usuario antes de solicitarlo al usuario.

21.13.1 Objetos FTP

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.

Las instancias de *FTP* tienen los siguientes métodos:

`FTP.set_debuglevel` (*level*)

Establece el nivel de depuración de la instancia. Esto controla la cantidad de salida de depuración impresa. El valor predeterminado, 0, no produce una salida de depuración. Un valor de 1 produce una cantidad moderada de salida de depuración, generalmente una sola línea por solicitud. Un valor de 2 produce la cantidad máxima de salida de depuración, registrando cada línea enviada y recibida en la conexión de control.

`FTP.connect` (*host=""*, *port=0*, *timeout=None*, *source_address=None*)

Conéctate al puerto y al *host* dados. El número de puerto por defecto es 21, como se establece en la especificación de protocolo FTP. Raramente se necesita un número de puerto diferente. Esta función debería llamarse una vez por cada instancia; no debería llamarse si el *host* fue dado cuando se creó la instancia. Todos los otros métodos se pueden usar solo después de que se hizo una conexión. Si no se pasa ningún parámetro opcional *timeout* en segundos para el intento de conexión. Si no se pasa ningún *timeout*, se usará la configuración de tiempo de espera global. *source_address* es una tupla de 2 (*host*, *port*) para que el socket se enlace como su dirección de origen antes de conectarse.

Lanza un *evento auditor* `ftplib.connect` con los argumentos `self`, `host`, `port`.

Distinto en la versión 3.3: Se agregó el parámetro *source_address*.

`FTP.getwelcome` ()

Retornar el mensaje de bienvenida enviado por el servidor como respuesta a la conexión inicial. (Este mensaje a veces contiene renuncias de responsabilidad o información de ayuda que puede ser relevante para el usuario.)

`FTP.login` (*user='anonymous'*, *passwd=""*, *acct=""*)

Inicia sesión como el *usuario* dado. Los parámetros *passwd* y *acct* son opcionales y tienen como valor predeterminado la cadena vacía. Si no se especifica ningún *usuario*, toma como valor predeterminado 'anónimo', el valor predeterminado de *passwd* es 'anonymous@'. Esta función debería ser invocada solo una vez por cada instancia, luego de que se haya establecido una conexión; no debería invocarse en lo absoluto si se dio un anfitrión y un usuario cuando se creó la instancia. La mayoría de los comandos FTP solo están permitidos luego de que el cliente ha iniciado sesión. El parámetro *acct* proporciona «información contable»; pocos sistemas implementan esto.

FTP.`abort()`

Anula una transferencia de archivo que está en progreso. Usarlo no siempre funciona, pero vale la pena intentarlo.

FTP.`sendcmd(cmd)`

Envía una cadena de comando simple al servidor y retorna la cadena de caracteres de respuesta.

Genera un *evento auditor* `ftplib.sendcmd` con los argumentos `self`, `cmd`.

FTP.`voidcmd(cmd)`

Envía una cadena de caracteres como comando simple al servidor y maneja la respuesta. No retorna nada si recibe el código de respuesta que corresponde a una transferencia exitosa (códigos en el rango 200–299). Lanza *error_reply* de lo contrario.

Genera un *evento auditor* `ftplib.sendcmd` con los argumentos `self`, `cmd`.

FTP.`retrbinary(cmd, callback, blocksize=8192, rest=None)`

Recupera un archivo en el modo de transferencia binaria. *cmd* debería ser un comando `RETR` apropiado: `'RETR filename'`. La función *callback* es invocada por cada bloque de datos recibido, con un único argumento `bytes` que proporciona el bloque de datos. El argumento opcional *blocksize* especifica el tamaño máximo de fragmento que se leerá en el socket de bajo nivel creado para hacer la transferencia real (que también será el tamaño máximo de fragmento que se pasará a *callback*). Se elige un valor predeterminado razonable. *rest* significa lo mismo que en el método `transfercmd()`.

FTP.`retrlines(cmd, callback=None)`

Recupera una lista de archivos o directorios en modo de transferencia ASCII. *cmd* debería ser un comando `RETR` apropiado (véase `retrbinary()`) o un comando como `LIST` o `NLST` (usualmente solo la cadena de caracteres `'LIST'`). `LIST` recupera una lista de archivos y la información sobre esos archivos. `NLST` recupera una lista de nombres de archivos. La función *callback* se invoca por cada línea con un argumento de cadena de caracteres que contiene la línea con el CRLF final eliminado. El *callback* predeterminado imprime la línea a `sys.stdout`.

FTP.`set_pasv(val)`

Habilita el modo pasivo si *val* es verdadero, de lo contrario lo inhabilita. El modo pasivo es el valor predeterminado.

FTP.`storbinary(cmd, fp, blocksize=8192, callback=None, rest=None)`

Almacena un archivo en el modo de transferencia binaria. *cmd* debería ser un comando `STOR` apropiado: `'STOR filename'`. *fp* es un *file object* (abierto en modo binario) que es leído hasta EOF (final del archivo) usando el método `read()` en bloques de tamaño *blocksize* para proporcionar los datos que serán almacenados. El argumento *blocksize* toma 8192 como valor predeterminado. *callback* es un único parámetro invocable que se llama en cada bloque de datos luego de que fue enviado. *rest* significa lo mismo que en el método `transfercmd()`.

Distinto en la versión 3.2: Se agregó el parámetro *rest*.

FTP.`storlines(cmd, fp, callback=None)`

Almacena un archivo en el modo de transferencia ASCII. *cmd* debería ser un comando `STOR` apropiado (véase `storbinary()`). Las líneas son leídas hasta EOF (final del archivo) desde el *file object* *fp* (abierto en modo binario) usando su método `readline()` para proporcionar los datos que serán almacenados. *callback* es un único parámetro opcional invocable que se llama en cada línea luego de que es enviada.

FTP.`transfercmd(cmd, rest=None)`

Inicia una transferencia sobre la conexión de datos. Si la transferencia es activa, envía un comando `EPRT` o `PORT` y el comando de transferencia especificado por *cmd*, y acepta la conexión. Si el servidor es pasivo, envía un comando `EPSV` o `PASV`, lo conecta, e inicia el comando de transferencia. De cualquier manera, retorna el socket para la conexión.

Si el opcional *rest* está dado, un comando `REST` se envía al servidor, pasando *rest* como argumento. *rest* es usualmente un offset de byte en el archivo solicitado y le pide al servidor que reinicie enviando los bytes del archivo al offset solicitado, saltándose los bytes iniciales. Nota, sin embargo, que *rfc:959* requiere no solo que *rest* sea una cadena de caracteres que contiene caracteres en el rango imprimible del código ASCII 33 al 126. El método `transfercmd()`, entonces, convierte *rest* a una cadena de caracteres, pero no se ejecuta ningún chequeo en los contenidos de la cadena de caracteres. Si el servidor no reconoce el comando `REST`, se lanza

una excepción `error_reply`. Si esto ocurre, simplemente invoca `transfercmd()` sin un argumento `rest`.

FTP.**transfercmd** (*cmd*, *rest=None*)

Como `transfercmd()`, pero retorna una tupla de conexión de datos y el tamaño esperado de los datos. Si el tamaño esperado no se pudo computar, retornará `None` como tal. *cmd* y *rest* significan lo mismo que en `transfercmd()`.

FTP.**mlsd** (*path=""*, *facts=[]*)

List a directory in a standardized format by using MLSD command (RFC 3659). If *path* is omitted the current directory is assumed. *facts* is a list of strings representing the type of information desired (e.g. ["type", "size", "perm"]). Return a generator object yielding a tuple of two elements for every file found in path. First element is the file name, the second one is a dictionary containing facts about the file name. Content of this dictionary might be limited by the *facts* argument but server is not guaranteed to return all requested facts.

Nuevo en la versión 3.3.

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.

Nota: Si tu servidor admite el comando, `mlsd()` ofrece una API mejor.

FTP.**dir** (*argument[, ...]*)

Produce una lista de directorios como se retorna por el comando LIST, imprimiéndola en una salida estándar. El *argument* opcional es un directorio a ser listado (el valor predeterminado es el directorio del servidor actual). Se pueden utilizar argumentos múltiples para pasar las opciones que no son estándar al comando LIST. Si el último argumento es una función, se usa como función *callback* como en `retrlines()`; el valor predeterminado imprime a `sys.stdout`. Este método retorna `None`.

Nota: Si tu servidor admite el comando, `mlsd()` ofrece una API mejor.

FTP.**rename** (*fromname*, *toname*)

Asigna un nombre nuevo al archivo en el servidor desde *fromname* a *toname*.

FTP.**delete** (*filename*)

Remueve el archivo nombrado *filename* del servidor. De ser exitoso, retorna el texto de la respuesta, de lo contrario, lanza `error_perm` sobre errores de permiso o `error_reply` sobre otros errores.

FTP.**cwd** (*pathname*)

Configura el directorio actual en el servidor.

FTP.**mkd** (*pathname*)

Crea un nuevo directorio en el servidor.

FTP.**pwd** ()

Retorna el nombre de ruta del directorio actual en el servidor.

FTP.**rmd** (*dirname*)

Elimina el directorio en el servidor llamado *dirname*.

FTP.**size** (*filename*)

Solicita el tamaño del archivo llamado *filename* en el servidor. De ser exitoso, se retorna el tamaño del archivo como un entero, de lo contrario retorna `None`. Nótese que el comando SIZE no está estandarizado, pero es admitido por muchas implementaciones comunes de servidor.

FTP.**quit** ()

Envía un comando QUIT al servidor y cierra la conexión. Esta es la forma «políticamente correcta» de cerrar una conexión, pero puede lanzar una excepción si el servidor responde con un error al comando QUIT. Esto implica una llamada al método `close()` que hace inútil la instancia de `FTP` para llamadas posteriores (véase más adelante).

`FTP.close()`

Cierra la conexión de forma unilateral. Esto no debería aplicarse a una conexión ya cerrada, como luego de una llamada exitosa a `quit()`. Después de esta llamada, la instancia `FTP` ya no debería utilizarse (luego de una llamada a `close()` o `quit()` no puedes abrir nuevamente la conexión emitiendo otro método `login()`).

21.13.2 Objetos FTP_TLS

La clase `FTP_TLS` hereda de `FTP`, definiendo los siguientes objetos adicionales:

`FTP_TLS.ssl_version`

La versión SSL para usar (toma como predeterminado `ssl.PROTOCOL_SSLv23`).

`FTP_TLS.auth()`

Establece una conexión de control segura usando TLS o SSL, dependiendo de qué esté especificado en el atributo `ssl_version`.

Distinto en la versión 3.4: El método ahora admite el chequeo del nombre de *host* con `ssl.SSLContext.check_hostname` y *Server Name Indication* (véase `ssl.HAS_SNI`).

`FTP_TLS.ccc()`

Revierde el canal de control a texto plano. Esto puede ser útil para aprovechar cortafuegos que saben manejar NAT con FTP no-seguro sin abrir puertos fijos.

Nuevo en la versión 3.3.

`FTP_TLS.prot_p()`

Configura conexión de datos segura.

`FTP_TLS.prot_c()`

Configura la conexión de datos de tipo texto común.

21.14 poplib — Cliente de protocolo POP3

Código fuente: `Lib/poplib.py`

Este módulo define una clase, `POP3`, que encapsula una conexión a un servidor POP3 e implementa el protocolo como está definido en **RFC 1939**. La clase `POP3` soporta los mínimos y opcionales conjuntos de comandos de **RFC 1939**. La clase `POP3` también soporta el comando STLS introducido en **RFC 2595** para habilitar comunicación encriptada en una conexión ya establecida.

Adicionalmente, este módulo provee una clase `POP3_SSL`, que provee soporte para conectar servidores POP3 que usan SSL como una capa de protocolo subyacente.

Note que POP3, aunque ampliamente soportado, es obsoleto. La calidad de implementación de servidores POP3 varía ampliamente, y muchos son bastante pobres. Si su servidor de correo soporta IMAP, sería mejor utilizar la clase `imaplib.IMAP4`, ya que los servidores IMAP tienden a estar mejor implementados.

El módulo `poplib` provee dos clases:

class `poplib.POP3` (*host*, *port*=`POP3_PORT`[, *timeout*])

Esta clase implementa el protocolo POP3 actual. La conexión es creada cuando la instancia es inicializada. Si *port* se omite, el puerto POP3 estándar (110) es utilizado. El parámetro opcional *timeout* especifica un tiempo de espera en segundos para el intento de conexión (si no se especifica, la configuración global de tiempo de espera será utilizada).

Genera un *evento de auditoría* `poplib.connect` con argumentos `self`, `host`, `port`.

Genera un *evento de auditoría* `poplib.putline` con argumentos `self`, `line`.

class poplib.POP3_SSL(*host*, *port*=POP3_SSL_PORT, *keyfile*=None, *certfile*=None, *timeout*=None, *context*=None)

Esta es una subclase de `POP3` que conecta al servidor sobre un socket SSL encriptado. Si *port* no es especificado, 995, el puerto POP3-over-SSL es utilizado. *timeout* funciona como en el constructor de la clase `POP3`. *context* es un objeto `ssl.SSLContext` opcional que permite empaquetar opciones de configuración SSL, certificados y llaves privadas en una única (potencialmente longeva) estructura. Por favor lea *Security considerations* para buenas prácticas.

keyfile y *certfile* son una alternativa heredada a *context* - pueden apuntar a llaves privadas PEM - y archivos de cadena de certificados, respectivamente, para la conexión SSL.

Genera un *evento de auditoría* `poplib.connect` con argumentos `self`, `host`, `port`.

Genera un *evento de auditoría* `poplib.putline` con argumentos `self`, `line`.

Distinto en la versión 3.2: Parámetro *context* agregado.

Distinto en la versión 3.4: La clase ahora soporta verificación de nombre de host con `ssl.SSLContext.check_hostname` e *Indicación de Nombre de Servidor* (vea `ssl.HAS_SNI`).

Obsoleto desde la versión 3.6: *keyfile* y *certfile* están obsoletos en favor de *context*. Por favor utilice `ssl.SSLContext.load_cert_chain()` en su lugar, o permita que `ssl.create_default_context()` seleccione el sistema de certificados CA de confianza para usted.

Una excepción es definida como un atributo del módulo `poplib`:

exception poplib.error_proto

Excepción generada en cualquier error de este módulo (errores del módulo `socket` no son capturadas). La razón para la excepción es pasada al constructor como una cadena.

Ver también:

Módulo `imaplib` El módulo IMAP de Python.

Preguntas Frecuentes Sobre Fetchmail Las preguntas frecuentes para el cliente POP/IMAP `fetchmail` colecciona información en las variaciones del servidor POP3 e incumplimiento de RFC que puede ser útil si usted necesita escribir una aplicación basada en el protocolo POP.

21.14.1 Objetos POP3

Todos los comandos POP3 están representados por métodos del mismo nombre, en minúscula; la mayoría retornan el texto de respuesta enviado por el servidor.

Una instancia `POP3` tiene los siguientes métodos:

`POP3.set_debuglevel(level)`

Establece el nivel de depuración de la instancia. Esto controla la cantidad de salida de depuración impresa. Por defecto, 0, no produce salida de depuración. Un valor de 1 produce una moderada cantidad de salida de depuración, generalmente una única línea por solicitud. Un valor de 2 o mayor produce la máxima cantidad de salida de depuración, registrando cada línea enviada y recibida en la conexión de control.

`POP3.getwelcome()`

Retorna la cadena de saludo enviada por el servidor POP3.

`POP3.capa()`

Consulta las capacidades del servidor como está especificado en **RFC 2449**. Retorna un diccionario en la forma `{'nombre': ['param'...]}`.

Nuevo en la versión 3.4.

`POP3.user(username)`

Envía el comando del usuario, la respuesta debería indicar que una contraseña es requerida.

`POP3.pass_(password)`

Envía la contraseña, la respuesta incluye un conteo de mensaje y el tamaño del buzón de correo. Nota: el buzón de correo en el servidor está bloqueado hasta que `quit()` es llamado.

POP3.**apop** (*user*, *secret*)

Utiliza la autenticación APOP (más segura) para registrar en el servidor POP3.

POP3.**rpop** (*user*)

Utiliza autenticación RPOP (similar a los comandos r de UNIX) para registrar en el servidor POP3.

POP3.**stat** ()

Obtiene el estado del buzón de correo. El resultado es una tupla de 2 enteros: (conteo de mensaje, tamaño del buzón de correo).

POP3.**list** ([*which*])

Solicita lista de mensajes, el resultado es en la forma (respuesta, ['mesg_num octets', ...], octets). Si *which* está establecido, es el mensaje a listar.

POP3.**retr** (*which*)

Recupera el número de mensaje completo *which*, y establece marca de visto. El resultado es en la forma (respuesta, ['line', ...], octets).

POP3.**dele** (*which*)

Marca el número de mensaje *which* para eliminación. En la mayoría de los servidores las eliminaciones no están actualmente presentadas hasta QUIT (la mayor excepción es Eudora QPOP, que deliberadamente viola las RFC haciendo eliminaciones pendientes en cada desconexión).

POP3.**rset** ()

Remueve las marcas de eliminación para el buzón de correo.

POP3.**noop** ()

No hace nada. Puede ser utilizado como keep-alive.

POP3.**quit** ()

Cierra sesión: envía los cambios, desbloquea el buzón de correo, desconecta.

POP3.**top** (*which*, *howmuch*)

Recupera la cabecera del mensaje mas *howmuch* las líneas del mensaje después del cabecera del número de mensajes *which*. El resultado es en la forma (respuesta, ['línea', ...]. octets).

El comando TOP POP3 que este método utiliza, a diferencia del comando RETR, no establece la marca de visto del mensaje; desafortunadamente, TOP está pobremente especificado en las RFC y se rompe con frecuencia en servidores off-brand. Pruebe este método a mano contra los servidores POP3 que usted utilizará antes de confiar en él.

POP3.**uidl** (*which=None*)

Retorna la lista del resumen de mensajes (id único). Si *which* es especificado, el resultado contiene el id único para ese mensaje en la forma 'response msgnum uid', de otra forma el resultado es una lista (respuesta, ['msgnum uid', ...], octets).

POP3.**utf8** ()

Trata de cambiar al modo UTF-8. Retorna la respuesta del servidor si es exitosa, genera *error_proto* si no. Especificado en [RFC 6856](#).

Nuevo en la versión 3.5.

POP3.**stls** (*context=None*)

Comienza una sesión TLS en la conexión activa como está especificado en [RFC 2595](#). Esto es únicamente permitido antes de la autenticación de usuario

El parámetro *context* es un objeto *ssl.SSLContext* que permite empaquetar opciones de configuración SSL, certificados y llaves privadas en una única (potencialmente longeva) estructura. Por favor lea *Security considerations* para buenas prácticas.

Este método soporta verificación de nombre del host vía *ssl.SSLContext.check_hostname* e *Indicación de Nombre del Servidor* (vea *ssl.HAS_SNI*).

Nuevo en la versión 3.4.

Instancias de *POP3_SSL* no tienen métodos adicionales. La interfaz de esta subclase es idéntica a su padre.

21.14.2 Ejemplo POP3

Este es un ejemplo mínimo (sin chequeo de errores) que abre un buzón de correo y retorna e imprime todos los mensajes:

```
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)
```

Al final del módulo, hay una sección de test que contiene un ejemplo más extensivo de uso.

21.15 imaplib — Protocolo del cliente IMAP4

Código fuente: [Lib/imaplib.py](#)

Este módulo define tres clases *IMAP4*, *IMAP4_SSL* y *IMAP4_stream*, que encapsula una conexión a un servidor IMAP4 e implementa un gran subconjunto del protocolo de cliente IMAP4rev1 como se define en :rfc:2060. Es compatible con los servidores IMAP4 (**RFC 1730**), pero tenga en cuenta que el comando STATUS no es compatible con IMAP4.

El módulo *imaplib* proporciona tres clases, *IMAP4* es la clase base:

class `imaplib.IMAP4` (*host*="", *port*=*IMAP4_PORT*)

Esta clase implementa el protocolo actual IMAP4. La conexión se crea y la versión del protocolo (IMAP4 o IMAP4rev1) se determina cuando se inicializa la instancia. Si no se especifica *host*, se usa '' (el host local). Si se omite *port*, se usa el puerto IMAP4 estándar (143).

La clase *IMAP4* soporta la sentencia `with`. Cuando se usa de esta manera, el comando IMAP4 LOGOUT se emite automáticamente cuando se cierra la declaración `with`. P.ej.:

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

Distinto en la versión 3.5: Se agregó soporte para la sentencia `with`.

Se definen tres excepciones como atributos de la clase *IMAP4*:

exception `IMAP4.error`

Excepción lanzada por cualquier error. El motivo de la excepción se pasa al constructor como una cadena de caracteres.

exception `IMAP4.abort`

Los errores del servidor IMAP4 causan que esta excepción sea lanzada. Esta es una subclase de *IMAP4.error*. Tenga en cuenta que cerrar la instancia e instanciar una nueva generalmente permitirá la recuperación de esta excepción.

exception `IMAP4.readonly`

Esta excepción es lanzada cuando el servidor cambia el estado de un buzón de correo de escritura. Esta es una subclase de *IMAP4.error*. Algún otro cliente ahora tiene permiso de escritura y será necesario volver a abrir el buzón para volver a obtener el permiso de escritura.

También hay una subclase para conexiones seguras:


```
class imaplib.IMAP4_SSL (host="", port=IMAP4_SSL_PORT, keyfile=None, certfile=None,
                        ssl_context=None)
```

Esta es una subclase derivada de *IMAP4* que se conecta a través de un socket cifrado SSL (para usar esta clase necesita un módulo de socket que se compiló con soporte SSL). Si no se especifica *host*, se usa `' '` (el host local). Si se omite *port*, se usa el puerto IMAP4 estándar sobre SSL (993). *ssl_context* es un objeto *ssl.SSLContext* que permite agrupar opciones de configuración SSL, certificados y claves privadas en una estructura única (potencialmente de larga duración). Leer *Security considerations* para conocer las mejores prácticas.

keyfile y *certfile* son una alternativa heredada a *ssl_context* - pueden apuntar a claves privadas con formato PEM y archivos de cadena de certificados para la conexión SSL. Tenga en cuenta que los parámetros *keyfile/certfile* son mutuamente excluyentes con *ssl_context*, un *ValueError* se lanzará si *keyfile/certfile* se proporciona junto con *ssl_context*.

Distinto en la versión 3.3: *ssl_context* parámetro agregado.

Distinto en la versión 3.4: La clase ahora admite la verificación del nombre de host con *ssl.SSLContext.check_hostname* y *Server Name Indication* (ver *ssl.HAS_SNI*).

Obsoleto desde la versión 3.6: *keyfile* y *certfile* están obsoletos en favor de *ssl_context*. Utilice *ssl.SSLContext.load_cert_chain()* en su lugar, o deje que *ssl.create_default_context()* seleccione los certificados CA de confianza del sistema para usted.

La segunda subclase permite conexiones creadas por un proceso hijo:

```
class imaplib.IMAP4_stream (command)
```

Esta es una subclase derivada de *IMAP4* que se conecta a los descriptores de archivo `stdin/stdout` creados al pasar *command* a `subprocess.Popen()`.

Se definen las siguientes funciones de utilidad:

```
imaplib.Internaldate2tuple (datestr)
```

Analiza una cadena de caracteres IMAP4 INTERNALDATE y retorna la hora local correspondiente. El valor de retorno es una tupla *time.struct_time* o `None` si la cadena de caracteres tiene un formato incorrecto.

```
imaplib.Int2AP (num)
```

Converts an integer into a bytes representation using characters from the set [A .. P].

```
imaplib.ParseFlags (flagstr)
```

Convierte una respuesta IMAP4 FLAGS a una tupla de indicadores individuales.

```
imaplib.Time2Internaldate (date_time)
```

Convierte *date_time* en una representación IMAP4 INTERNALDATE. El valor de retorno es una cadena de caracteres en la forma: "DD-Mmm-YYYY HH:MM:SS +HHMM" (incluyendo comillas dobles). El argumento *date_time* puede ser un número (int o float) que representa segundos en un espacio de tiempo (como lo retorna *time.time()*), una tupla de 9 que representa la hora local como una instancia de *time.struct_time* (según lo retornado por *time.localtime()*), una instancia actualizada de *datetime.datetime*, o una cadena de caracteres entre comillas dobles. En el último caso, se supone que ya está en el formato correcto.

Tenga en cuenta que los números de mensaje IMAP4 cambian a medida que cambia el buzón de correo; en particular, después de que un comando EXPUNGE realiza eliminaciones, los mensajes restantes se vuelven a numerar. Por lo tanto, es muy recomendable usar UIDs en su lugar, con el comando UID.

Al final del módulo, hay una sección de prueba que contiene un ejemplo más extenso de uso.

Ver también:

Documents describing the protocol, sources for servers implementing it, by the University of Washington's IMAP Information Center can all be found at (**Source Code**) <https://github.com/uw-imap/imap> (**Not Maintained**).

21.15.1 Objetos de IMAP4

Todos los comandos IMAP4rev1 están representados por métodos del mismo nombre, mayúsculas o minúsculas.

Todos los argumentos de los comandos se convierten en cadenas de caracteres, excepto `AUTHENTICATE` y el último argumento de `APPEND` que se pasa como un literal IMAP4. Si es necesario (la cadena de caracteres contiene caracteres sensibles al protocolo IMAP4 y no está entre paréntesis ni comillas dobles), se cita cada cadena. Sin embargo, siempre se cita el argumento *password* para el comando `LOGIN`. Si desea evitar que se cite una cadena de argumento (por ejemplo: el argumento *flags* para `STORE`), encierre la cadena entre paréntesis (por ejemplo: `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 `bytes`, 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).

Las opciones *message_set* de los siguientes comandos son una cadena de caracteres que especifica uno o más mensajes sobre los que se debe actuar. Puede ser un número de mensaje simple (`'1'`), un rango de números de mensaje (`'2:4'`) o un grupo de rangos no contiguos separados por comas (`'1:3,6:9'`). Un rango puede contener un asterisco para indicar un límite superior infinito (`'3:*'`).

Una instancia de `IMAP4` tiene los siguientes métodos:

`IMAP4.append(mailbox, flags, date_time, message)`

Agregar *mensaje* al buzón de correo con nombre.

`IMAP4.authenticate(mechanism, authobject)`

Autenticar comando — requiere procesamiento de respuesta.

mechanism especifica qué mecanismo de autenticación se utilizará; debe aparecer en la variable de instancia *capabilities* en la forma `AUTH=mechanism`.

authobject debe ser un objeto invocable:

```
data = authobject(response)
```

Se llamará para procesar las respuestas de continuación del servidor; el argumento *response* que se pasa será `bytes`. Debería retornar `bytes` *data* que se codificarán en base64 y se enviarán al servidor. Debería retornar `None` si la respuesta de cancelación de cliente * se debe enviar en su lugar.

Distinto en la versión 3.5: los nombres de usuario y las contraseñas de cadena de caracteres ahora están codificados para `utf-8` en lugar de limitarse a ASCII.

`IMAP4.check()`

Control del buzón de correo en el servidor.

`IMAP4.close()`

Cerrar el buzón de correo seleccionado actualmente. Los mensajes eliminados se eliminan del buzón de correo de escritura. Este es el comando recomendado antes de `LOGOUT`.

`IMAP4.copy(message_set, new_mailbox)`

Copia mensajes *message_set* al final de *new_mailbox*.

`IMAP4.create(mailbox)`

Crea un nuevo buzón de correo llamado *mailbox*.

`IMAP4.delete(mailbox)`

Elimina el buzón de correo antiguo llamado *mailbox*.

`IMAP4.deleteacl(mailbox, who)`

Elimina las ACLs (elimina cualquier derecho) establecidas para quién en el buzón de correo.

`IMAP4.enable(capability)`

Habilita *capability* (ver [RFC 5161](#)). La mayoría de las capacidades no necesitan estar habilitadas. Actualmente solo esta soportada la capacidad `UTF8=ACCEPT` (consulte [RFC 6855](#)).

Nuevo en la versión 3.5: El método `enable()` en sí, y soporte [RFC 6855](#).

IMAP4.**expunge** ()

Elimina permanentemente los elementos eliminados del buzón de correo seleccionado. Genera una respuesta EXPUNGE para cada mensaje eliminado. Los datos retornados contienen una lista de números de mensaje EXPUNGE en el orden recibido.

IMAP4.**fetch** (*message_set*, *message_parts*)

Obtiene (partes de) mensajes. *message_parts* debe ser una cadena de nombres de partes de mensajes encerrados entre paréntesis, por ejemplo: " (UID BODY [TEXT]) ". Los datos retornados son una tupla de mensaje parte sobre y datos.

IMAP4.**getacl** (*mailbox*)

Obtiene la ACLs para *mailbox*. El método no es estándar, pero es compatible con el servidor Cyrus.

IMAP4.**getannotation** (*mailbox*, *entry*, *attribute*)

Recupera la ANNOTATIONS especificada para *mailbox*. El método no es estándar, pero es compatible con el servidor Cyrus.

IMAP4.**getquota** (*root*)

Obtiene el uso y los límites de los recursos de la cuota de *root*. Este método es parte de la extensión IMAP4 QUOTA definida en rfc2087.

IMAP4.**getquotaroot** (*mailbox*)

Obtiene la lista de quota roots para el nombrado *mailbox*. Este método es parte de la extensión IMAP4 QUOTA definida en rfc2087.

IMAP4.**list** ([*directory*], [*pattern*])

Lista los nombres de buzones de correo en *directory* coincidiendo *pattern*. *directory* por defecto es la carpeta de correo de nivel superior, y *pattern* por defecto coincide con cualquier cosa. Los datos retornados contienen una lista de respuestas LIST.

IMAP4.**login** (*user*, *password*)

Identifica al cliente con una contraseña de texto sin formato. El *password* será citado.

IMAP4.**login_cram_md5** (*user*, *password*)

Fuerza el uso de la autenticación CRAM-MD5 al identificar al cliente para proteger la contraseña. Solo funcionará si la respuesta CAPABILITY'' del servidor incluye la frase ``AUTH=CRAM-MD5.

IMAP4.**logout** ()

Cierra la conexión al servidor. Retorna la respuesta BYE desde el servidor .

Distinto en la versión 3.8: El método ya no ignora las excepciones silenciosamente arbitrarias.

IMAP4.**lsub** (*directory*="'", *pattern*='*')

Lista los nombres de buzones de correos suscritos en el patrón de coincidencia del directorio *directory* por defecto para el directorio de nivel superior y *pattern* por defecto para que coincida con cualquier buzón de correo. Los datos retornados son una tupla de mensaje parte sobre y datos.

IMAP4.**myrights** (*mailbox*)

Muestra mis ACLs para un buzón de correo (es decir, los derechos que tengo sobre el buzón de correo).

IMAP4.**namespace** ()

Retorna espacios de nombres IMAP como se define en [RFC 2342](#).

IMAP4.**noop** ()

Envía NOOP al servidor.

IMAP4.**open** (*host*, *port*)

Abre el socket al *port* en *host*. Este método lo llama implícitamente al constructor *IMAP4*. Los objetos de conexión establecidos por este método se utilizarán en los métodos *IMAP4.read()*, *IMAP4.readline()*, *IMAP4.send()* e *IMAP4.shutdown()*. Podemos sobrescribir este método.

Genera un *evento de auditoría* *imaplib.open* con argumentos *self*, *host*, *port*.

IMAP4.**partial** (*message_num*, *message_part*, *start*, *length*)

Obtiene partes truncadas de un mensaje. Los datos retornados son una tupla de mensaje parte sobre y datos.

IMAP4.proxyauth (*user*)

Asume la autenticación como *user*. Permite a un administrador autorizado hacer un proxy en el buzón de correo de cualquier usuario.

IMAP4.read (*size*)

Lee *size* bytes del servidor remoto. Podemos sobrescribir este método.

IMAP4.readline ()

Lee una línea del servidor remoto. Podemos sobrescribir este método.

IMAP4.recent ()

Solicita al servidor una actualización. Los datos retornados son `None` si no hay mensajes nuevos, de lo contrario el valor de respuesta es `RECENT`.

IMAP4.rename (*oldmailbox*, *newmailbox*)

Cambia el nombre del buzón de correo llamado *oldmailbox* a *newmailbox*.

IMAP4.response (*code*)

Retorna los datos para la respuesta *code* si se recibió, o `None`. Retorna el código dado, en lugar del tipo habitual.

IMAP4.search (*charset*, *criterion*[, ...])

Busca en el buzón de correo mensajes coincidentes. El *charset* puede ser `None`, en cuyo caso no se especificará `CHARSET` en la solicitud al servidor. El protocolo IMAP requiere que se especifique al menos un criterio; se lanzará una excepción cuando el servidor retorne un error. *charset* debe ser `None` si la capacidad `UTF8=ACCEPT` se habilitó utilizando el comando `enable()`.

Ejemplo:

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', 'LDJ')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.select (*mailbox*=`'INBOX'`, *readonly*=`False`)

Selecione un buzón de correo. Los datos retornados son el recuento de mensajes en *mailbox* (respuesta `EXISTS`). El *mailbox* predeterminado es `'INBOX'`. Si se establece el indicador *readonly*, no se permiten modificaciones en el buzón de correo.

IMAP4.send (*data*)

Envía *data* al servidor remoto. Podemos sobrescribir este método.

Lanza un *evento de auditoría* `imaplib.send` con argumentos `self`, *data*.

IMAP4.setacl (*mailbox*, *who*, *what*)

Establece una ACL para *mailbox*. El método no es estándar, pero es compatible con el servidor Cyrus.

IMAP4.setannotation (*mailbox*, *entry*, *attribute*[, ...])

Establece `ANNOTATIONS` para *mailbox*. El método no es estándar, pero es compatible con el servidor Cyrus.

IMAP4.setquota (*root*, *limits*)

Establece los recursos *limits* de la *quota* de los *root*. Este método es parte de la extensión IMAP4 QUOTA definida en `rfc2087`.

IMAP4.shutdown ()

Cierra la conexión establecida en `open`. Este método es llamado implícitamente por `IMAP4.logout()`. Podemos sobrescribir este método.

IMAP4.socket ()

Retorna la instancia de socket utilizada para conectarse al servidor.

IMAP4.sort (*sort_criteria*, *charset*, *search_criterion*[, ...])

El comando `sort` es una variante de `search` con semántica de clasificación para los resultados. Los datos retornados contienen una lista separada por espacios de números de mensajes coincidentes.

Sort tiene dos argumentos antes del argumento (s) *search_criterion*; una lista entre paréntesis de *sort_criteria*, y la búsqueda del *charset*. Tenga en cuenta que, a diferencia de `search`, el argumento de búsqueda *charset*

es obligatorio. También hay un comando `uid sort` que corresponde a `sort` de la misma manera que `uid search` corresponde a `search`. El comando `sort` primero busca en el buzón de correo mensajes que coincidan con los criterios de búsqueda dados utilizando el argumento *charset* para la interpretación de cadenas de caracteres en los criterios de búsqueda. Luego retorna los números de mensajes coincidentes.

Este es un comando de extensión IMAP4rev1.

IMAP4.**starttls** (*ssl_context=None*)

Envía un comando STARTTLS. El argumento *ssl_context* es opcional y debe ser un objeto *ssl.SSLContext*. Esto habilitará el cifrado en la conexión IMAP. Leer *Security considerations* para conocer las mejores prácticas.

Nuevo en la versión 3.2.

Distinto en la versión 3.4: El método ahora admite la verificación del nombre de host con *ssl.SSLContext.check_hostname* y *Server Name Indication* (ver *ssl.HAS_SNI*).

IMAP4.**status** (*mailbox, names*)

Solicita condiciones de estado con nombre para *mailbox*.

IMAP4.**store** (*message_set, command, flag_list*)

Altera las disposiciones de los indicadores para los mensajes en el buzón de correo. *command* esta especificado en la sección 6.4.6 de **RFC 2060** siendo como uno de «FLAGS», «+FLAGS» o «-FLAGS», opcionalmente con un sufijo «.SILENT».

Por ejemplo, para establecer el indicador de eliminación en todos los mensajes:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

Nota: Crear indicadores que contengan “]” (por ejemplo: «[test]») viola **RFC 3501** (el protocolo IMAP). Sin embargo, *imaplib* ha permitido históricamente la creación de tales etiquetas, y los servidores IMAP populares, como Gmail, aceptan y producen tales indicadores. Hay programas que no son de Python que también crean tales etiquetas. Aunque es una violación de RFC y se supone que los clientes y servidores IMAP son estrictos, *imaplib* continúa permitiendo que tales etiquetas se creen por razones de compatibilidad con versiones anteriores y, a partir de Python 3.6, las maneja si se envían desde el servidor, ya que esto mejora la compatibilidad en el mundo real.

IMAP4.**subscribe** (*mailbox*)

Suscribe al nuevo buzón de correo.

IMAP4.**thread** (*threading_algorithm, charset, search_criterion[, ...]*)

El comando `thread` es una variante de `search` con semántica de hilos para los resultados. Los datos retornados contienen una lista de miembros de hilos separados por espacios.

Los miembros de del hilo (*thread*) consisten en cero o más números de mensajes, delimitados por espacios, que indican sucesivos padres e hijos.

Thread tiene dos argumentos antes del argumento (s) *search_criterion*; un *threading_algorithm*, y la búsqueda del *charset*. Tenga en cuenta que, a diferencia de `search`, el argumento de búsqueda *charset* es obligatorio. También hay un comando `uid thread` que corresponde a `thread` de la misma manera que `uid search` corresponde a `search`. El comando `thread` primero busca en el buzón de correo mensajes que coincidan con los criterios de búsqueda dados utilizando el argumento *charset* para la interpretación de cadenas de caracteres en los criterios de búsqueda. Luego retorna los mensajes coincidentes enfilados según el algoritmo de subproceso especificado.

Este es un comando de extensión IMAP4rev1.

IMAP4.**uid** (*command, arg[, ...]*)

Ejecuta argumentos de comando con mensajes identificados por UID, en lugar de número de mensaje. Retorna

la respuesta apropiada al comando. Se debe proporcionar al menos un argumento; Si no se proporciona ninguno, el servidor retornará un error y se lanzará una excepción.

IMAP4.**unsubscribe** (*mailbox*)

Darse de baja del antiguo buzón de correo.

IMAP4.**xatom** (*name*[, ...])

Permite comandos de extensión simples notificados por el servidor en la respuesta CAPABILITY.

Los siguientes atributos se definen en instancias de *IMAP4*:

IMAP4.**PROTOCOL_VERSION**

El protocolo mas recientemente admitido en la respuesta CAPABILITY desde el servidor.

IMAP4.**debug**

Valor entero para controlar la salida de depuración. El valor de inicialización se toma de la variable del módulo Debug. Valores mayores de tres rastrean cada comando.

IMAP4.**utf8_enabled**

Valor booleano que normalmente es False, pero se establece en True si un comando *enable()* es exitosamente emitido para la capacidad UTF8=ACCEPT.

Nuevo en la versión 3.5.

21.15.2 Ejemplo IMAP4

Aquí hay un ejemplo mínimo (sin verificación de errores) que abre un buzón de correo y recupera e imprime todos los mensajes:

```
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()
```

21.16 nntplib — Protocolo de cliente NNTP

Código fuente: [Lib/nntplib.py](#)

This module defines the class *NNTP* which implements the client side of the Network News Transfer Protocol. It can be used to implement a news reader or poster, or automated news processors. It is compatible with **RFC 3977** as well as the older **RFC 977** and **RFC 2980**.

Aquí hay dos pequeños ejemplos de cómo se puede utilizar. Para enumerar algunas estadísticas sobre un grupo de noticias e imprimir los temas de los últimos 10 artículos:

```
>>> s = nntplib.NNTP('news.gmane.io')
>>> 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 1096 articles, range 1 to 1096
>>> resp, overviews = s.over((last - 9, last))
>>> for id, over in overviews:
...     print(id, nntplib.decode_header(over['subject']))
```

(continué en la próxima página)

(proviene de la página anterior)

```

...
1087 Re: Commit privileges for Łukasz Langa
1088 Re: 3.2 alpha 2 freeze
1089 Re: 3.2 alpha 2 freeze
1090 Re: Commit privileges for Łukasz Langa
1091 Re: Commit privileges for Łukasz Langa
1092 Updated ssh key
1093 Re: Updated ssh key
1094 Re: Updated ssh key
1095 Hello fellow committers!
1096 Re: Hello fellow committers!
>>> s.quit()
'205 Bye!'

```

Para publicar un artículo desde un archivo binario (esto supone que el artículo tiene encabezados válidos y que tienes permitido publicar en el grupo de noticias en particular):

```

>>> s = nntplib.NNTP('news.gmane.io')
>>> f = open('article.txt', 'rb')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 Bye!'

```

El módulo en sí define las siguientes clases:

class `nntplib.NNTP` (*host*, *port=119*, *user=None*, *password=None*, *readermode=None*, *usenetr= False* [, *timeout*])

Retorna un nuevo objeto *NNTP* que representa una conexión con el servidor NNTP ejecutándose en el *host*, escuchando en el puerto *port*. Se puede especificar un *timeout* opcional para la conexión de socket. Si se proporcionan las credenciales opcionales *user* y *password*, o si hay credenciales adecuadas en `.netrc` y el indicador opcional *usenetr* es verdadero, los comandos AUTHINFO USER y AUTHINFO PASS se utilizan para identificar y autenticar al usuario en el servidor. Si el indicador opcional *readermode* es verdadero, se envía un comando `mode reader` antes de que se realice la autenticación. El modo de lector a veces es necesario si se conecta a un servidor NNTP en el equipo local y tiene la intención de llamar a comandos específicos del lector, como `group`. Si obtienes un valor inesperado *NNTPPermanentError*, es posible que debas establecer *readermode*. La clase *NNTP* admite la instrucción `with` para consumir incondicionalmente las excepciones *OSError* y para cerrar la conexión NNTP cuando haya terminado, e.g.:

```

>>> from nntplib import NNTP
>>> with NNTP('news.gmane.io') as n:
...     n.group('gmane.comp.python.committers')
...
('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755, 'gmane.comp.
python.committers')
>>>

```

Genera un *evento de auditoría* `nntplib.connect` con los argumentos `self`, `host`, `port`.

Genera un *evento de auditoría* `nntplib.putline` con los argumentos `self`, `line`.

Distinto en la versión 3.2: *usenetr* es ahora `False` por defecto.

Distinto en la versión 3.3: El soporte para la declaración `with` fue añadido.

class `nntplib.NNTP_SSL` (*host*, *port=563*, *user=None*, *password=None*, *ssl_context=None*, *readermode=None*, *usenetr= False* [, *timeout*])

Retorna un nuevo objeto *NNTP_SSL*, que representa una conexión cifrada con el servidor NNTP ejecutándose en el *host*, escuchando en el puerto *port*. Los objetos *NNTP_SSL* tienen los mismos métodos que los objetos *NNTP*. Si se omite el *port* se utiliza el puerto 563 (NNTPS). *ssl_context* también es opcional, y también el objeto *SSLContext*. Por favor, lea *Security considerations* para conocer las buenas prácticas. Todos los demás parámetros se comportan igual que para *NNTP*.

Tenga en cuenta que SSL-on-563 no es recomendado por [RFC 4642](#), en favor de STARTTLS como se describe abajo. Sin embargo, algunos servidores solo admiten el primero.

Genera un *evento de auditoría* `nntplib.connect` con los argumentos `self`, `host`, `port`.

Genera un *evento de auditoría* `nntplib.putline` con los argumentos `self`, `line`.

Nuevo en la versión 3.2.

Distinto en la versión 3.4: La clase ahora admite la verificación del nombre de host con `ssl.SSLContext.check_hostname` e *Indicador del nombre del servidor* (SNI por sus siglas en inglés, consulte `ssl.HAS_SNI`).

exception nntplib.NNTPError

Derivado de la excepción estándar *Exception*, esta es la clase base para todas las excepciones generadas por el módulo `nntplib`. Las instancias de esta clase tienen el siguiente atributo:

response

La respuesta del servidor, si está disponible, como un objeto *str*.

exception nntplib.NNTPReplyError

Excepción generada cuando se recibe una respuesta inesperada del servidor.

exception nntplib.NNTPTemporaryError

Excepción generada cuando se recibe un código de respuesta dentro del rango del 400–499.

exception nntplib.NNTPPermanentError

Excepción generada cuando se recibe un código de respuesta dentro del rango del 500–599.

exception nntplib.NNTPProtocolError

Excepción generada cuando se recibe una respuesta del servidor que no comienza con un dígito dentro del rango 1–5.

exception nntplib.NNTPDataError

Excepción generada cuando hay algún error en los datos de la respuesta.

21.16.1 Objetos NNTP

Cuando están conectados, los objetos *NNTP* y *NNTP_SSL* admiten los siguientes métodos y atributos.

Atributos

NNTP.nttp_version

Un entero que representa la versión del protocolo NNTP compatible con el servidor. En la práctica, esto debería ser 2 para los servidores que anuncian el cumplimiento [RFC 3977](#) y 1 para otros.

Nuevo en la versión 3.2.

NNTP.nttp_implementation

Cadena que describe el nombre de software y la versión del servidor NNTP, o *None* si el servidor no lo anuncia.

Nuevo en la versión 3.2.

Métodos

La *response* que es retornada como el primer elemento de la tupla de retorno de casi todos los métodos es la respuesta del servidor: una cadena que comienza con un código de tres dígitos. Si la respuesta del servidor indica un error, el método genera una de las excepciones anteriores.

Muchos de los métodos siguientes toman un argumento opcional de solamente palabra clave *file*. Cuando se proporciona el argumento *file*, debe ser un *file object* abierto para la escritura binaria o el nombre de un archivo en disco a ser escrito. El método escribirá los datos retornados por el servidor (excepto la línea de respuesta y el punto de terminación) en el archivo; cualquier lista de líneas, tuplas u objetos que el método retorna normalmente estará vacía.

Distinto en la versión 3.2: Muchos de los siguientes métodos se han rediseñado y corregido, lo que los hace incompatibles con sus contrapartes 3.1.

NNTP.quit()

Envía un comando QUIT y cierra la conexión. Una vez que se ha invocado este método, no se debe invocar ningún otro método del objeto NNTP.

NNTP.getwelcome()

Retorna el mensaje de bienvenida enviado por el servidor en respuesta a la conexión inicial. (Este mensaje a veces contiene aclaraciones o información de ayuda que puede ser relevante para el usuario.)

NNTP.getcapabilities()

Retorna las capacidades [RFC 3977](#) anunciadas por el servidor, como una instancia *dict* mapeando nombres de capacidades a listas de valores (posiblemente vacías). En los servidores heredados que no entienden el comando CAPABILITIES, se retorna un diccionario vacío en su lugar.

```
>>> s = NNTP('news.gmane.io')
>>> 'POST' in s.getcapabilities()
True
```

Nuevo en la versión 3.2.

NNTP.login(user=None, password=None, usenetrc=True)

Envía comandos AUTHINFO con el nombre de usuario y la contraseña. Si *user* y *password* son *None* y *usenetrc* es verdadero, se utilizarán las credenciales de `~/ .netrc` si su uso es posible.

A menos que se retrase intencionalmente, el inicio de sesión se realiza normalmente durante la inicialización del objeto *NNTP* y no es necesario invocar esta función por separado. Para forzar el retraso de la autenticación, no debes establecer *user* o *password* al crear el objeto y debes establecer *usenetrc* en *False*.

Nuevo en la versión 3.2.

NNTP.starttls(context=None)

Envía un comando STARTTLS. Esto habilitará el cifrado en la conexión NNTP. El argumento *context* es opcional y debe ser el objeto *ssl.SSLContext*. Por favor lea [Security considerations](#) para conocer las buenas prácticas.

Tenga en cuenta que esto no se puede hacer después de que se haya transmitido la información de autenticación y la autenticación se produce de forma predeterminada, si es posible, durante la inicialización de un objeto *NNTP*. Consulte *NNTP.login()* para obtener información sobre cómo suprimir este comportamiento.

Nuevo en la versión 3.2.

Distinto en la versión 3.4: El método ahora admite la comprobación del nombre de host con *ssl.SSLContext.check_hostname* y *Indicador del nombre del servidor* (SNI por sus siglas en inglés, consulte *ssl.HAS_SNI*).

NNTP.newgroups(date, *, file=None)

Envía un comando NEWGROUPS. El argumento *date* debe ser un objeto *datetime.date* o *datetime.datetime*. Retorna un par (*response*, *groups*) donde *groups* es una lista que representa los grupos que son nuevos desde la fecha determinada. Sin embargo, si se proporciona *file*, entonces *groups* estará vacío.


```
>>> from datetime import date, timedelta
>>> resp, groups = s.newgroups(date.today() - timedelta(days=3))
>>> len(groups)
85
>>> groups[0]
GroupInfo(group='gmane.network.tor.devel', last='4', first='1', flag='m')
```

NNTP **.newnews** (*group*, *date*, *, *file=None*)

Envía un comando NEWNEWS. Aquí, *group* es un nombre de grupo o '*', y *date* tiene el mismo significado que para *newgroups()*. Retorna un par (*response*, *articles*) donde *articles* es una lista de identificadores de mensaje.

Este comando es inhabilitado frecuentemente por los administradores del servidor NNTP.

NNTP **.list** (*group_pattern=None*, *, *file=None*)

Envía un comando LIST o LIST ACTIVE. Retorna un par (*response*, *list*) donde *list* es una lista de tuplas que representan todos los grupos disponibles desde este servidor NNTP, opcionalmente coincidiendo con el patrón de cadena *group_pattern*. Cada tupla tiene el formato (*group*, *last*, *first*, *flag*), donde *group* es un nombre de grupo, *last* y *first* son los últimos y primeros números de artículo, y *flag* suele tomar uno de estos valores:

- y: Se permiten publicaciones locales y artículos de pares.
- m: El grupo está moderado y todas las publicaciones deben ser aprobadas.
- n: No se permiten publicaciones locales, solo artículos de pares.
- j: Los artículos de pares se archivan en el grupo de basura en su lugar.
- x: No hay publicaciones locales y los artículos de pares son ignorados.
- =foo.bar: Los artículos se archivan en el grupo foo.bar en su lugar.

Si *flag* tiene otro valor, el estado del grupo de noticias debe considerarse como desconocido.

Este comando puede devolver resultados muy grandes, especialmente si no se especifica *group_pattern*. Es mejor almacenar en caché los resultados sin conexión a menos que realmente necesite actualizarlos.

Distinto en la versión 3.2: *group_pattern* fue añadido.

NNTP **.descriptions** (*grouppattern*)

Envía un comando LIST NEWSGROUPS, donde *grouppattern* es una cadena comodín como se especifica en [RFC 3977](#) (es esencialmente lo mismo que las cadenas comodín de shell DOS o UNIX). Retorna un par (*response*, *descriptions*), donde *descriptions* es un diccionario que asigna nombres de grupos a descripciones textuales.

```
>>> resp, descs = s.descriptions('gmane.comp.python.*')
>>> len(descs)
295
>>> descs.popitem()
('gmane.comp.python.bio.general', 'BioPython discussion list (Moderated)')
```

NNTP **.description** (*group*)

Obtiene una descripción para un único grupo *group*. Si más de un grupo coincide (si “*group*” es una cadena comodín real), retorna la primera coincidencia. Si ningún grupo coincide, retorna una cadena vacía.

Esto elude el código de respuesta del servidor. Si necesita el código de respuesta, use *descriptions()*.

NNTP **.group** (*name*)

Envía un comando GROUP, donde *name* es el nombre del grupo. El grupo se selecciona como el grupo actual, si este existe. Retorna una tupla (*response*, *count*, *first*, *last*, *name*) donde *count* es el número (estimado) de artículos en el grupo, *first* es el primer número de artículo del grupo, *last* es el último número de artículo en el grupo y *name* es el nombre del grupo.

NNTP **.over** (*message_spec*, *, *file=None*)

Envía un comando OVER o un comando XOVER en servidores heredados. *message_spec* puede ser una cadena

que represente un identificador de mensaje o una tupla de números (*first*, *None*) que indique un rango de artículos en el grupo actual, o una tupla (*first*, *None*) que indique un rango de artículos comenzando desde *first* hasta el último artículo del grupo actual, o *None* para seleccionar el artículo actual en el grupo actual.

Retorna un par (*response*, *overviews*). *overviews* es una lista de tuplas del tipo (*article_number*, *overview*), una para cada artículo seleccionado por *message_spec*. Cada *overview* es un diccionario con el mismo número de elementos, pero este número depende del servidor. Estos elementos son encabezados de mensajes (la clave es entonces el nombre del encabezado en minúsculas) o elementos de metadatos (la clave es entonces el nombre de los metadatos precedido de " : "). Se garantiza la presencia de los siguientes elementos por la especificación NNTP:

- los encabezados *subject*, *from*, *date*, *message-id* y *references*
- los metadatos *:bytes*: el número de bytes en todo el artículo sin procesar (incluidos los encabezados y el cuerpo)
- los metadatos *:lines*: el número de líneas en el cuerpo del artículo

El valor de cada elemento es una cadena o *None* si no está presente.

Es aconsejable utilizar la función *decode_header()* en los valores del encabezado cuando pueden contener caracteres no-ASCII:

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, overviews = s.over((last, last))
>>> art_num, over = overviews[0]
>>> art_num
117216
>>> list(over.keys())
['xref', 'from', ':lines', ':bytes', 'references', 'date', 'message-id',
↳ 'subject']
>>> over['from']
'=?UTF-8?B?Ik1hcnRpbIB2LiBMw7Z3aXMi?= <martin@v.loewis.de>'
>>> nntplib.decode_header(over['from'])
'"Martin v. Löwis" <martin@v.loewis.de>'
```

Nuevo en la versión 3.2.

NNTP.**help** (*, *file=None*)

Envía un comando HELP. Retorna un par (*response*, *list*) donde *list* es una lista de cadenas de caracteres de ayuda.

NNTP.**stat** (*message_spec=None*)

Envía un comando STAT, donde *message_spec* es un identificador de mensaje (incluido en '<' y '>') o un número de artículo en el grupo actual. Si se omite *message_spec* o es *None* se considera el artículo actual del grupo actual. Retorna un triple (*response*, *number*, *id*) donde *number* es el número de artículo e *id* es el identificador del mensaje.

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, number, message_id = s.stat(first)
>>> number, message_id
(9099, '<20030112190404.GE29873@epoch.metaslash.com>')
```

NNTP.**next** ()

Envía un comando NEXT. Retorna como para *stat()*.

NNTP.**last** ()

Envía un comando LAST. Retorna como para *stat()*.

NNTP.**article** (*message_spec=None*, *, *file=None*)

Envía un comando ARTICLE, donde *message_spec* tiene el mismo significado que para *stat()*. Retorna una tupla (*response*, *info*) donde *info* es un *namedtuple* con tres atributos *number*, *message_id* y *lines* (en ese orden). *number* es el número de artículo del grupo (o 0 si la información no está disponible),

message_id el identificador del mensaje como una cadena y *lines* una lista de líneas (sin terminar líneas nuevas) que comprende el mensaje sin procesar, incluidos los encabezados y el cuerpo.

```
>>> resp, info = s.article('<20030112190404.GE29873@epoch.metaslash.com>')
>>> info.number
0
>>> info.message_id
'<20030112190404.GE29873@epoch.metaslash.com>'
>>> len(info.lines)
65
>>> info.lines[0]
b'Path: main.gmane.org!not-for-mail'
>>> info.lines[1]
b'From: Neal Norwitz <neal@metaslash.com>'
>>> info.lines[-3:]
[b'There is a patch for 2.3 as well as 2.2.', b'', b'Neal']
```

NNTP.**head** (*message_spec=None*, *, *file=None*)

Igual que *article()*, pero envía un comando HEAD. Las *lines* retornadas (o escritas a *file*) solo contendrán los encabezados del mensaje, no el cuerpo.

NNTP.**body** (*message_spec=None*, *, *file=None*)

Igual que *article()*, pero envía un comando BODY. Las *lines* retornadas (o escritas a *file*) solo contendrán los encabezados del mensaje, no el cuerpo.

NNTP.**post** (*data*)

Publica un artículo utilizando el comando POST. El argumento *data* es un *file object* abierto para la lectura binaria o cualquier iterable de objetos bytes (que representa las líneas sin procesar del artículo que se va a publicar). Debe representar un artículo de noticias bien formado, incluidos los encabezados requeridos. El método *post()* escapa automáticamente las líneas que comienzan con *.* y añade la línea de terminación.

Si el método tiene éxito, se retorna la respuesta del servidor. Si el servidor se niega a publicarlo, se genera un *NNTPReplyError*.

NNTP.**ihave** (*message_id*, *data*)

Envía un comando IHAVE. *message_id* es el identificador del mensaje que se enviará al servidor (incluido entre '<' y '>'). El parámetro *data* y el valor de retorno son los mismos que para *post()*.

NNTP.**date** ()

Devuelve un par (*response*, *date*). *date* es un objeto *datetime* que contiene la fecha y hora actuales del servidor.

NNTP.**slave** ()

Envía un comando SLAVE. Retorna la *response* del servidor.

NNTP.**set_debuglevel** (*level*)

Establece el nivel de depuración de la instancia. Esto controla la cantidad de salida de depuración impresa. El valor por defecto, 0, no produce salida de depuración. Un valor de 1 produce una cantidad moderada de salida de depuración, generalmente una sola línea por solicitud o por respuesta. Un valor de 2 o superior produce la cantidad máxima de salida de depuración, registrando cada línea enviada y recibida en la conexión (incluyendo el texto del mensaje).

Las siguientes son extensiones NNTP opcionales definidas en **RFC 2980**. Algunas de ellas han sido reemplazados por comandos más nuevos en **RFC 3977**.

NNTP.**xhdr** (*hdr*, *str*, *, *file=None*)

Envía un comando XHDR. El argumento *hdr* es una palabra clave de encabezado, por ejemplo 'subject'. El argumento *str* debe tener la forma 'first-last' donde *first* y *last* son el primer y último número de artículo para buscar. Retorna un par (*response*, *list*), donde *list* es una lista de pares (*id*, *text*), donde *id* es un número de artículo (como una cadena) y *text* es el texto del encabezado solicitado para ese artículo. Si se proporciona el parámetro *file*, entonces la salida del comando XHDR se almacena en un archivo. Si *file* es una cadena, entonces el método abrirá un archivo con ese nombre, que escribirá en él y luego lo cerrará. Si *file* es un *file object*, entonces comenzará invocando *write()* en él para almacenar las líneas de la salida del comando. Si se proporciona *file*, entonces retorna *list* o una lista vacía.

NNTP.**xover** (*start*, *end*, *, *file=None*)

Envía un comando XOVER. *start* and *end* son números de artículo que delimitan el rango de artículos a seleccionar. El valor de retorno es el mismo que para *over()*. Se recomienda usar *over()* en su lugar, ya que se usará automáticamente el comando más nuevo OVER si está disponible.

NNTP.**xpath** (*id*)

Retorna un par (*resp*, *path*), donde *path* es la ruta del directorio al artículo con un ID de mensaje *id*. La mayoría de las veces, los administradores del servidor NNTP no habilitan esta extensión.

Obsoleto desde la versión 3.3: La extensión XPATH no se utiliza activamente.

21.16.2 Funciones de utilidad

El módulo también define la siguiente función de utilidad:

nntplib.**decode_header** (*header_str*)

Decodifica un valor de encabezado, eliminando los caracteres de escape que no sean ASCII. *header_str* debe ser un objeto *str*. Se retorna el valor sin escape. Se recomienda utilizar esta función para mostrar algunos encabezados en una forma legible por humanos:

```
>>> decode_header("Some subject")
'Some subject'
>>> decode_header("=?ISO-8859-15?Q?D=E9buter_en_Python?=")
'Débuter en Python'
>>> decode_header("Re: =?UTF-8?B?cHJvYmzDqG1lIGRlIG1hdHJpY2U=?=")
'Re: problème de matrice'
```

21.17 smtplib — Cliente de protocolo SMTP

Código fuente: [Lib/smtplib.py](#)

El módulo *smtplib* define un objeto de sesión de cliente SMTP que se puede usar para mandar correo a cualquier máquina de Internet con un demonio de escucha SMTP o ESMTP. Para detalles sobre el funcionamiento de SMTP o ESMTP, consulta **RFC 821** (Simple Mail Transfer Protocol) y **RFC 1869** (Extensiones de Servicio SMTP).

class smtplib.**SMTP** (*host=""*, *port=0*, *local_hostname=None*[, *timeout*], *source_address=None*)

Una instancia *SMTP* encapsula una conexión SMTP. Tiene métodos que admiten un repertorio completo de operaciones SMTP y ESMTP. Si se proporcionan los parámetros opcionales de *host* y *puerto*, el método *connect()* de SMTP se llama con esos parámetros durante la inicialización. Si se especifica, *local_hostname* se usa como FQDN del host local in el comando HELO/EHLO. De lo contrario, el *hostname* local se busca usando *socket.getfqdn()*. Si la llamada a *connect()* retorna cualquier cosa que no sea un código de éxito, se lanza un *SMTPConnectError*. El parámetro *timeout* opcional especifica un timeout en segundos para bloquear operaciones como el intento de conexión (si no se especifica, se utilizará la configuración global del timeout por defecto). Si el timeout de espera expira, se lanza *socket.timeout*. El parámetro opcional *source_address* permite el enlace a alguna dirección de origen específica en una máquina con múltiples interfaces de red, y/o a algún puerto TCP de origen específico. Se necesita una tupla de 2 (*host*, *puerto*), para que el socket se enlace como su dirección de origen antes de conectarse. Si se omite (o si el *host* o el *puerto* son '' y/o 0 respectivamente) se utilizara el comportamiento por defecto del SO.

Para un uso normal, solo debe requerir los métodos *initialization/connect*, *sendmail()* y *SMTP.quit()*. A continuación se incluye un ejemplo.

La clase *SMTP* admite la instrucción *with*. Cuando se usa así, el comando SMTP QUIT se emite automáticamente cuando la *with* sale de la instrucción. por ejemplo:

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

Genera un *auditing event* `smtplib.send` con argumentos `self`, `data`.

Distinto en la versión 3.3: Se agregó soporte para la sentencia `with`.

Distinto en la versión 3.3: se agregó el argumento `source_address`.

Nuevo en la versión 3.5: La extensión SMTPUTF8 ([RFC 6531](#)) ahora es compatible.

class `smtplib.SMTP_SSL` (`host=""`, `port=0`, `local_hostname=None`, `keyfile=None`, `certfile=None`, `timeout`, `context=None`, `source_address=None`)

Una instancia de `SMTP_SSL` se comporta exactamente igual que las instancias de `SMTP`. `SMTP_SSL` debe usarse para situaciones donde se requiere SSL desde el comienzo de la conexión y el uso `starttls()` no es apropiado. Si no se especifica `host`, se utiliza el host local. Si `port` es cero, se utiliza el puerto estándar SMTP sobre SSL (465). Los argumentos opcionales `local_hostname`, `timeout` y `source_address` tienen el mismo significado que en la clase `SMTP`. `context`, también opcional, puede contener una `SSLContext` y permite configurar varios aspectos de la conexión segura. Por favor lea *Security considerations* para conocer las mejores prácticas.

`keyfile` y `certfile` son una alternativa heredada a `context` y pueden apuntar a una clave privada con formato PEM y un archivo de cadena de certificados para la conexión SSL.

Distinto en la versión 3.3: se agregó `contexto`.

Distinto en la versión 3.3: se agregó el argumento `source_address`.

Distinto en la versión 3.4: La clase ahora admite la verificación del nombre de host con `ssl.SSLContext.check_hostname` y *Server Name Indication* (ver `ssl.HAS_SNI`).

Obsoleto desde la versión 3.6: `keyfile` y `certfile` están obsoletos en favor de `context`. Por favor use `ssl.SSLContext.load_cert_chain()` en su lugar, o deje que `ssl.create_default_context()` seleccione los certificados CA confiables del sistema para usted.

class `smtplib.LMTP` (`host=""`, `port=LMTP_PORT`, `local_hostname=None`, `source_address=None`)

El protocolo LMTP, que es muy similar a ESMTP, se basa en gran medida en el cliente SMTP estándar. Es común usar sockets Unix para LMTP, por lo que nuestro método `connect()` debe ser compatible con eso, así como con un servidor `host:puerto` normal. Los argumentos opcionales `local_hostname` y `source_address` tienen el mismo significado que en la clase `SMTP`. Para especificar un socket Unix, debe usar una ruta absoluta para `host`, comenzando con `"/`.

Se admite la autenticación mediante el mecanismo SMTP habitual. Cuando se usa un socket Unix, LMTP generalmente no admite ni requiere autenticación, pero su millaje puede variar.

También se define una buena selección de excepciones:

exception `smtplib.SMTPException`

Subclase de `OSError` que es la clase de excepción base para todas las demás excepciones proporcionadas por este módulo.

Distinto en la versión 3.4: `SMTPException` se convirtió en subclase de `OSError`

exception `smtplib.SMTPServerDisconnected`

Esta excepción se genera cuando el servidor se desconecta inesperadamente o cuando se intenta usar la instancia `SMTP` antes de conectarlo a un servidor.

exception `smtplib.SMTPResponseException`

Clase base para todas las excepciones que incluyen un código de error SMTP. Estas excepciones se generan en algunos casos cuando el servidor SMTP devuelve un código de error. El código de error se almacena en el atributo `smtp_code` del error, y el atributo `smtp_error` se establece en el mensaje de error.

exception `smtplib.SMTPSenderRefused`

Dirección del remitente rechazada. Además de los atributos establecidos por todas las excepciones `SMTPResponseException`, éste establece “remitente” para la cadena de caracteres que el servidor SMTP rechazó.

exception `smtplib.SMTPRecipientsRefused`

Se rechazaron todas las direcciones de destinatarios. Los errores para cada destinatario son accesibles mediante el atributo `recipients`, el cual es un diccionario del mismo tipo que el `SMTP.sendmail()` retorna.

exception `smtplib.SMTPDataError`

El servidor SMTP se negó a aceptar los datos del mensaje.

exception `smtplib.SMTPConnectError`

Se produjo un error durante el establecimiento de conexión con el servidor.

exception `smtplib.SMTPHeloError`

El servidor rechazó nuestro mensaje HELO.

exception `smtplib.SMTPNotSupportedError`

El servidor no admite el comando o la opción que se intentó.

Nuevo en la versión 3.5.

exception `smtplib.SMTPAuthenticationError`

La autenticación SMTP salió mal. Lo más probable es que el servidor no aceptó la combinación proporcionada de `username/password`.

Ver también:

RFC 821 - Simple Mail Transfer Protocol Definición de protocolo para SMTP. Este documento cubre el modelo, el procedimiento operativo y los detalles del protocolo para SMTP.

RFC 1869 - Extensiones de Servicio SMTP Definición de las extensiones ESMTP para SMTP. Esto describe un marco para extender SMTP con nuevos comandos, que admite el descubrimiento dinámico de los comandos proporcionados por el servidor y define algunos comandos adicionales.

21.17.1 Objetos SMTP

Una instancia `SMTP` tiene los siguientes métodos:

SMTP.set_debuglevel (*level*)

Establezca el nivel de salida de depuración. Un valor de 1 o `True` para *level* da como resultado mensajes de depuración para la conexión y para todos los mensajes enviados y recibidos desde el servidor. Un valor de 2 para *level* da como resultado que estos mensajes tengan una marca de tiempo.

Distinto en la versión 3.5: Se agregó el nivel de depuración 2.

SMTP.docmd (*cmd*, *args=""*)

Envíe un comando *cmd* al servidor. El argumento opcional *args* simplemente se concatena al comando, separado por un espacio.

Esto devuelve una tupla de 2 compuestos por un código de respuesta numérico y la línea de respuesta real (las respuestas de varias líneas se unen en una línea larga).

En funcionamiento normal, no debería ser necesario llamar a este método explícitamente. Se utiliza para implementar otros métodos y puede resultar útil para probar extensiones privadas.

Si se pierde la conexión con el servidor mientras se espera la respuesta, se activará `SMTPServerDisconnected`.

SMTP.connect (*host='localhost'*, *port=0*)

Conéctese a un host en un puerto determinado. Los valores predeterminados son para conectarse al host local en el puerto SMTP estándar (25). Si el nombre de host termina con dos puntos (':') seguido de un número, ese sufijo se eliminará y el número se interpretará como el número de puerto a utilizar. El constructor invoca automáticamente este método si se especifica un host durante la instanciación. Devuelve una tupla de 2 del código de respuesta y el mensaje enviado por el servidor en su respuesta de conexión.

Genera un *evento de auditoría* `smtpplib.connect` con argumentos `self`, `host`, `port`.

SMTP.**hello** (*name*="")

Identifíquese en el servidor SMTP usando HELO. El argumento del nombre de host tiene como valor predeterminado el nombre de dominio completo del host local. El mensaje devuelto por el servidor se almacena como el atributo `hello_resp` del objeto.

En funcionamiento normal, no debería ser necesario llamar a este método explícitamente. Será llamado implícitamente por `sendmail()` cuando sea necesario.

SMTP.**ehlo** (*name*="")

Identifíquese en un servidor ESMTP usando EHLO. El argumento del nombre de host tiene como valor predeterminado el nombre de dominio completo del host local. Examine la respuesta para la opción ESMTP y guárdelos para que los use `has_extn()`. También establece varios atributos informativos: el mensaje devuelto por el servidor se almacena como el atributo `ehlo_resp`, `does_esmtp` se establece en verdadero o falso dependiendo de si el servidor admite ESMTP, y `esmtp_features` será un diccionario que contiene los nombres de las extensiones de servicio SMTP que admite este servidor, y sus parámetros (si los hay).

A menos que desee utilizar `has_extn()` antes de enviar correo, no debería ser necesario llamar a este método explícitamente. Se llamará implícitamente por `sendmail()` cuando sea necesario.

SMTP.**ehlo_or_helo_if_needed**()

Este método llama a `ehlo()` o `helo()` si no ha habido ningún comando EHLO o HELO anterior en esta sesión. Primero prueba ESMTP EHLO.

SMTPHeloError El servidor no respondió correctamente al saludo HELO.

SMTP.**has_extn** (*name*)

Retorna `True` si *name* está en el conjunto de extensiones de servicio SMTP devueltas por el servidor, `False` en caso contrario. El método es insensible a la presencia de mayúsculas en *name*.

SMTP.**verify** (*address*)

Verifique la validez de una dirección en este servidor usando SMTP VRFY. Retorna una tupla que consta del código 250 y una dirección completa **RFC 822** (incluido el nombre humano) si la dirección del usuario es válida. De lo contrario, devuelve un código de error SMTP de 400 o más y una cadena de error.

Nota: Muchos sitios desactivan SMTP VRFY para frustrar a los spammers.

SMTP.**login** (*user*, *password*, *, *initial_response_ok*=True)

Inicie sesión en un servidor SMTP que requiera autenticación. Los argumentos son el nombre de usuario y la contraseña para autenticarse. Si no ha habido ningún comando EHLO o HELO anterior en esta sesión, este método prueba primero ESMTP EHLO. Este método regresará normalmente si la autenticación fue exitosa o puede generar las siguientes excepciones:

SMTPHeloError El servidor no respondió correctamente al saludo HELO.

SMTPAuthenticationError El servidor no aceptó la combinación de nombre de username/password.

SMTPNotSupportedError El servidor no admite el comando AUTH.

SMTPException No se encontró ningún método de autenticación adecuado.

Cada uno de los métodos de autenticación admitidos por `smtpplib` se prueban a su vez si se anuncian como admitidos por el servidor. Consulte `auth()` para obtener una lista de los métodos de autenticación admitidos. *initial_response_ok* se pasa a `auth()`.

El argumento de palabra clave opcional *initial_response_ok* especifica si, para los métodos de autenticación que lo admiten, se puede enviar una «respuesta inicial» como se especifica en **RFC 4954** junto con el comando AUTH, en lugar de requerir un desafío/respuesta.

Distinto en la versión 3.5: **SMTPNotSupportedError** se puede generar y se agregó el parámetro *initial_response_ok*.

SMTP.**auth** (*mechanism*, *authobject*, *, *initial_response_ok*=True)

Emita un comando SMTP AUTH para el *mechanism* de autenticación especificado y maneje la respuesta de desafío a través de *authobject*.

mechanism especifica qué mecanismo de autenticación se utilizará como argumento para el comando AUTH; los valores válidos son los enumerados en el elemento *auth* de *esmtplib.features*.

authobject debe ser un objeto invocable que tome un único argumento opcional:

```
data = authobject(challenge=None)
```

Si la verificación de respuesta inicial devuelve *None*, o si *initial_response_ok* es falso, se llamará a *authobject()* para procesar la respuesta de desafío del servidor; el argumento *challenge* que se pasa será un *bytes*. Debería devolver *data* ASCII *str* que serán codificados en base64 y enviados al servidor.

Si la verificación de respuesta inicial devuelve *None*, o si *initial_response_ok* es falso, se llamará a *authobject()* para procesar la respuesta de desafío del servidor; el argumento *challenge* que se pasa será un *bytes*. Debería devolver *data* ASCII *str* que serán codificados en base64 y enviados al servidor.

La clase SMTP proporciona *authobjects* para los mecanismos CRAM-MD5, PLAIN y LOGIN; se denominan SMTP.*auth_cram_md5*, SMTP.*auth_plain* y SMTP.*auth_login* respectivamente. Todos requieren que las propiedades de *user* y *password* de la instancia SMTP se establezcan en los valores adecuados.

El código de usuario normalmente no necesita llamar a *auth* directamente, sino que puede llamar al método *login()*, que probará cada uno de los mecanismos anteriores a su vez, en el orden indicado. *auth* está expuesto para facilitar la implementación de métodos de autenticación que no (o aún no) son compatibles directamente con *smtplib*.

Nuevo en la versión 3.5.

SMTP.**starttls** (*keyfile*=None, *certfile*=None, *context*=None)

Ponga la conexión SMTP en modo TLS (Seguridad de la capa de transporte). Todos los comandos SMTP que siguen se cifrarán. Entonces debería llamar a *ehlo()* de nuevo.

Si se proporcionan *keyfile* y *certfile*, se utilizan para crear una *ssl.SSLContext*.

El parámetro *context* opcional es un objeto *ssl.SSLContext*; Esta es una alternativa al uso de un archivo de claves y un archivo de certificado y, si se especifica, tanto *keyfile* como *certfile* deben ser *None*.

Si no ha habido ningún comando EHLO o HELO anterior en esta sesión, este método intenta ESMTP EHLO primero.

Obsoleto desde la versión 3.6: *keyfile* y *certfile* están obsoletos en favor de *context*. Por favor use *ssl.SSLContext.load_cert_chain()* en su lugar, o deje que *ssl.create_default_context()* seleccione los certificados CA confiables del sistema para usted.

SMTPHeloError El servidor no respondió correctamente al saludo HELO.

SMTPNotSupportedError El servidor no admite la extensión STARTTLS.

RuntimeError La compatibilidad con SSL/TLS no está disponible para su intérprete de Python.

Distinto en la versión 3.3: se agregó *contexto*.

Distinto en la versión 3.4: El método ahora admite la verificación del nombre de host con *SSLContext.check_hostname* y *Server Name Indicator* (ver *HAS_SNI*).

Distinto en la versión 3.5: El error generado por falta de compatibilidad con STARTTLS ahora es la subclase *SMTPNotSupportedError* en lugar de la base *SMTPException*.

SMTP.**sendmail** (*from_addr*, *to_addrs*, *msg*, *mail_options*=(), *rcpt_options*=())

Enviar correo. Los argumentos requeridos son **RFC 822** cadena de dirección de origen, una lista de **RFC 822** cadenas de dirección (una cadena simple se tratará como una lista con 1 dirección) y una cadena de mensaje. La persona que llama puede pasar una lista de opciones de ESMTP (como *8bitmime*) para usar en los comandos MAIL FROM como *mail_options*. Las opciones de ESMTP (como los comandos DSN) que deben usarse con todos los comandos RCPT se pueden pasar como *rcpt_options*. (Si necesita usar diferentes opciones de ESMTP

para diferentes destinatarios, debe usar los métodos de bajo nivel como `mail()`, `rcpt()` y `data()` para enviar el mensaje).

Nota: Los parámetros `from_addr` y `to_addrs` se utilizan para construir el sobre del mensaje utilizado por los agentes de transporte. `sendmail` no modifica los encabezados de los mensajes de ninguna manera.

`msg` puede ser una cadena que contenga caracteres en el rango ASCII o una cadena de bytes. Una cadena se codifica en bytes utilizando el códec `ascii`, y los caracteres `\r` y `\n` solitarios se convierten en caracteres `\r\n`. Una cadena de bytes no se modifica.

Si no ha habido ningún comando `EHLO` o `HELO` anterior en esta sesión, este método prueba primero ESMTP `EHLO`. Si el servidor utiliza ESMTP, se le pasará el tamaño del mensaje y cada una de las opciones especificadas (si la opción está en el conjunto de funciones que anuncia el servidor). Si `EHLO` falla, se probará `HELO` y se eliminarán las opciones de ESMTP.

Este método volverá normalmente si se acepta el correo para al menos un destinatario. De lo contrario, generará una excepción. Es decir, si este método no genera una excepción, alguien debería recibir su correo. Si este método no genera una excepción, devuelve un diccionario, con una entrada para cada destinatario rechazado. Cada entrada contiene una tupla del código de error SMTP y el mensaje de error adjunto enviado por el servidor.

Si se incluye `SMTPUTF8` en `mail_options` * y el servidor lo admite, *`from_addr` y `to_addrs` pueden contener caracteres no ASCII.

Este método puede lanzar las siguientes excepciones:

`SMTPRecipientsRefused` Todos los destinatarios fueron rechazados. Nadie recibió el correo. El atributo `recipients` del objeto de excepción es un diccionario con información sobre los destinatarios rechazados (como el que se retorna cuando se aceptó al menos un destinatario).

`SMTPHeloError` El servidor no respondió correctamente al saludo `HELO`.

`SMTPSenderRefused` El servidor no aceptó el `from_addr`.

`SMTPDataError` El servidor respondió con un código de error inesperado (que no sea el rechazo de un destinatario).

`SMTPNotSupportedError` Se proporcionó `SMTPUTF8` en `mail_options` pero el servidor no lo admite.

A menos que se indique lo contrario, la conexión estará abierta incluso después de que se lance una excepción.

Distinto en la versión 3.2: `msg` puede ser una cadena de bytes.

Distinto en la versión 3.5: Se agregó compatibilidad con `SMTPUTF8`, y **`SMTPNotSupportedError`** puede aparecer si se especifica `SMTPUTF8` pero el servidor no lo admite.

SMTP **`.send_message`** (`msg`, `from_addr=None`, `to_addrs=None`, `mail_options=()`, `rcpt_options=()`)

Este es un método conveniente para llamar a `sendmail()` con el mensaje representado por un objeto `email.message.Message`. Los argumentos tienen el mismo significado que para `sendmail()`, excepto que `msg` es un objeto `Mensaje`.

Si `from_addr` es `None` o `to_addrs` es `None`, `send_message` llena esos argumentos con direcciones extraídas de los encabezados de `msg` como se especifica en **RFC 5322**: `from_addr` se establece en el campo `Sender` si está presente, y de lo contrario, en el campo `From`. `to_addrs` combina los valores (si los hay) de los campos `To`, `Cc` y `Bcc` de `msg`. Si aparece exactamente un conjunto de encabezados `Resent-*` en el mensaje, los encabezados normales se ignoran y en su lugar se utilizan los encabezados `Resent-*`. Si el mensaje contiene más de un conjunto de encabezados `Resent-*`, se lanza un `ValueError`, ya que no hay forma de detectar sin ambigüedades el conjunto más reciente de encabezados `Resent-`.

`send_message` serializa `msg` usando `BytesGenerator` con `"r n"` como `linesep`, y llama a `sendmail()` para transmitir el mensaje resultante. Independientemente de los valores de `from_addr` y `to_addrs`, `send_message` no transmite ningún encabezado `Bcc` o `Resent-Bcc` que puedan aparecer en `msg`. Si alguna de las direcciones en `from_addr` y `to_addrs` contiene caracteres que no son ASCII y el servidor no anuncia la compatibilidad con `SMTPUTF8`, se lanza un error `SMTPNotSupported`. De lo contrario, el

Message se serializa con un clon de su *policy* con el atributo *utf8* establecido en True y SMTPUTF8 y BODY=8BITMIME se agregan a *mail_options*.

Nuevo en la versión 3.2.

Nuevo en la versión 3.5: Soporte para direcciones internacionalizadas (SMTPUTF8).

SMTP.**quit** ()

Termine la sesión SMTP y cierre la conexión. Retorna el resultado del comando SMTP QUIT.

Los métodos de bajo nivel correspondientes a los comandos estándar SMTP/ESMTP `HELP, RSET, NOOP, MAIL, RCPT, y DATA también están soportados. Normalmente, no es necesario llamarlos directamente, por lo que no se documentan aquí. Para más detalles, consulte el código del módulo.

21.17.2 Ejemplo SMTP

Este ejemplo solicita al usuario las direcciones necesarias en el sobre del mensaje (direcciones “To” y “From”) y el mensaje que se entregará. Tenga en cuenta que los encabezados que se incluirán con el mensaje deben incluirse en el mensaje tal y como se introdujeron; este ejemplo no procesa los encabezados **RFC 822**. En particular, las direcciones “To” y “From” deben incluirse explícitamente en los encabezados de los mensajes.

```
import smtplib

def prompt(prompt):
    return 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 True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print("Message length is", len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Nota: En general, querrá usar las características del paquete *email* para construir un mensaje de correo electrónico, que luego puede enviar a través de *send_message()*; ver *email: Ejemplos*.

21.18 smtpd — Servidor SMTP

Source code: [Lib/smtpd.py](#)

Este módulo ofrece varias clases para implementar servidores SMTP (correo electrónico).

Ver también:

El paquete [aiosmtpd](#) es un reemplazo recomendado para este módulo. Se basa en [asyncio](#) y proporciona una API más sencilla. [smtpd](#) debería considerarse obsoleto.

Este módulo ofrece varias implementaciones del servidor; una es una implementación genérica de no hace nada, pero cuyos métodos pueden ser sobrescritos para crear una implementación concreta, mientras que las otras dos ofrecen estrategias específicas de envío de correo.

Además, SMTPChannel puede ampliarse para implementar un comportamiento de interacción muy específico con clientes SMTP.

El código admite [RFC 5321](#), más las extensiones [RFC 1870](#) SIZE y [RFC 6531](#) SMTPUTF8.

21.18.1 Objetos SMTPServer

class `smtpd.SMTPServer` (*localaddr*, *remoteaddr*, *data_size_limit=33554432*, *map=None*,
enable_SMTPUTF8=False, *decode_data=False*)

Crea un nuevo objeto *SMTPServer*, que se vincula a la dirección local *localaddr*. Tratará *remoteaddr* como un transmisor SMTP ascendente. Tanto *localaddr* como *remoteaddr* deben ser una tupla (*host*, *port*). El objeto hereda de *asyncore.dispatcher*, por lo que se insertará en el bucle de eventos de *asyncore* en la instancia.

data_size_limit especifica el número máximo de bytes que se aceptarán en un comando DATA. Un valor de None o 0 significa que no hay límite.

map es el mapa de conectores que se utilizará para las conexiones (un diccionario inicialmente vacío es un valor adecuado). Si no se especifica, se utiliza el mapa de socket global *asyncore*.

enable_SMTPUTF8 determina si la extensión SMTPUTF8 (como se define en [RFC 6531](#)) debe estar habilitada. El valor predeterminado es *False*. Cuando es *True*, SMTPUTF8 se acepta como parámetro para el comando MAIL y cuando está presente se pasa a *process_message()* en la lista *kwargs*['*mail_options*']. *decode_data* y *enable_SMTPUTF8* no se pueden establecer en *True* al mismo tiempo.

decode_data especifica si la porción de datos de la transacción SMTP debe decodificarse usando UTF-8. Cuando *decode_data* es *False* (el valor predeterminado), el servidor anuncia la extensión 8BITMIME ([RFC 6152](#)), acepta el parámetro BODY=8BITMIME al comando MAIL, y cuando está presente lo pasa a *process_message()* en la lista *kwargs*['*mail_options*']. *decode_data* y *enable_SMTPUTF8* no se pueden establecer en *True* al mismo tiempo.

process_message (*peer*, *mailfrom*, *rcpttos*, *data*, ***kwargs*)

Lanza una excepción *NotImplementedError*. Sobrescribe este método en subclases para hacer algo útil con este mensaje. Todo lo que se haya pasado en el constructor como *remoteaddr* estará disponible en el atributo *_remoteaddr*. *peer* es la dirección del host remoto, *mailfrom* es el creador del sobre, *rcpttos* son los destinatarios del sobre y *data* es una cadena de caracteres que contiene el contenido del correo electrónico (que debe estar en formato [RFC 5321](#)).

Si la palabra clave del constructor *decode_data* se establece en *True*, el argumento *data* será una cadena Unicode. Si se establece en *False*, será un objeto de bytes.

kwargs es un diccionario que contiene información adicional. Está vacío si se proporcionó *decode_data=True* como argumento de inicialización; de lo contrario, contiene las siguientes claves:

mail_options: una lista de todos los parámetros recibidos para el comando MAIL (los elementos son cadenas en mayúsculas; ejemplo: [*'BODY=8BITMIME'*, *'SMTPUTF8'*]).

rcpt_options: igual que *mail_options* pero para el comando RCPT. Actualmente, no se admiten las opciones RCPT TO, por lo que, por ahora, siempre será una lista vacía.

Las implementaciones de `process_message` deben usar la firma `**kwargs` para aceptar argumentos por palabra clave arbitrarios, ya que las mejoras de características futuras pueden agregar claves al diccionario `kwargs`.

Retorne `None` para solicitar una respuesta normal de 250 Ok; de lo contrario, retorne la cadena de respuesta deseada en formato [RFC 5321](#).

channel_class

Sobrescriba este método en las subclases para usar una clase `SMTPChannel` personalizada para administrar clientes SMTP.

Nuevo en la versión 3.4: El argumento del constructor *map*.

Distinto en la versión 3.5: *localaddr* y *remoteaddr* ahora pueden contener direcciones IPv6.

Nuevo en la versión 3.5: Los parámetros del constructor *decode_data* y *enable_SMTPUTF8*, y el parámetro *kwargs* para `process_message()` cuando *decode_data* es `False`.

Distinto en la versión 3.6: *decode_data* ahora es `False` por defecto.

21.18.2 Objetos `DebuggingServer`

class `smtpd.DebuggingServer` (*localaddr*, *remoteaddr*)

Crea un nuevo servidor de depuración. Los argumentos son iguales que en `SMTPServer`. Los mensajes se descartarán y se imprimirán en la salida estándar.

21.18.3 Objetos `PureProxy`

class `smtpd.PureProxy` (*localaddr*, *remoteaddr*)

Crea un nuevo servidor proxy puro. Los argumentos son iguales que en `SMTPServer`. Todo se transmitirá a *remoteaddr*. Tenga en cuenta que ejecutar esto implica una buena posibilidad de convertirlo en un relé abierto, así que tenga cuidado.

21.18.4 Objetos `MailmanProxy`

class `smtpd.MailmanProxy` (*localaddr*, *remoteaddr*)

Crea un nuevo servidor proxy puro. Los argumentos son iguales que en `SMTPServer`. Todo se transmitirá a *remoteaddr*, a menos que las configuraciones locales de mailman conozcan una dirección, en cuyo caso se manejará a través de mailman. Tenga en cuenta que ejecutar esto implica una buena posibilidad de convertirlo en un relé abierto, así que tenga cuidado.

21.18.5 Objetos `SMTPChannel`

class `smtpd.SMTPChannel` (*server*, *conn*, *addr*, *data_size_limit*=33554432, *map*=None, *enable_SMTPUTF8*=False, *decode_data*=False)

Crea un nuevo objeto `SMTPChannel` que gestiona la comunicación entre el servidor y un único cliente SMTP. *conn* y *addr* son según las variables de instancia que se describen a continuación.

data_size_limit especifica el número máximo de bytes que se aceptarán en un comando DATA. Un valor de `None` o 0 significa que no hay límite.

enable_SMTPUTF8 determina si la extensión SMTPUTF8 (como se define en [RFC 6531](#)) debe estar habilitada. El valor predeterminado es `False`. *decode_data* y *enable_SMTPUTF8* no se pueden establecer en `True` al mismo tiempo.

Se puede especificar un diccionario en *map* para evitar el uso de un mapa de socket global.

`decode_data` especifica si la porción de datos de la transacción SMTP debe decodificarse usando UTF-8. El valor predeterminado es `False`. `decode_data` y `enable_SMTPUTF8` no se pueden establecer en `True` al mismo tiempo.

Para utilizar una implementación `SMTPChannel` personalizada, debe anular `SMTPServer.channel_class` de su `SMTPServer`.

Distinto en la versión 3.5: Se agregaron los parámetros `decode_data` y `enable_SMTPUTF8`.

Distinto en la versión 3.6: `decode_data` ahora es `False` por defecto.

La clase `SMTPChannel` tiene las siguientes variables de instancia:

smtp_server

Contiene el `SMTPServer` que generó este canal.

conn

Contiene el objeto de socket que se conecta al cliente.

addr

Contiene la dirección del cliente, el segundo valor retornado por `socket.accept`

received_lines

Contiene una lista de las cadenas de línea (decodificadas mediante UTF-8) recibidas del cliente. Las líneas tienen su final de línea `"\r\n"` traducido a `"\n"`.

smtp_state

Contiene el estado actual del canal. Será `COMMAND` inicialmente y luego `DATA` después de que el cliente envíe una línea «DATA».

seen_greeting

Contiene una cadena de caracteres que contiene el saludo enviado por el cliente en su «HELO».

mailfrom

Contiene una cadena de caracteres que contiene la dirección identificada en la línea «MAIL FROM:» del cliente.

rcpttos

Contiene una lista de cadenas de caracteres que contienen las direcciones identificadas en las líneas «RCPT TO:» del cliente.

received_data

Contiene una cadena de caracteres que contiene todos los datos enviados por el cliente durante el estado de `DATA`, hasta pero sin incluir la terminación `"\r\n.\r\n"`.

fqdn

Contiene el nombre de dominio completo del servidor como lo retorna `socket.getfqdn()`.

peer

Contiene el nombre del par del cliente como lo retorna `conn.getpeername()` donde `conn` es `conn`.

La clase `SMTPChannel` opera invocando métodos llamados `smtp_<command>` al recibir una línea de comando del cliente. Construidos en la clase base `SMTPChannel`, son métodos para manejar los siguientes comandos (y responder a ellos de manera apropiada):

Co- man- do	Acción tomada
HE- LO	Acepta el saludo del cliente y lo almacena en <code>seen_greeting</code> . Establece el servidor en el modo de comando base.
EH- LO	Acepta el saludo del cliente y lo almacena en <code>seen_greeting</code> . Establece el servidor en el modo de comando extendido.
NOOP	No realiza ninguna acción.
QUIT	Cierra la conexión limpiamente.
MAIL	Acepta la sintaxis «MAIL FROM:» y almacena la dirección proporcionada como <code>mailfrom</code> . En el modo de comando extendido, acepta el atributo RFC 1870 SIZE y responde apropiadamente según el valor de <code>data_size_limit</code> .
RCPT	Acepta la sintaxis «RCPT TO:» y almacena las direcciones proporcionadas en la lista <code>rcpttos</code> .
RSET	Restablece <code>mailfrom</code> , <code>rcpttos</code> y <code>received_data</code> , pero no el saludo.
DA- TA	Establece el estado interno en DATA y almacena las líneas restantes del cliente en <code>received_data</code> hasta que se recibe el terminador <code>"\r\n.\r\n"</code> .
HELP	Retorna información mínima sobre la sintaxis del comando
VERFY	Retorna el código 252 (el servidor no sabe si la dirección es válida)
EXPN	Informa que el comando no está implementado.

21.19 telnetlib — cliente Telnet

Código fuente: [Lib/telnetlib.py](#)

El módulo `telnetlib` proporciona una clase `Telnet` que implementa el protocolo Telnet. Consulte **RFC 854** para obtener más información sobre el protocolo. Además, proporciona constantes simbólicas para los caracteres de protocolo (ver más abajo) y para las opciones telnet. Los nombres simbólicos de las opciones de telnet siguen las definiciones de `arpa/telnet.h`, con el `TELOPT_` principal eliminado. Para conocer los nombres simbólicos de las opciones que tradicionalmente no se incluyen en `arpa/telnet.h`, consulte la propia fuente del módulo.

Las constantes simbólicas para los comandos telnet son: 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 (Eliminar Character), EL (Erase Line), GA (Go Ahead), SB (Subnegotiation Begin).

class `telnetlib.Telnet` (*host=None, port=0[, timeout]*)

`Telnet` representa una conexión a un servidor Telnet. La instancia inicialmente no está conectada de forma predeterminada; el método `open()` debe utilizarse para establecer una conexión. Como alternativa, el nombre de *host* y el número de puerto opcional también se pueden pasar al constructor, en cuyo caso se establecerá la conexión con el servidor antes de que se retorne el constructor. El parámetro opcional *timeout* especifica un tiempo de espera en segundos para bloquear operaciones como el intento de conexión (si no se especifica, se usará la configuración de tiempo de espera predeterminada global).

No vuelve a abrir una instancia ya conectada.

Esta clase tiene muchos métodos `read_*()`. Tenga en cuenta que algunos de ellos generan `EOFError` cuando se lee el final de la conexión, porque pueden retornar una cadena vacía por otros motivos. Vea las descripciones individuales a continuación.

Un objeto `Telnet` es un gestor de contexto y se puede utilizar en una instrucción `with`. Cuando finaliza el bloque `with`, se llama al método `close()`:

```
>>> from telnetlib import Telnet
>>> with Telnet('localhost', 23) as tn:
...     tn.interact()
... 
```

Distinto en la versión 3.6: Soporte de gestor de contexto añadido

Ver también:

RFC 854 - Especificación del protocolo Telnet Definición del protocolo Telnet.

21.19.1 Objetos Telnet

Las instancias `Telnet` contienen los siguientes métodos:

`Telnet.read_until(expected, timeout=None)`

Lee hasta que se encuentre una cadena de bytes determinada, *expected*, o hasta que hayan pasado los segundos *timeout*.

Cuando no se encuentra ninguna coincidencia, retorna lo que esté disponible en su lugar, posiblemente bytes vacíos. Lanza `EOFError` si la conexión está cerrada y no hay datos cocinados disponibles.

`Telnet.read_all()`

Lee todos los datos hasta EOF como bytes; bloquea hasta que se cierre la conexión.

`Telnet.read_some()`

Lee al menos un byte de datos cocinados a menos que se golpee EOF. Retorna `b''` si se llega a EOF. Bloquea si no hay datos disponibles inmediatamente.

`Telnet.read_very_eager()`

Lee todo lo que puede ser sin bloquear en E/S (ansioso).

Lanza `EOFError` si la conexión está cerrada y no hay datos cocidos disponibles. Retorna `b''` si no hay datos cocinados disponibles de otra manera. No bloquea a menos que esté en medio de una secuencia IAC.

`Telnet.read_eager()`

Lee los datos disponibles.

Lanza `EOFError` si la conexión está cerrada y no hay datos cocidos disponibles. Retorna `b''` si no hay datos cocinados disponibles de otra manera. No bloquea a menos que esté en medio de una secuencia IAC.

`Telnet.read_lazy()`

Procesa y retorna datos ya en las colas (perezoso).

Lanza `EOFError` si la conexión está cerrada y no hay datos disponibles. Retorna `b''` si no hay datos cocinados disponibles de otra manera. No bloquea a menos que esté en medio de una secuencia IAC.

`Telnet.read_very_lazy()`

Retorna los datos disponibles en la cola cocida (muy perezoso).

Lanza `EOFError` si la conexión está cerrada y no hay datos disponibles. Retorna `b''` si no hay datos cocinados disponibles de otra manera. Este método nunca se bloquea.

`Telnet.read_sb_data()`

Retorna los datos recopilados entre un par SB/SE (suboptionbegin/end). La retrollamada debe tener acceso a estos datos cuando se invocó con un comando SE. Este método nunca se bloquea.

`Telnet.open(host, port=0, [timeout])`

Conecta a un host. El segundo argumento opcional es el número de puerto, que tiene como valor predeterminado el puerto Telnet estándar (23). El parámetro opcional *timeout* especifica un tiempo de espera en segundos para bloquear operaciones como el intento de conexión (si no se especifica, se usará la configuración de tiempo de espera predeterminada global).

Do not try to reopen an already connected instance.

Genera un *evento de auditoría* `telnetlib`. `Telnet.open` con argumentos `self`, `host`, `port`.

`Telnet.msg(msg, *args)`

Imprime un mensaje de depuración cuando el nivel de depuración sea > 0 . Si hay argumentos adicionales, se substituyen en el mensaje mediante el operador de formato de cadena de caracteres estándar.

`Telnet.set_debuglevel(debuglevel)`

Establece el nivel de depuración. Cuanto mayor sea el valor de *debuglevel*, más salida de depuración obtendrá (en `sys.stdout`).

`Telnet.close()`

Cierra la conexión.

`Telnet.get_socket()`

Retorna el objeto de socket utilizado internamente.

`Telnet.fileno()`

Retorna el descriptor de archivo del objeto de socket utilizado internamente.

`Telnet.write(buffer)`

Escribe una cadena de bytes en el socket, duplicando los caracteres IAC. Esto puede bloquearse si la conexión está bloqueada. Puede generar `OSError` si la conexión está cerrada.

Lanza un *evento de auditoría* `telnetlib.Telnet.write` con argumentos `self, buffer`.

Distinto en la versión 3.3: Este método se utiliza para lanzar `socket.error`, que ahora es un alias de `OSError`.

`Telnet.interact()`

Función de interacción, emula a un cliente Telnet muy tonto.

`Telnet.mt_interact()`

Versión multiproceso de `interact()`.

`Telnet.expect(list, timeout=None)`

Lee hasta que uno de una lista de expresiones regulares coincida.

El primer argumento es una lista de expresiones regulares, compiladas (*objetos regex*) o no compiladas (cadenas de bytes). El segundo argumento opcional es un tiempo de espera, en segundos; el valor predeterminado es bloquear indefinidamente.

Retorna una tupla de tres elementos: el índice de la lista de la primera expresión regular que coincide; el objeto de coincidencia retornado; y los bytes leen hasta e incluyendo la coincidencia.

Si se encuentra el final del archivo y no se leyó ningún bytes, lanza `EOFError`. De lo contrario, cuando nada coincide, retorna `(-1, None, data)` donde `data` es los bytes recibidos hasta ahora (pueden ser bytes vacíos si se ha producido un tiempo de espera).

Si una expresión regular termina con una coincidencia expansiva (como `.*`) o si más de una expresión puede coincidir con la misma entrada, los resultados no son deterministas y pueden depender de la sincronización de E/S.

`Telnet.set_option_negotiation_callback(callback)`

Cada vez que se lee una opción telnet en el flujo de entrada, se llama a esta *callback* (si se establece) con los siguientes parámetros: `callback(telnet socket, command (DO/DONT/WILL/WONT), opción)`. No hay ninguna otra acción se realiza después por `telnetlib`.

21.19.2 Ejemplo de Telnet

Un ejemplo sencillo que ilustra el uso típico:

```
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until(b"login: ")
tn.write(user.encode('ascii') + b"\n")
if password:
    tn.read_until(b"Password: ")
```

(continué en la próxima página)

(proviene de la página anterior)

```
tn.write(password.encode('ascii') + b"\n")

tn.write(b"ls\n")
tn.write(b"exit\n")

print(tn.read_all().decode('ascii'))
```

21.20 uuid — objetos UUID según RFC 4122

Código fuente: [Lib/uuid.py](#)

Este módulo proporciona objetos *UUID* inmutables (la clase *UUID*) y las funciones *uuid1()*, *uuid3()*, *uuid4()*, *uuid5()* para generar los UUID de las versiones 1, 3, 4 y 5 como se especifica en **RFC 4122**.

Si todo lo que quieres es una identificación única, probablemente deberías llamar a *uuid1()* o *uuid4()*. Ten en cuenta que *uuid1()* puede comprometer la privacidad ya que crea un UUID que contiene la dirección de red del ordenador. *uuid4()* crea un UUID aleatorio.

Dependiendo del soporte de la plataforma subyacente, *uuid1()* puede o no retornar un UUID «seguro». Un UUID seguro es aquel que se genera mediante métodos de sincronización que aseguran que ningún proceso pueda obtener el mismo UUID. Todas las instancias de *UUID* tienen un atributo *is_safe* que transmite cualquier información sobre la seguridad del UUID, usando esta enumeración:

class *uuid.SafeUUID*

Nuevo en la versión 3.7.

safe

The UUID was generated by the platform in a multiprocessing-safe way.

unsafe

El UUID no fue generado por la plataforma de una manera segura de multiprocesamiento.

unknown

La plataforma no proporciona información sobre si el UUID se generó de forma segura o no.

class *uuid.UUID* (*hex=None*, *bytes=None*, *bytes_le=None*, *fields=None*, *int=None*, *version=None*, *, *is_safe=SafeUUID.unknown*)

Crea un UUID a partir de una cadena de 32 dígitos hexadecimales, una cadena de 16 bytes en orden *big-endian* como el argumento *bytes*, una cadena de 16 bytes en orden *little-endian* como el argumento *bytes_le*, una tupla de seis enteros (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*) como el argumento *fields*, o un solo entero de 128-bit como el argumento *int*. Cuando se da una cadena de dígitos hexadecimales, los corchetes, los guiones y el prefijo URN son todos opcionales. Por ejemplo, todas estas expresiones producen el mismo UUID:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
        b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

Exactamente uno de *hex*, *bytes*, *bytes_le*, *fields*, o *int* debe ser dado. El argumento *version* es opcional; si se da, el UUID resultante tendrá su variante y número de versión establecidos de acuerdo con **RFC 4122**, anulando los bits en la *hex*, *bytes*, *bytes_le*, *fields*, o *int* dados.

La comparación de los objetos UUID se hace mediante la comparación de sus atributos *UUID.int*. La comparación con un objeto no UUID produce un *TypeError*.

`str(uuid)` retorna una cadena en la forma 12345678-1234-5678-1234-567812345678 donde los 32 dígitos hexadecimales representan el UUID.

Las instancias de `UUID` tienen estos atributos de sólo lectura:

`UUID.bytes`

El UUID como una cadena de 16 bytes (que contiene los seis campos enteros en orden de bytes *big-endian*).

`UUID.bytes_le`

El UUID como una cadena de 16 bytes (con *time_low*, *time_mid*, y *time_hi_version* en el orden de bytes *little-endian*).

`UUID.fields`

Una tupla de los seis campos enteros de la UUID, que también están disponibles como seis atributos individuales y dos atributos derivados:

Campo	Significado
<code>time_low</code>	los primeros 32 bits del UUID
<code>time_mid</code>	los siguientes 16 bits del UUID
<code>time_hi_version</code>	los siguientes 16 bits del UUID
<code>clock_seq_hi_variant</code>	los siguientes 8 bits del UUID
<code>clock_seq_low</code>	los siguientes 8 bits del UUID
<code>node</code>	los siguientes 48 bits del UUID
<code>time</code>	el timestamp de 60-bit
<code>clock_seq</code>	el número de secuencia de 14-bit

`UUID.hex`

El UUID como una cadena hexadecimal de 32 caracteres.

`UUID.int`

El UUID como un entero de 128 bits.

`UUID.urn`

El UUID como URN como se especifica en [RFC 4122](#).

`UUID.variant`

La variante UUID, que determina la disposición interna del UUID. Esta será una de las constantes `RESERVED_NCS`, `RFC_4122`, `RESERVED_MICROSOFT`, o `RESERVED_FUTURE`.

`UUID.version`

El número de versión UUID (del 1 al 5, significativo sólo cuando la variante es `RFC_4122`).

`UUID.is_safe`

Una enumeración de `SafeUUID` que indica si la plataforma generó el UUID de forma segura para el multi-procesamiento.

Nuevo en la versión 3.7.

El módulo `uuid` define las siguientes funciones:

`uuid.getnode()`

Obtiene la dirección del hardware como un entero positivo de 48 bits. La primera vez que esto se ejecute, puede lanzar un programa separado, que podría ser bastante lento. Si todos los intentos de obtener la dirección de hardware fallan, elegimos un número aleatorio de 48 bits con el bit multicast (el bit menos significativo del primer octeto) puesto a 1 como se recomienda en [RFC 4122](#). «Dirección de hardware» significa la dirección MAC de una interfaz de red. En una máquina con múltiples interfaces de red, las direcciones MAC administradas universalmente (es decir, donde el segundo bit menos significativo del primer octeto es *unset*) se preferirán a las direcciones MAC administradas localmente, pero sin otras garantías de orden.

Distinto en la versión 3.7: Se prefieren las direcciones MAC administradas universalmente a las administradas localmente, ya que se garantiza que las primeras son únicas a nivel mundial, mientras que las segundas no lo son.

`uuid.uuid1 (node=None, clock_seq=None)`

Genera un UUID a partir de un ID de host, número de secuencia y la hora actual. Si no se da *node*, se usa `getnode()` para obtener la dirección del hardware. Si se da *clock_seq*, se utiliza como número de secuencia; de lo contrario se elige un número de secuencia aleatorio de 14 bits.

`uuid.uuid3 (namespace, name)`

Genera un UUID basado en el hash MD5 de un identificador de espacio de nombres (que es un UUID) y un nombre (que es una cadena).

`uuid.uuid4 ()`

Genera un UUID aleatorio.

`uuid.uuid5 (namespace, name)`

Genera un UUID basado en el hash SHA-1 de un identificador de espacio de nombres (que es un UUID) y un nombre (que es una cadena). Generar un UUID basado en el hash SHA-1 de un identificador de espacio de nombres (que es un UUID) y un nombre (que es una cadena).

El módulo `uuid` define los siguientes identificadores de espacios de nombres para su uso con `uuid3()` o `uuid5()`.

`uuid.NAMESPACE_DNS`

Cuando se especifica este espacio de nombres, la cadena *name* es un nombre de dominio completamente calificado.

`uuid.NAMESPACE_URL`

Cuando se especifica este espacio de nombres, la cadena *name* es una URL.

`uuid.NAMESPACE_OID`

Cuando se especifica este espacio de nombres, la cadena *name* es un OID ISO.

`uuid.NAMESPACE_X500`

Cuando se especifica este espacio de nombres, la cadena *name* es un X.500 DN en DER o un formato de salida de texto.

El módulo `uuid` define las siguientes constantes para los posibles valores del atributo *variant*:

`uuid.RESERVED_NCS`

Reservado para la compatibilidad con NCS.

`uuid.RFC_4122`

Especifica el diseño del UUID dado en [RFC 4122](#).

`uuid.RESERVED_MICROSOFT`

Reservado para la compatibilidad con Microsoft.

`uuid.RESERVED_FUTURE`

Reservado para una futura definición.

Ver también:

RFC 4122 - A Universally Unique Identifier (UUID) URN Namespace Esta especificación define un espacio de nombres de recursos uniforme para los UUID, el formato interno de los UUID y los métodos de generación de los UUID.

21.20.1 Ejemplo

Aquí hay algunos ejemplos del uso típico del módulo `uuid`:

```
>>> 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

21.21 socketserver — Un framework para servidores de red

Código fuente: [Lib/socketserver.py](#)

El módulo *socketserver* simplifica la tarea de escribir servidores de red.

Hay cuatro clases básicas de servidores concretos:

class `socketserver.TCPServer` (*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
Utiliza el protocolo TCP de Internet, que proporciona flujos continuos de datos entre el cliente y el servidor. Si *bind_and_activate* es verdadero, el constructor automáticamente intenta invocar a *server_bind()* y *server_activate()*. Los otros parámetros se pasan a la clase base *BaseServer*.

class `socketserver.UDPServer` (*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
Esto utiliza datagramas, que son paquetes discretos de información que pueden llegar fuera de servicio o perderse durante el tránsito. Los parámetros son los mismos que para *TCPServer*.

class `socketserver.UnixStreamServer` (*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

class `socketserver.UnixDatagramServer` (*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

Estas clases que se usan con menos frecuencia son similares a las clases TCP y UDP, pero usan sockets de dominio Unix; no están disponibles en plataformas que no sean Unix. Los parámetros son los mismos que para *TCPServer*.

Estas cuatro clases procesan solicitudes *sincrónicamente*; cada solicitud debe completarse antes de que se pueda iniciar la siguiente. Esto no es adecuado si cada solicitud tarda mucho en completarse, porque requiere mucho cálculo o porque retorna muchos datos que el cliente tarda en procesar. La solución es crear un proceso o hilo independiente para manejar cada solicitud; las clases mixtas *ForkingMixIn* y *ThreadingMixIn* pueden usarse para soportar el comportamiento asíncronico.

La creación de un servidor requiere varios pasos. Primero, debes crear una clase de controlador de solicitudes subclassificando la clase *BaseRequestHandler* y anulando su método *handle()*; este método procesará las soli-

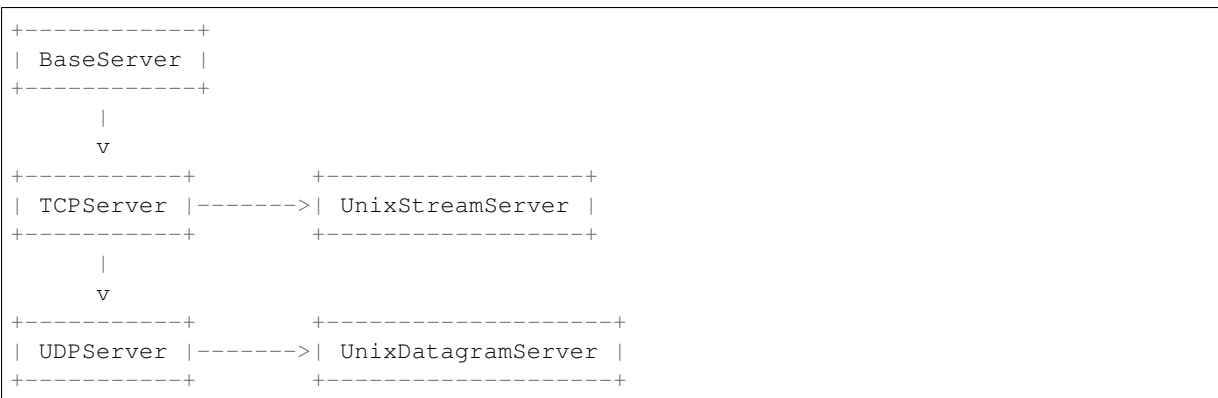
citudes entrantes. En segundo lugar, debe crear una instancia de una de las clases de servidor, pasándole la dirección del servidor y la clase del controlador de solicitudes. Se recomienda utilizar el servidor en una declaración `with`. Luego llame al método `handle_request()` o `serve_forever()` del objeto de servidor para procesar una o muchas solicitudes. Finalmente, llame a `server_close()` para cerrar el socket (a menos que haya usado una `with` declaración).

Al heredar de `ThreadingMixIn` para el comportamiento de la conexión con subprocessos, debe declarar explícitamente cómo desea que se comporten sus subprocessos en un cierre abrupto. La clase `ThreadingMixIn` define un atributo `daemon_threads`, que indica si el servidor debe esperar o no la terminación del hilo. Debe establecer la bandera explícitamente si desea que los subprocessos se comporten de forma autónoma; el valor predeterminado es `False`, lo que significa que Python no se cerrará hasta que todos los hilos creados por `ThreadingMixIn` hayan salido.

Las clases de servidor tienen los mismos métodos y atributos externos, independientemente del protocolo de red que utilicen.

21.21.1 Notas de creación del servidor

Hay cinco clases en un diagrama de herencia, cuatro de las cuales representan servidores síncronos de cuatro tipos:



Tenga en cuenta que `UnixDatagramServer` deriva de `UDPServer`, no de `UnixStreamServer` — la única diferencia entre una IP y un servidor de flujo Unix es la familia de direcciones, que simplemente se repite en ambos Clases de servidor Unix.

class `socketserver.ForkingMixIn`

class `socketserver.ThreadingMixIn`

Se pueden crear versiones de Forking y threading de cada tipo de servidor utilizando estas clases mixtas. Por ejemplo, `ThreadingUDPServer` se crea de la siguiente manera:

```

class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass

```

La clase de mezcla es lo primero, ya que anula un método definido en `UDPServer`. La configuración de los distintos atributos también cambia el comportamiento del mecanismo del servidor subyacente.

`ForkingMixIn` y las clases de Forking mencionadas a continuación solo están disponibles en plataformas POSIX que admiten `os.fork()`.

`socketserver.ForkingMixIn.server_close()` espera hasta que se completen todos los procesos secundarios, excepto si `socketserver.ForkingMixIn.block_on_close` atributo es falso.

`socketserver.ThreadingMixIn.server_close()` espera hasta que se completen todos los subprocessos que no son demonios, excepto si el atributo `socketserver.ThreadingMixIn.block_on_close` es falso. Use subprocessos demoníacos configurando `ThreadingMixIn.daemon_threads` en Verdadero para no esperar hasta que se completen los subprocessos.

Distinto en la versión 3.7: `socketserver.ForkingMixIn.server_close()` y `socketserver.ThreadingMixIn.server_close()` ahora espera hasta que se completen todos los procesos se-

cundarios y los subprocesos no demoníacos. Agregue un nuevo atributo de clase `socketserver.ForkingMixIn.block_on_close` para optar por el comportamiento anterior a 3.7.

```
class socketserver.ForkingTCPServer
class socketserver.ForkingUDPServer
class socketserver.ThreadingTCPServer
class socketserver.ThreadingUDPServer
```

Estas clases están predefinidas utilizando las clases mixtas.

Para implementar un servicio, debes derivar una clase de `BaseRequestHandler` y redefinir su método `handle()`. Luego, puede ejecutar varias versiones del servicio combinando una de las clases de servidor con su clase de controlador de solicitudes. La clase del controlador de solicitudes debe ser diferente para los servicios de datagramas o flujos. Esto se puede ocultar usando las subclases del controlador `StreamRequestHandler` o `DatagramRequestHandler`.

Por supuesto, ¡todavía tienes que usar la cabeza! Por ejemplo, no tiene sentido usar un servidor de bifurcación si el servicio contiene un estado en la memoria que puede ser modificado por diferentes solicitudes, ya que las modificaciones en el proceso hijo nunca alcanzarían el estado inicial que se mantiene en el proceso padre y se pasa a cada hijo. En este caso, puede usar un servidor de subprocesos, pero probablemente tendrá que usar bloqueos para proteger la integridad de los datos compartidos.

Por otro lado, si está creando un servidor HTTP donde todos los datos se almacenan externamente (por ejemplo, en el sistema de archivos), una clase síncrona esencialmente hará que el servicio sea «sordo» mientras se maneja una solicitud, que puede ser durante mucho tiempo si un cliente tarda en recibir todos los datos que ha solicitado. Aquí es apropiado un servidor de enhebrado o bifurcación.

En algunos casos, puede ser apropiado procesar parte de una solicitud de forma síncrona, pero para finalizar el procesamiento en un hijo bifurcado según los datos de la solicitud. Esto se puede implementar usando un servidor sincrónico y haciendo un `fork` explícito en la clase del controlador de solicitudes el método `handle()`.

Otro enfoque para manejar múltiples solicitudes simultáneas en un entorno que no admite subprocesos ni `fork()` (o donde estos son demasiado costosos o inapropiados para el servicio) es mantener una tabla explícita de solicitudes parcialmente terminadas y utilizar `selectors` para decidir en qué solicitud trabajar a continuación (o si manejar una nueva solicitud entrante). Esto es particularmente importante para los servicios de transmisión en los que cada cliente puede potencialmente estar conectado durante mucho tiempo (si no se pueden utilizar subprocesos o subprocesos). Consulte `asyncore` para ver otra forma de gestionar esto.

21.21.2 Objetos de servidor

```
class socketserver.BaseServer(server_address, RequestHandlerClass)
```

Esta es la superclase de todos los objetos de servidor en el módulo. Define la interfaz, que se indica a continuación, pero no implementa la mayoría de los métodos, que se realiza en subclases. Los dos parámetros se almacenan en los atributos respectivos `server_address` y `RequestHandlerClass`.

fileno()

Retorna un descriptor de archivo entero para el socket en el que escucha el servidor. Esta función se pasa comúnmente a `selectors`, para permitir monitorear múltiples servidores en el mismo proceso.

handle_request()

Procese una sola solicitud. Esta función llama a los siguientes métodos en orden: `get_request()`, `verify_request()`, y `process_request()`. Si el método proporcionado por el usuario `handle()` de la clase del controlador lanza una excepción, se llamará al método `handle_error()` del servidor. Si no se recibe ninguna solicitud en `timeout` segundos, se llamará a `handle_timeout()` y `handle_request()` regresará.

serve_forever(poll_interval=0.5)

Manejar solicitudes hasta una solicitud explícita `shutdown()`. Sondeo de apagado cada `poll_interval` segundos. Ignora el atributo `timeout`. También llama `service_actions()`, que puede ser utilizado por una subclase o mixin para proporcionar acciones específicas para un servicio dado. Por ejemplo, la clase `ForkingMixIn` usa `service_actions()` para limpiar procesos secundarios zombies.

Distinto en la versión 3.3: Se agregó la llamada `service_actions` al método `serve_forever`.

service_actions()

Esto se llama en el ciclo `serve_forever()`. Este método puede ser reemplazado por subclases o clases mixtas para realizar acciones específicas para un servicio dado, como acciones de limpieza.

Nuevo en la versión 3.3.

shutdown()

Dile al bucle `serve_forever()` que se detenga y espere hasta que lo haga. `shutdown()` debe llamarse mientras `serve_forever()` se está ejecutando en un hilo diferente, de lo contrario, se bloqueará.

server_close()

Limpiar el servidor. Puede ser sobrescrito.

address_family

La familia de protocolos a la que pertenece el socket del servidor. Algunos ejemplos comunes son `socket.AF_INET` y `socket.AF_UNIX`.

RequestHandlerClass

La clase de controlador de solicitudes proporcionada por el usuario; se crea una instancia de esta clase para cada solicitud.

server_address

La dirección en la que escucha el servidor. El formato de las direcciones varía según la familia de protocolos; consulte la documentación del módulo `socket` para obtener más detalles. Para los protocolos de Internet, esta es una tupla que contiene una cadena que proporciona la dirección y un número de puerto entero: `('127.0.0.1', 80)`, por ejemplo.

socket

El objeto de socket en el que el servidor escuchará las solicitudes entrantes.

Las clases de servidor admiten las siguientes variables de clase:

allow_reuse_address

Si el servidor permitirá la reutilización de una dirección. Este valor predeterminado es `False`, y se puede establecer en subclases para cambiar la política.

request_queue_size

El tamaño de la cola de solicitudes. Si toma mucho tiempo procesar una sola solicitud, cualquier solicitud que llegue mientras el servidor está ocupado se coloca en una cola, hasta `request_queue_size`. Una vez que la cola está llena, más solicitudes de clientes obtendrán un error de «Conexión denegada». El valor predeterminado suele ser 5, pero las subclases pueden anularlo.

socket_type

El tipo de socket utilizado por el servidor; `socket.SOCK_STREAM` y `socket.SOCK_DGRAM` son dos valores comunes.

timeout

Duración del tiempo de espera, medida en segundos, o `None` si no se desea un tiempo de espera. Si `handle_request()` no recibe solicitudes entrantes dentro del período de tiempo de espera, se llama al método `handle_timeout()`.

Hay varios métodos de servidor que pueden ser anulados por subclases de clases de servidor base como `TCPServer`; estos métodos no son útiles para los usuarios externos del objeto de servidor.

finish_request(request, client_address)

En realidad, procesa la solicitud creando instancias `RequestHandlerClass` y llamando a su método `handle()`.

get_request()

Debe aceptar una solicitud del socket y retornar una tupla de 2 que contenga el objeto de socket *nuevo* que se utilizará para comunicarse con el cliente y la dirección del cliente.

handle_error(request, client_address)

Esta función se llama si el método `handle()` de una instancia `RequestHandlerClass` lanza una

excepción. La acción predeterminada es imprimir el rastreo al error estándar y continuar manejando más solicitudes.

Distinto en la versión 3.6: Ahora solo se solicitan excepciones derivadas de la clase `Exception`.

handle_timeout()

Esta función se llama cuando el atributo `timeout` se ha establecido en un valor distinto de `None` y el tiempo de espera ha pasado sin que se reciban solicitudes. La acción predeterminada para los servidores de forking es recopilar el estado de cualquier proceso hijo que haya salido, mientras que en los servidores de threading este método no hace nada.

process_request(request, client_address)

Llama a `finish_request()` para crear una instancia de `RequestHandlerClass`. Si lo desea, esta función puede crear un nuevo proceso o hilo para manejar la solicitud; las clases `ForkingMixIn` y `ThreadingMixIn` hacen esto.

server_activate()

Lo llama el constructor del servidor para activar el servidor. El comportamiento predeterminado para un servidor TCP simplemente invoca `listen()` en el socket del servidor. Puede anularse.

server_bind()

Lo llama el constructor del servidor para vincular el socket a la dirección deseada. Puede anularse.

verify_request(request, client_address)

Debe retornar un valor booleano; si el valor es `True`, la solicitud se procesará, y si es `False`, la solicitud será denegada. Esta función se puede anular para implementar controles de acceso para un servidor. La implementación predeterminada siempre retorna `True`.

Distinto en la versión 3.6: Se agregó soporte para el protocolo `context manager`. Salir del administrador de contexto es equivalente a llamar a `server_close()`.

21.21.3 Solicitar objetos de controlador

class socketserver.BaseRequestHandler

Esta es la superclase de todos los objetos de manejo de solicitudes. Define la interfaz, que se muestra a continuación. Una subclase de controlador de solicitudes concreta debe definir un nuevo método `handle()` y puede anular cualquiera de los otros métodos. Se crea una nueva instancia de la subclase para cada solicitud.

setup()

Se llama antes del método `handle()` para realizar las acciones de inicialización necesarias. La implementación predeterminada no hace nada.

handle()

Esta función debe realizar todo el trabajo necesario para atender una solicitud. La implementación predeterminada no hace nada. Dispone de varios atributos de instancia; la solicitud está disponible como `self.request`; la dirección del cliente como `self.client_address`; y la instancia del servidor como `self.server`, en caso de que necesite acceder a la información por servidor.

El tipo de `self.request` es diferente para datagramas o servicios de flujo. Para los servicios de transmisión, `self.request` es un objeto de socket; para servicios de datagramas, `self.request` es un par de string y socket.

finish()

Se llama después del método `handle()` para realizar las acciones de limpieza necesarias. La implementación predeterminada no hace nada. Si `setup()` lanza una excepción, no se llamará a esta función.

class socketserver.StreamRequestHandler

class socketserver.DatagramRequestHandler

Estas subclases `BaseRequestHandler` anulan los métodos `setup()` y `finish()`, y proporcionan `self.rfile` y `self.wfile` atributos. Los atributos `self.rfile` y `self.wfile` se pueden leer o escribir, respectivamente, para obtener los datos de la solicitud o retornar los datos al cliente.

Los atributos `rfile` de ambas clases admiten la interfaz legible `io.BufferedReader`, y `DatagramRequestHandler.wfile` admite la `io.BufferedReader` interfaz de escritura.

Distinto en la versión 3.6: `StreamRequestHandler.wfile` también admite la interfaz de escritura `io.BufferedIOBase`.

21.21.4 Ejemplos

`socketserver.TCPServer` Ejemplo

Este es el lado del servidor:

```
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
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

Una clase de controlador de solicitudes alternativa que hace uso de secuencias (objetos similares a archivos que simplifican la comunicación al proporcionar la interfaz de archivo estándar):

```
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())
```

La diferencia es que la llamada `readline()` en el segundo controlador llamará a `recv()` varias veces hasta que encuentre un carácter de nueva línea, mientras que la llamada única `recv()` en el primer controlador simplemente retornará lo que se ha enviado desde el cliente en una llamada `sendall()`.

Este es el lado del cliente:

```
import socket
import sys
```

(continué en la próxima página)

(proviene de la página anterior)

```
HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent: {}".format(data))
print("Received: {}".format(received))
```

La salida del ejemplo debería verse así:

Servidor:

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

Cliente:

```
$ 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 Ejemplo

Este es el lado del servidor:

```
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)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

Este es el lado del cliente:

```

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(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent:      {}".format(data))
print("Received: {}".format(received))

```

La salida del ejemplo debería verse exactamente como en el ejemplo del servidor TCP.

Mixins asincrónicos

Para construir controladores asincrónicos, use las clases *ThreadingMixIn* y *ForkingMixIn*.

Un ejemplo para la clase *ThreadingMixIn* class

```

import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    with server:
        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()

```

(continué en la próxima página)

(proviene de la página anterior)

```
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()
```

La salida del ejemplo debería verse así:

```
$ 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
```

La clase `ForkingMixIn` se usa de la misma manera, excepto que el servidor generará un nuevo proceso para cada solicitud. Disponible solo en plataformas POSIX que admitan `fork()`.

21.22 http.server — Servidores HTTP

Código fuente: [Lib/http/server.py](#)

Este módulo define clases para implementar servidores HTTP (servidores Web).

Advertencia: `http.server` is not recommended for production. It only implements *basic security checks*.

Una clase, `HTTPServer`, es una subclase `socketserver.TCPServer`. Crea y escucha en el socket HTTP, enviando las peticiones a un handler. El código para crear y ejecutar el servidor se ve así:

```
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class `http.server.HTTPServer` (*server_address*, *RequestHandlerClass*)

Esta clase se basa en la clase `TCPServer` almacenando la dirección del servidor como variables de instancia llamadas `nombre_del_servidor` y `puerto_del_servidor`. El servidor es accesible por el handler, típicamente a través de la variable de instancia `servidor` del handler.

class `http.server.ThreadingHTTPServer` (*server_address*, *RequestHandlerClass*)

Esta clase es idéntica a `HTTPServer`, pero utiliza subprocesos para controlar las solicitudes mediante el uso de `ThreadingMixIn`. Esto es útil para controlar los sockets de pre-apertura de los navegadores web, en los que `HTTPServer` esperaría indefinidamente.

Nuevo en la versión 3.7.

El `HTTPServer` y `ThreadingHTTPServer` deben recibir un `RequestHandlerClass` en la creación de instancias, de los cuales este módulo proporciona tres variantes diferentes:

class `http.server.BaseHTTPRequestHandler` (*request*, *client_address*, *server*)

Esta clase se utiliza para controlar las solicitudes HTTP que llegan al servidor. Por sí mismo, no puede responder a ninguna solicitud HTTP real; debe ser subclase para manejar cada método de solicitud (por ejemplo, GET o POST). `BaseHTTPRequestHandler` proporciona una serie de variables de clase e instancia, y métodos para su uso por subclases.

El controlador analizará la solicitud y los encabezados y, a continuación, llamará a un método específico del tipo de solicitud. El nombre del método se construye a partir de la solicitud. Por ejemplo, para el método de solicitud SPAM, se llamará al método `do_SPAM()` sin argumentos. Toda la información relevante se almacena en variables de instancia del controlador. Las subclases no deben tener que reemplazar o extender el método `__init__()`.

`BaseHTTPRequestHandler` tiene las siguientes variables de instancia:

client_address

Contiene una tupla con el formato `(host, port)` que hace referencia a la dirección del cliente.

server

Contiene la instancia del servidor.

close_connection

Booleano que se debe establecer antes de `handle_one_request()` retorna, que indica si se puede esperar otra solicitud o si la conexión debe cerrarse.

requestline

Contiene la representación de cadena de la línea de solicitud HTTP. Se elimina el CRLF de terminación. Este atributo debe establecerse mediante `handle_one_request()`. Si no se ha procesado ninguna línea de solicitud válida, debe establecerse en la cadena vacía.

command

Contiene el comando (tipo de petición). Por ejemplo, `'GET'`.

path

Contains the request path. If query component of the URL is present, then path includes the query. Using the terminology of [RFC 3986](#), path here includes hier-part and the query.

request_version

Contiene la versión de la cadena de caracteres para la petición. Por ejemplo, `HTTP/1.0`.

headers

Contiene una instancia de la clase especificada por la variable de clase `MessageClass`. Esta instancia analiza y gestiona las cabeceras de la petición HTTP. La función `parse_headers()` de `http.client` se usa para parsear las cabeceras y requiere que la petición HTTP proporcione una cabecera válida de estilo [RFC 2822](#).

rfile

Un flujo de entrada `io.BufferedReader`, listo para leer desde el inicio de los datos de entrada opcionales.

wfile

Contiene el flujo de salida para escribir una respuesta al cliente. Se debe utilizar la adherencia apropiada al protocolo HTTP cuando se escribe en este flujo para lograr una interoperación exitosa con los clientes HTTP.

Distinto en la versión 3.6: Este es un flujo de `io.BufferedReader`.

`BaseHTTPRequestHandler` tiene los siguientes atributos:

server_version

Especifica la versión del software del servidor. Es posible que desee anular esto. El formato es de múltiples cadenas separadas por espacio en blanco, donde cada cadena es de la forma `nombre[versión]`. Por ejemplo, `BaseHTTP/0.2`.

sys_version

Contiene la versión del sistema Python, en una forma utilizable por el método `version_string` y la variable de clase `server_version`. Por ejemplo, `Python/1.4`.

error_message_format

Especifica una cadena de formato que debe ser usada por el método `send_error()` para construir una respuesta de error al cliente. La cadena se rellena por defecto con variables de `responses` basadas en el código de estado que pasó a `send_error()`.

error_content_type

Especifica el encabezado *HTTP Content-Type* de las respuestas de error enviadas al cliente. El valor predeterminado es `'text/html'`.

protocol_version

Esto especifica la versión del protocolo HTTP utilizada en las respuestas. Si se establece en `'HTTP/1.1'`, el servidor permitirá conexiones persistentes HTTP; sin embargo, el servidor *debe* incluir un encabezado exacto `Content-Length` (usando `send_header()`) en todas sus respuestas a los clientes. Para la compatibilidad con versiones anteriores, el valor predeterminado es `'HTTP/1.0'`.

MessageClass

Especifica una `email.message.Message` como clase para analizar los encabezados HTTP. Típicamente, esto no es anulado, y por defecto es `http.client.HTTPMessage`.

responses

Este atributo contiene una asignación de enteros de código de error a tuplas de dos elementos que contienen un mensaje corto y largo. Por ejemplo, `{code (shortmessage, longmessage)}`. El *shortmessage* se utiliza normalmente como la clave *message* en una respuesta de error, y *longmessage* como la clave *explain*. Es utilizado por `send_response_only()` y `send_error()` métodos.

Una instancia `BaseHTTPRequestHandler` tiene los siguientes métodos:

handle()

Llama `handle_one_request()` una vez (o, si las conexiones persistentes están habilitadas, varias veces) para manejar las peticiones HTTP entrantes. Nunca debería necesitar anularlo; en su lugar, simplemente los métodos apropiados de `do_*` ().

handle_one_request()

Este método analizará y enviará la solicitud al método apropiado `do_*` (). Nunca deberías necesitar anularlo.

handle_expect_100()

Cuando un servidor compatible con HTTP/1.1 recibe un encabezado de solicitud `Expect: 100-continue` responde con un encabezado `100 Continue` seguido de `200 OK`. Este método puede ser anulado para generar un error si el servidor no quiere que el cliente continúe. Por ejemplo, el servidor puede elegir enviar `417 Expectation Failed` como encabezado de respuesta y `return False`.

Nuevo en la versión 3.2.

send_error(code, message=None, explain=None)

Envía y registra una respuesta de error completa al cliente. El *code* numérico especifica el código de error HTTP, con *message* como una descripción opcional, corta y legible por el ser humano del error. El argumento *explain* puede ser usado para proporcionar información más detallada sobre el error; será formateado usando el atributo `error_message_format` y emitido, después de un conjunto completo de encabezados, como el cuerpo de la respuesta. El atributo `responses` contiene los valores por defecto para *message* y *explain* que se usarán si no se proporciona ningún valor; para los códigos desconocidos el valor por defecto para ambos es la cadena `???`. El cuerpo estará vacío si el método es `HEAD` o el código de respuesta es uno de los siguientes: `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`.

Distinto en la versión 3.4: La respuesta de error incluye un encabezado de longitud de contenido. Añadido el argumento *explain*.

send_response(code, message=None)

Agrega un encabezado de respuesta al búfer de encabezados y registra la solicitud aceptada. La línea de respuesta HTTP se escribe en el búfer interno, seguido de los encabezados *Server* y *Date*. Los valores de estos dos encabezados se recogen de los métodos `version_string()` y `date_time_string()`, respectivamente. Si el servidor no tiene la intención de enviar ningún otro encabezado utilizando el método `send_header()`, entonces `send_response()` debe ir seguido de una llamada `end_headers()`.

Distinto en la versión 3.3: Los encabezados se almacenan en un búfer interno y `end_headers()` debe llamarse explícitamente.

send_header (*keyword, value*)

Agrega el encabezado HTTP a un búfer interno que se escribirá en la secuencia de salida cuando se invoca `end_headers()` o `flush_headers()`. *keyword* debe especificar la palabra clave *header*, con *value* especificando su valor. Tenga en cuenta que, después de que se realizan las llamadas `send_header`, `end_headers()` DEBE llamarse para completar la operación.

Distinto en la versión 3.2: Los encabezados se almacenan en un búfer interno.

send_response_only (*code, message=None*)

Envía el encabezado de respuesta solamente, usado para los propósitos cuando la respuesta 100 Continue es enviada por el servidor al cliente. Los encabezados no se almacenan en el buffer y envían directamente el flujo de salida. Si no se especifica el *message*, se envía el mensaje HTTP correspondiente al *code* de respuesta.

Nuevo en la versión 3.2.

end_headers ()

Añade una línea en blanco (indicando el final de las cabeceras HTTP en la respuesta) al buffer de cabeceras y llama a `flush_headers()`.

Distinto en la versión 3.2: Los encabezados del buffer se escriben en el flujo de salida.

flush_headers ()

Finalmente envía los encabezados al flujo de salida y limpia el buffer interno de los cabezales.

Nuevo en la versión 3.3.

log_request (*code='-', size='-'*)

Registra una solicitud aceptada (exitosa). El *code* debe especificar el código numérico HTTP asociado a la respuesta. Si un tamaño de la respuesta está disponible, entonces debe ser pasado como el parámetro *size*.

log_error (...)

Registra un error cuando una solicitud no puede ser cumplida. Por defecto, pasa el mensaje a `log_message()`, por lo que toma los mismos argumentos (*format* y valores adicionales).

log_message (*format, ...*)

Registra un mensaje arbitrario en `sys.stderr`. Normalmente se anula para crear mecanismos personalizados de registro de errores. El argumento *format* es una cadena de formato estándar de estilo de impresión, donde los argumentos adicionales a `log_message()` se aplican como entradas al formato. La dirección ip del cliente y la fecha y hora actual son prefijadas a cada mensaje registrado.

version_string ()

Retorna la cadena de versiones del software del servidor. Esta es una combinación de los atributos `server_version` y `sys_version`.

date_time_string (*timestamp=None*)

Retorna la fecha y la hora dadas por *timestamp* (que debe ser `None` o en el formato retornado por `time.time()`), formateado para un encabezado de mensaje. Si se omite *timestamp*, utiliza la fecha y la hora actuales.

El resultado se muestra como `Sun, 06 Nov 1994 08:49:37 GMT`.

log_date_time_string ()

Retorna la fecha y la hora actuales, formateadas para el registro.

address_string ()

Retorna la dirección del cliente.

Distinto en la versión 3.3: Anteriormente, se realizó una búsqueda de nombres. Para evitar retrasos en la resolución del nombre, ahora siempre retorna la dirección IP.

class `http.server.SimpleHTTPRequestHandler` (*request, client_address, server, directory=None*)

Esta clase sirve a los archivos del directorio actual y los de abajo, mapeando directamente la estructura del directorio a las peticiones HTTP.

La carga de trabajo, como el análisis de la solicitud, lo hace la clase base `BaseHTTPRequestHandler`. Esta clase implementa las funciones `do_GET()` y `do_HEAD()`.

Los siguientes se definen como atributos de clase de `SimpleHTTPRequestHandler`:

server_version

Esto sería `"SimpleHTTP/" + __version__`, donde `__version__` se define a nivel de módulo.

extensions_map

Un diccionario que mapea los sufijos en los tipos de MIME. El valor por defecto es una cadena vacía, y se considera que es `application/octet-stream`. El mapeo se usa sin tener en cuenta las mayúsculas y minúsculas, por lo que sólo debe contener claves en minúsculas.

directory

Si no se especifica, el directorio a servir es el directorio de trabajo actual.

Una instancia `SimpleHTTPRequestHandler` tiene los siguientes métodos:

do_HEAD()

Este método sirve para el tipo de petición: 'HEAD' envía los encabezados que enviaría para la petición equivalente GET. Ver el método `do_GET()` para una explicación más completa de los posibles encabezados.

do_GET()

La solicitud se asigna a un archivo local interpretando la solicitud como una ruta relativa al directorio de trabajo actual.

Si la solicitud fue mapeada a un directorio, el directorio se comprueba para un archivo llamado `index.html` or `index.htm` (en ese orden). Si se encuentra, se retorna el contenido del archivo; de lo contrario, se genera un listado del directorio llamando al método `list_directory()`. Este método utiliza `os.listdir()` para escanear el directorio, y retorna una respuesta de error 404 si falla el `listdir()`.

Si la solicitud fue asignada a un archivo, se abre. Cualquier excepción `OSError` al abrir el archivo solicitado se asigna a un error 404, 'File not found'. Si había un encabezado 'If-Modified-Since' en la solicitud, y el archivo no fue modificado después de este tiempo, se envía una respuesta 304, 'Not Modified'. De lo contrario, el tipo de contenido se adivina llamando al método `guess_type()`, que a su vez utiliza la variable `extensions_map`, y se retorna el contenido del archivo.

Un encabezado de 'Content-type:' con el tipo de contenido adivinado, seguido de un encabezado de 'Content-Length:' con el tamaño del archivo y un encabezado de 'Last-Modified:' con el tiempo de modificación del archivo.

Luego sigue una línea en blanco que significa el final de los encabezados, y luego se imprime el contenido del archivo. Si el tipo MIME del archivo comienza con `text/` el archivo se abre en modo de texto; en caso contrario se utiliza el modo binario.

Por ejemplo, ver la implementación de la invocación de la función `test()` en el módulo `http.server`.

Distinto en la versión 3.7: Soporta la cabecera 'If-Modified-Since'.

La clase `SimpleHTTPRequestHandler` puede ser usada de la siguiente manera para crear un servidor web muy básico que sirva archivos relativos al directorio actual:

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```


`http.server` también puede ser invocado directamente usando el interruptor `-m` del intérprete con un argumento `port number`. Como en el ejemplo anterior, esto sirve a los archivos relativos al directorio actual:

```
python -m http.server 8000
```

Por defecto, el servidor se vincula a todas las interfaces. La opción `-b/--bind` especifica una dirección específica a la que se debe vincular. Tanto las direcciones IPv4 como las IPv6 están soportadas. Por ejemplo, el siguiente comando hace que el servidor se vincule sólo al localhost:

```
python -m http.server 8000 --bind 127.0.0.1
```

Nuevo en la versión 3.4: Se introdujo el argumento `--bind`.

Nuevo en la versión 3.8: El argumento `--bind` se ha mejorado para soportar IPv6

Por defecto, el servidor utiliza el directorio actual. La opción `-d/--directory` especifica un directorio al que debe servir los archivos. Por ejemplo, el siguiente comando utiliza un directorio específico:

```
python -m http.server --directory /tmp/
```

Nuevo en la versión 3.7: `--directory` especificar directorio alternativo

class `http.server.CGIHTTPRequestHandler` (*request, client_address, server*)

Esta clase se utiliza para servir tanto a los archivos como a la salida de los scripts CGI del directorio actual y del siguiente. Note que el mapeo de la estructura jerárquica de HTTP a la estructura del directorio local es exactamente como en `SimpleHTTPRequestHandler`.

Nota: Los scripts CGI ejecutados por la clase `CGIHTTPRequestHandler` no pueden ejecutar redirecciones (código HTTP 302), porque el código 200 (la salida del script sigue) se envía antes de la ejecución del script CGI. Esto adelanta el código de estado.

La clase, sin embargo, ejecutará el script CGI, en lugar de servirlo como un archivo, si adivina que es un script CGI. Sólo se usan CGI basados en directorios — la otra configuración común del servidor es tratar las extensiones especiales como denotando los scripts CGI.

Las funciones `do_GET()` y `do_HEAD()` se modifican para ejecutar scripts CGI y servir la salida, en lugar de servir archivos, si la petición lleva a algún lugar por debajo de la ruta `cgi_directories`.

La `CGIHTTPRequestHandler` define el siguiente miembro de datos:

cgi_directories

Esto por defecto es `['/cgi-bin', '/htbin']` y describe los directorios a tratar como si contuvieran scripts CGI.

La `CGIHTTPRequestHandler` define el siguiente método:

do_POST()

Este método sirve para el tipo de petición `'POST'`, sólo permitido para scripts CGI. El error 501, «Can only POST to CGI scripts», se produce cuando se intenta enviar a una url no CGI.

Tenga en cuenta que los scripts CGI se ejecutarán con UID de usuario *nobody*, por razones de seguridad. Los problemas con el script CGI serán traducidos al error 403.

`CGIHTTPRequestHandler` puede ser activado en la línea de comandos pasando la opción `--cgi`:

```
python -m http.server --cgi 8000
```

21.22.1 Security Considerations

`SimpleHTTPRequestHandler` will follow symbolic links when handling requests, this makes it possible for files outside of the specified directory to be served.

Earlier versions of Python did not scrub control characters from the log messages emitted to stderr from `python -m http.server` or the default `BaseHTTPRequestHandler.log_message` implementation. This could allow remote clients connecting to your server to send nefarious control codes to your terminal.

Nuevo en la versión 3.8.16: scrubbing control characters from log messages

21.23 `http.cookies` — Gestión del estado HTTP

Source code: <Lib/http/cookies.py>

El módulo `http.cookies` define clases para abstraer el concepto de cookies, un mecanismo de gestión de estado HTTP. Admite cookies simples de solo cadenas de caracteres y proporciona una abstracción para tener cualquier tipo de datos serializable como valor de cookie.

Anteriormente, el módulo aplicaba estrictamente las reglas de análisis descritas en las especificaciones **RFC 2109** y **RFC 2068**. Desde entonces se ha descubierto que MSIE 3.0x no sigue las reglas de caracteres descritas en esas especificaciones y también muchos navegadores y servidores actuales tienen reglas de análisis relajadas en lo que respecta al manejo de cookies. Como resultado, las reglas de análisis utilizadas son un poco menos estrictas.

El conjunto de caracteres, `string.ascii_letters`, `string.digits` y `!#$%&'*+-.^_`|~:` Denota el conjunto de caracteres válidos permitidos por este módulo en el nombre de la cookie (como `key`).

Distinto en la versión 3.3: Se permite “:” como un carácter de nombre de cookie válido.

Nota: Al encontrar una cookie no válida, se lanza `CookieError`, por lo que si los datos de su cookie provienen de un navegador, siempre debe prepararse para los datos no válidos y detectar `CookieError` en el análisis.

exception `http.cookies.CookieError`

Error de excepción debido a **RFC 2109** invalidity: atributos incorrectos, encabezado `Set-Cookie` incorrecto, etc.

class `http.cookies.BaseCookie` (`[input]`)

Esta clase es un objeto similar a un diccionario cuyas claves son cadenas de caracteres y cuyos valores son `Morsel`. Tenga en cuenta que al establecer una clave en un valor, el valor se convierte primero en `Morsel` que contiene la clave y el valor.

Si se proporciona `input`, se pasa al método `load()`.

class `http.cookies.SimpleCookie` (`[input]`)

Esta clase se deriva de `BaseCookie` y anula `value_decode()` y `value_encode()`. `SimpleCookie` admite cadenas de caracteres como valores de cookies. Al establecer el valor, `SimpleCookie` llama al incorporado `str()` para convertir el valor en una cadenas de caracteres. Los valores recibidos de HTTP se mantienen como cadenas de caracteres.

Ver también:

Módulo `http.cookiejar` Manejo de cookies HTTP para web *clients*. Los módulos `http.cookiejar` and `http.cookies` no dependen el uno del otro.

RFC 2109 - Mecanismo de gestión de estado HTTP Esta es la especificación de gestión de estado implementada por este módulo.

21.23.1 Objetos de cookie

`BaseCookie.value_decode(val)`

Retorna una tupla (`real_value`, `coded_value`) de una representación de cadena de caracteres. `real_value` puede ser de cualquier tipo. Este método no decodifica en `BaseCookie` — existe por lo que puede ser anulado.

`BaseCookie.value_encode(val)`

Retorna una tupla (`real_value`, `coded_value`). `val` puede ser de cualquier tipo, pero `coded_value` siempre se convertirá en una cadena de caracteres. Este método no codifica en `BaseCookie` — existe por lo que se puede anular.

En general, debería darse el caso de que `value_encode()` y `value_decode()` sean inversas en el rango de `value_decode`.

`BaseCookie.output(attrs=None, header='Set-Cookie:', sep='\r\n')`

Retorna una representación de cadena de caracteres adecuada para enviarse como encabezados HTTP. `attrs` y `header` se envían a cada método `Morsel's output()`. `sep` se usa para unir los encabezados y es por defecto la combinación `'\r\n'` (CRLF).

`BaseCookie.js_output(attrs=None)`

Retorna un fragmento de código JavaScript que, si se ejecuta en un navegador que admita JavaScript, actuará de la misma forma que si se enviaran los encabezados HTTP.

El significado de `attrs` es el mismo que en `output()`.

`BaseCookie.load(rawdata)`

Si `rawdata` es una cadena de caracteres, analízela como un `HTTP_COOKIE` y agregue los valores que se encuentran allí como `Morsels`. Si es un diccionario, equivale a:

```
for k, v in rawdata.items():
    cookie[k] = v
```

21.23.2 Objetos Morsel

`class http.cookies.Morsel`

Resumen de un par clave/valor, que tiene algunos atributos **RFC 2109**.

Los `Morsels` son objetos similares a diccionarios, cuyo conjunto de claves es constante — los atributos válidos **RFC 2109**, que son

- `expires`
- `path`
- `comment`
- `domain`
- `max-age`
- `secure`
- `version`
- `httponly`
- `samesite`

El atributo `httponly` especifica que la cookie solo se transfiere en solicitudes HTTP y no es accesible a través de JavaScript. Esto tiene como objetivo mitigar algunas formas de secuencias de comandos entre sitios.

El atributo `samesite` especifica que el navegador no puede enviar la cookie junto con solicitudes entre sitios. Esto ayuda a mitigar los ataques CSRF. Los valores válidos para este atributo son «Strict» y «Lax».

Las claves no distinguen entre mayúsculas y minúsculas y su valor predeterminado es `' '`.

Distinto en la versión 3.5: `__eq__()` ahora toma *key* y *value* en cuenta.

Distinto en la versión 3.7: Los atributos *key*, *value* y *coded_value* son de solo lectura. Utilice `set()` para configurarlos.

Distinto en la versión 3.8: Se agregó soporte para el atributo `samesite`.

Morsel.value

El valor de la cookie.

Morsel.coded_value

El valor codificado de la cookie — esto es lo que se debe enviar.

Morsel.key

El nombre de la cookie.

Morsel.set (*key*, *value*, *coded_value*)

Establezca los atributos *key*, *value* y *coded_value*.

Morsel.isReservedKey (*K*)

Si *K* es miembro del conjunto de claves de una *Morsel*.

Morsel.output (*attrs=None*, *header='Set-Cookie:'*)

Retorna una representación de cadena de caracteres del Morsel, adecuada para enviarse como un encabezado HTTP. De forma predeterminada, se incluyen todos los atributos, a menos que se proporcione *attrs*, en cuyo caso debería ser una lista de atributos a utilizar. *header* es por defecto `"Set-Cookie:"`.

Morsel.js_output (*attrs=None*)

Retorna un fragmento de código JavaScript que, si se ejecuta en un navegador que admita JavaScript, actuará de la misma forma que si se hubiera enviado el encabezado HTTP.

El significado de *attrs* es el mismo que en `output()`.

Morsel.OutputString (*attrs=None*)

Retorna una cadena de caracteres que representa el Morsel, sin ningún HTTP o JavaScript circundante.

El significado de *attrs* es el mismo que en `output()`.

Morsel.update (*values*)

Actualice los valores en el diccionario Morsel con los valores en el diccionario *values*. Lanza un error si alguna de las claves en el *values* dict no es un atributo válido **RFC 2109**.

Distinto en la versión 3.5: se lanza un error para claves no válidas.

Morsel.copy (*value*)

Retorna una copia superficial del objeto Morsel.

Distinto en la versión 3.5: retorna un objeto Morsel en lugar de un dict.

Morsel.setdefault (*key*, *value=None*)

Lanza un error si la clave no es un atributo válido **RFC 2109**; de lo contrario, se comporta igual que `dict.setdefault()`.

21.23.3 Ejemplo

El siguiente ejemplo demuestra cómo utilizar el módulo `http.cookies`.

```
>>> from http import cookies
>>> C = cookies.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
```

(continué en la próxima página)

(proviene de la página anterior)

```

Set-Cookie: sugar=wafer
>>> C = cookies.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 = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\\"Loves\\\"; fudge=\\\"012;\";')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\\\"Loves\\\"; fudge=\\\"012;\"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = cookies.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

```

21.24 http.cookiejar — Cookie handling for HTTP clients

Source code: Lib/http/cookiejar.py

The `http.cookiejar` 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. `http.cookiejar` 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.

Nota: 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 `http.cookiejar.LoadError`

Instances of `FileCookieJar` raise this exception on failure to load cookies from a file. `LoadError` is a subclass of `OSError`.

Distinto en la versión 3.3: `LoadError` was made a subclass of `OSError` instead of `IOError`.

The following classes are provided:

class `http.cookiejar.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 `http.cookiejar.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*.

Distinto en la versión 3.8: The `filename` parameter supports a *path-like object*.

class `http.cookiejar.CookiePolicy`

This class is responsible for deciding whether each cookie should be accepted from / returned to the server.

class `http.cookiejar.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, secure_protocols=("https", "wss")*)

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. *secure_protocols* is a sequence of protocols for which secure cookies can be added to. By default `https` and `wss` (secure websocket) are considered secure protocols. 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 `http.cookiejar.Cookie`

This class represents Netscape, **RFC 2109** and **RFC 2965** cookies. It is not expected that users of `http.cookiejar` construct their own `Cookie` instances. Instead, if necessary, call `make_cookies()` on a `CookieJar` instance.

Ver también:

Module `urllib.request` URL opening with automatic cookie handling.

Module `http.cookies` HTTP cookie classes, principally useful for server-side code. The `http.cookiejar` and `http.cookies` 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 `http.cookiejar`) 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

21.24.1 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 `urllib.request.Request` instance) must support the methods `get_full_url()`, `get_host()`, `get_type()`, `unverifiable()`, `has_header()`, `get_header()`, `header_items()`, `add_unredirected_header()` and `origin_req_host` attribute as documented by `urllib.request`.

Distinto en la versión 3.3: *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

`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 `urllib.request.urlopen()`, or similar) should support an `info()` method, which returns an `email.message.Message` instance.

The *request* object (usually a `urllib.request.Request` instance) must support the methods `get_full_url()`, `get_host()`, `unverifiable()`, and `origin_req_host` attribute, as documented by `urllib.request`. The request is used to set default values for cookie-attributes as well as for checking that the cookie is allowed to be set.

Distinto en la versión 3.3: *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

`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, `OSError` may be raised, for example if the file does not exist.

Distinto en la versión 3.3: `IOError` used to be raised, it is now an alias of `OSError`.

`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.

21.24.2 FileCookieJar subclasses and co-operation with web browsers

The following *CookieJar* subclasses are provided for reading and writing.

class `http.cookiejar.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).

Nota: This loses information about **RFC 2965** cookies, and also about newer or non-standard cookie-attributes such as `port`.

Advertencia: 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 `http.cookiejar.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.

Distinto en la versión 3.8: The filename parameter supports a *path-like object*.

21.24.3 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 *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 *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).

21.24.4 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 http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.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.

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`.

21.24.5 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 `http.cookiejar` 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 `http.cookiejar` 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 `http.cookiejar` may “downgrade” RFC 2109 cookies to Netscape cookies, in which case *version* is 0.

`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.

21.24.6 Examples

The first example shows the most common usage of `http.cookiejar`:

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.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, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.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 urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

21.25 xmlrpc — Módulos XMLRPC para cliente y servidor

XML-RPC es un método de Llamada a Procedimiento Remoto (o RPC en inglés) que usa XML usando HTTP como vía de transporte. Con esto, un cliente puede llamar a método con parámetros en un servidor remoto (el servidor se identifica mediante una URI) y traer de vuelta datos de forma estructurada.

`xmlrpc` es un paquete que agrupa los módulos, tanto de cliente como de servidor, que implementan XML-RPC. Los módulos son:

- `xmlrpc.client`
- `xmlrpc.server`

21.26 `xmlrpc.client` — acceso cliente XML-RPC

Source code: [Lib/xmlrpc/client.py](#)

XML-RPC es un método de llamada a procedimiento remoto que utiliza XML pasado a través de HTTP(S) como transporte. Con él, un cliente puede llamar a métodos con parámetros en un servidor remoto (el servidor es nombrado por un URI) y recuperar datos estructurados. Este módulo admite la escritura de código de cliente XML-RPC; maneja todos los detalles de la traducción entre objetos de Python conformes y XML en el cable.

Advertencia: El módulo `xmlrpc.client` no es seguro contra datos contruidos maliciosamente. Si necesita analizar datos que no son de confianza o no autenticados, consulte [Vulnerabilidades XML](#).

Distinto en la versión 3.5: Para HTTPS URI, `xmlrpc.client` ahora realiza todas las comprobaciones necesarias de certificados y nombres de host de forma predeterminada.

```
class xmlrpc.client.ServerProxy(uri, transport=None, encoding=None, verbose=False, allow_none=False, use_datetime=False, use_builtin_types=False, *, headers=(), context=None)
```

Una `ServerProxy` instancia un objeto que gestiona la comunicación con un servidor XML-RPC remoto. El primer argumento requerido es un URI (*Uniform Resource Indicator*) y normalmente será la URL del servidor. El segundo argumento opcional es una instancia de fábrica de transporte; de forma predeterminada, es una instancia interna `SafeTransport` para https: URL y una instancia interna `HTTP Transport` en caso contrario. El tercer argumento opcional es una codificación, por defecto UTF-8. El cuarto argumento opcional es un indicador de depuración.

Los siguientes parámetros rigen el uso de la instancia de proxy retornada. Si `allow_none` es verdadero, la constante de Python `None` se traducirá a XML; el comportamiento predeterminado es que `None` genere un `TypeError`. Esta es una extensión de uso común para la especificación XML-RPC, pero no todos los clientes y servidores la admiten; ver <http://ontosys.com/xml-rpc/extensions.php> <<https://web.archive.org/web/20130120074804/http://ontosys.com/xml-rpc/extensions.php>> _ para una descripción. La flag `use_builtin_types` puede usarse para hacer que los valores de fecha/hora se presenten como : clase: objetos `datetime.datetime` y datos binarios que se presenten como objetos `datetime.datetime` y los datos binarios pueden ser presentados como objetos `bytes`; esta flag es falsa por defecto. Los objetos `datetime.datetime`, `bytes` y `bytearray` se pueden pasar a las llamadas. El parámetro `header` es una secuencia opcional de encabezados HTTP para enviar con cada solicitud, expresada como una secuencia de 2 tuplas que representan el nombre y el valor del encabezado. (por ejemplo, `[("Header-Name", "value")]`). La flag obsoleta `use_datetime` es similar a `use_builtin_types` pero solo se aplica a los valores de fecha/hora.

Distinto en la versión 3.3: La flag `use_builtin_types` fue añadida.

Distinto en la versión 3.8: El parámetro `headers` fue añadida.

Tanto los transportes HTTP como HTTPS admiten la extensión de sintaxis de URL para la autenticación básica `HTTP:http://user:pass@host:port/path`. La parte `user:pass` se codificará en base64 como un encabezado HTTP de “Authorization” y se enviará al servidor remoto como parte del proceso de conexión al invocar un método XML-RPC. Solo necesita usar esto si el servidor remoto requiere un usuario y contraseña de autenticación básica. Si se proporciona una URL HTTPS, el `context` puede ser `ssl.SSLContext` y configura la configuración SSL de la conexión HTTPS subyacente.

La instancia retornada es un objeto proxy con métodos que se pueden utilizar para invocar las correspondientes llamadas RPC en el servidor remoto. Si el servidor remoto admite la API de introspección, el proxy también se puede utilizar para consultar al servidor remoto los métodos que admite (descubrimiento de servicios) y recuperar otros metadatos asociados al servidor.

Los tipos que son conformes (por ejemplo, que se pueden clasificar a través de XML) incluyen lo siguiente (y, excepto donde se indique, no se clasifican como el mismo tipo de Python):

Tipo XML-RPC	Tipo de Python
boolean	<i>bool</i>
int, i1, i2, i4, i8 or biginteger	<i>int</i> en el rango de -2147483648 a 2147483647. Los valores obtienen la etiqueta <code><int></code> .
double o float	<i>float</i> . Los valores obtienen la etiqueta <code><double></code> .
string	<i>str</i>
array	<i>list</i> o <i>tuple</i> que contiene elementos determinados. Las matrices se retornan como <i>lists</i> .
struct	<i>dict</i> . Las claves deben ser cadenas de caracteres, los valores pueden ser de cualquier tipo determinado. Pueden pasarse objetos de clases definidas por el usuario; sólo se transmite su atributo <code>__dict__</code> .
dateTime.iso8601	<i>DateTime</i> o <i>datetime.datetime</i> . El tipo retornado depende de los valores de los indicadores <i>use_builtin_types</i> y <i>use_datetime</i> .
base64	<i>Binary</i> , <i>bytes</i> o <i>bytearray</i> . El tipo retornado depende del valor de la marca <i>use_builtin_types</i> .
nil	La constante <code>None</code> . Solo se permite pasar si <i>allow_none</i> es verdadero.
bigdecimal	<i>decimal.Decimal</i> . Retornado solo el tipo.

Este es el conjunto completo de tipos de datos admitidos por XML-RPC. Las llamadas a métodos también pueden generar una instancia especial *Fault*, que se usa para señalar errores del servidor XML-RPC, o *ProtocolError* que se usa para señalar un error en la capa de transporte HTTP/HTTPS. Ambos *Fault* y *ProtocolError* derivan de una clase base llamada *Error*. Tenga en cuenta que el módulo de cliente *xmlrpc* actualmente no clasifica instancias de subclases de tipos integrados.

Al pasar cadenas de caracteres, los caracteres especiales de XML como `<`, `>` y `&` se escaparán automáticamente. Sin embargo, es responsabilidad de la persona que llama asegurarse de que la cadena de caracteres esté libre de caracteres que no están permitidos en XML, como los caracteres de control con valores ASCII entre 0 y 31 (excepto, por supuesto, tabulación, nueva línea y retorno de carro); no hacer esto resultará en una solicitud XML-RPC que no es XML bien formado. Si tiene que pasar bytes arbitrarios a través de XML-RPC, use las clases *bytes* o *bytearray* o la clase contenedora *Binary* descrita a continuación.

Server se conserva como un alias para *ServerProxy* para compatibilidad con versiones anteriores. El nuevo código debe usar *ServerProxy*.

Distinto en la versión 3.5: Se agregó el argumento *context*.

Distinto en la versión 3.6: Se agregó soporte para etiquetas de tipo con prefijos (por ejemplo. `ex:nil`). Se agregó soporte para desagrupar los tipos adicionales utilizados por la implementación Apache XML-RPC para números: *i1*, *i2*, *i8*, *biginteger*, *float* y *bigdecimal*. Consulte <http://ws.apache.org/xmlrpc/types.html> para obtener una descripción.

Ver también:

XML-RPC HOWTO Una buena descripción del funcionamiento de XML-RPC y del software cliente en varios idiomas. Contiene prácticamente todo lo que un desarrollador de cliente XML-RPC necesita saber.

XML-RPC Introspection Describe la extensión del protocolo XML-RPC para la introspección.

XML-RPC Specification La especificación oficial.

Unofficial XML-RPC Errata «Las erratas no oficiales de Fredrik Lundh, destinadas a aclarar ciertos detalles en la especificación XML-RPC, así como dar pistas sobre las “mejores prácticas” para usar al diseñar sus propias implementaciones XML-RPC».

21.26.1 Objetos *ServerProxy*

La instancia de *ServerProxy* tiene un método correspondiente a cada llamada de procedimiento remoto aceptada por el servidor XML-RPC. Llamar al método realiza un RPC, enviado por nombre y firma de argumento (por ejemplo, el mismo nombre de método puede sobrecargarse con múltiples firmas de argumento). El RPC finaliza retornando un valor, que puede ser datos retornados en un tipo conforme o un objeto *Fault* o *ProtocolError* que indica un error.

Los servidores que admiten la API de introspección XML admiten algunos métodos comunes agrupados bajo el atributo reservado `system`:

`ServerProxy.system.listMethods()`

Este método retorna una lista de cadenas, una para cada método (que no es del sistema) admitido por el servidor XML-RPC.

`ServerProxy.system.methodSignature(name)`

Este método toma un parámetro, el nombre de un método implementado por el servidor XML-RPC. Retorna una matriz de posibles firmas para este método. Una firma es una variedad de tipos. El primero de estos tipos es el tipo de retorno del método, el resto son parámetros.

Debido a que se permiten múltiples firmas (es decir, sobrecarga), este método retorna una lista de firmas en lugar de un singleton.

Las propias firmas están restringidas a los parámetros de nivel superior esperados por un método. Por ejemplo, si un método espera una matriz de estructuras como parámetro y retorna una cadena de caracteres, su firma es simplemente «cadena, matriz». Si espera tres enteros y retorna una cadena, su firma es «string, int, int, int».

Si no se define una firma para el método, se retorna un valor que no es una matriz. En Python, esto significa que el tipo de valor retornado será diferente a una lista.

`ServerProxy.system.methodHelp(name)`

Este método toma un parámetro, el nombre de un método implementado por el servidor XML-RPC. Retorna una cadena de caracteres de documentación que describe el uso de ese método. Si no hay tal cadena de caracteres disponible, se retorna una cadena de caracteres vacía. La cadena de caracteres de documentación puede contener marcado HTML.

Distinto en la versión 3.5: Las instancias de *ServerProxy* admiten el protocolo *context manager* para cerrar el transporte subyacente.

A continuación se muestra un ejemplo práctico. El código del servidor:

```
from xmlrpc.server 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()
```

El código de cliente para el servidor anterior:

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```


21.26.2 Objetos *DateTime*

class xmlrpc.client.DateTime

Esta clase puede inicializarse con segundos desde la época, una tupla de tiempo, una cadena de fecha/hora ISO 8601 o una instancia `datetime.datetime`. Tiene los siguientes métodos, soportados principalmente para uso interno por el código de clasificación/eliminación de clasificación:

decode (*string*)

Acepta una cadena de caracteres como el nuevo valor de tiempo de la instancia.

encode (*out*)

Escribe la codificación XML-RPC de este elemento *DateTime* en el objeto de flujo *out*.

También es compatible con algunos de los operadores integrados de Python a través de una rica comparación y métodos `__repr__()`.

A continuación se muestra un ejemplo práctico. El código del servidor:

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

El código de cliente para el servidor anterior:

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.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"))
```

21.26.3 Objetos binarios

class xmlrpc.client.Binary

Esta clase puede inicializarse a partir de datos de bytes (que pueden incluir NUL). El acceso principal al contenido de un objeto *Binary* lo proporciona un atributo:

data

Los datos binarios encapsulados por la instancia *Binary*. Los datos se proporcionan como un objeto *bytes*.

Los objetos *Binary* tienen los siguientes métodos, soportados principalmente para uso interno por el código de clasificación/desagrupación:

decode (*bytes*)

Acepta un objeto base64 *bytes* y se descodifica como los nuevos datos de la instancia.

encode (*out*)

Escribe la codificación XML-RPC base 64 de este elemento binario en el objeto de flujo *out*.

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.

También admite algunos de los operadores integrados de Python a través de los métodos `__eq__()` and `__ne__()`.

Ejemplo de uso de los objetos binarios. Vamos a transferir una imagen sobre XMLRPC:

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

El cliente obtiene la imagen y la guarda en un archivo:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

21.26.4 Objetos Faults

class `xmlrpc.client.Fault`

Un objeto *Fault* encapsula el contenido de una etiqueta de error XML-RPC. Los objetos de error tienen los siguientes atributos:

faultCode

Una cadena de caracteres que indica el tipo de fallo.

faultString

Una cadena de caracteres que contiene un mensaje de diagnóstico asociado con el fallo.

En el siguiente ejemplo vamos a causar intencionalmente un *Fault* al retornar un objeto de tipo complejo. El código del servidor:

```
from xmlrpc.server 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()
```

El código de cliente para el servidor anterior:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
```

(continué en la próxima página)

(proviene de la página anterior)

```
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

21.26.5 Objetos ProtocolError

class `xmlrpc.client.ProtocolError`

El objeto *ProtocolError* describe un error de protocolo en la capa de transporte subyacente (como un error 404 “no encontrado” si el servidor nombrado por el URI no existe). Tiene los siguientes atributos:

url

El URI o URL que provocó el error.

errcode

El código de error.

errmsg

El mensaje de error o la cadena de caracteres de diagnóstico.

headers

Un diccionario que contiene los encabezados de la solicitud HTTP/HTTPS que desencadenó el error.

En el siguiente ejemplo, vamos a causar intencionalmente un *ProtocolError* proporcionando un URI inválido:

```
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.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)
```

21.26.6 Objetos MultiCall

El objeto *MultiCall* proporciona una forma de encapsular múltiples llamadas a un servidor remoto en una sola solicitud¹.

class `xmlrpc.client.MultiCall(server)`

Crea un objeto usado para llamadas al método `boxcar`. *server* es el objetivo final de la llamada. Se pueden realizar llamadas al objeto de resultado, pero retornarán inmediatamente `None` y solo almacenarán el nombre y los parámetros de la llamada en el objeto *MultiCall*. Llamar al objeto en sí hace que todas las llamadas almacenadas se transmitan como una única solicitud de `system.multicall`. El resultado de esta llamada es un *generator*; iterar sobre este generador produce los resultados individuales.

A continuación se muestra un ejemplo de uso de esta clase. El código del servidor:

```
from xmlrpc.server import SimpleXMLRPCServer
```

(continué en la próxima página)

¹ Este enfoque se presentó por primera vez en una discusión en xmlrpc.com.

(proviene de la página anterior)

```
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()
```

El código de cliente para el servidor anterior:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.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))
```

21.26.7 Funciones de Conveniencia

`xmlrpc.client.dumps` (*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

Convierta *params* en una solicitud XML-RPC, o en una respuesta si *methodresponse* es verdadero. *params* puede ser una tupla de argumentos o una instancia de la clase de excepción *Fault*. Si *methodresponse* es verdadero, solo se puede devolver un único valor, lo que significa que *params* debe tener una longitud de 1. *encoding*, si se proporciona, es la codificación que se utilizará en el XML generado; el predeterminado es UTF-8. El valor de Python *None* no se puede usar en XML-RPC estándar; para permitir su uso a través de una extensión, proporcione un valor verdadero para *allow_none*.

`xmlrpc.client.loads` (*data*, *use_datetime=False*, *use_builtin_types=False*)

Convierte una solicitud o respuesta XML-RPC en objetos Python, un (*params*, *methodname*). *params* es una tupla de argumento; *methodname* es una cadena de caracteres, o *None* si no hay ningún nombre de método presente en el paquete. Si el paquete XML-RPC representa una condición de falla, esta función generará una excepción *Fault*. La flag *use_builtin_types* puede usarse para hacer que los valores de fecha/hora se presenten como objetos de *datetime.datetime* y datos binarios que se presenten como objetos de *bytes*; esta flag es falsa por defecto.

La flag obsoleta *use_datetime* es similar a *use_builtin_types* pero esto aplica solo a valores fecha/hora.

Distinto en la versión 3.3: La flag *use_builtin_types* fue añadida.

21.26.8 Ejemplo de Uso de Cliente

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

Para acceder a un servidor XML-RPC a través de un proxy HTTP, debe definir un transporte personalizado. El siguiente ejemplo muestra cómo:

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com',
↳transport=transport)
print(server.examples.getStateName(41))
```

21.26.9 Ejemplo de Uso de Cliente y Servidor

Vea *Ejemplo de SimpleXMLRPCServer*.

Pie de notas

21.27 xmlrpc.server — Servidores básicos XML-RPC

Código fuente: [Lib/xmlrpc/server.py](#)

El módulo `xmlrpc.server` proporciona un marco de servidor básico para servidores XML-RPC escritos en Python. Los servidores pueden ser independientes, utilizando *SimpleXMLRPCServer*, o integrados en un entorno CGI, utilizando *CGIXMLRPCRequestHandler*.

Advertencia: El módulo `xmlrpc.server` no es seguro contra datos contruidos maliciosamente. Si necesita analizar sintácticamente datos no confiables o no autenticados, consulte [Vulnerabilidades XML](#).

```
class xmlrpc.server.SimpleXMLRPCServer(addr, requestHandler=SimpleXMLRPCRequestHandler, logRequests=True, allow_none=False, encoding=None, bind_and_activate=True, use_built_in_types=False)
```

Crea una nueva instancia de servidor. Esta clase proporciona métodos para el registro de funciones que pueden ser llamados por el protocolo XML-RPC. El parámetro *requestHandler* debe ser un generador para las instancias del controlador de solicitudes; el valor predeterminado es *SimpleXMLRPCRequestHandler*. Los parámetros *addr* y *requestHandler* se pasan al constructor *socketserver.TCPServer*. Si *logRequests* es verdadero (valor predeterminado), se registrarán las solicitudes; establecer este parámetro como falso desactivará el registro. Los parámetros *allow_none* y *encoding* se pasan a *xmlrpc.client* y controlan las respuestas XML-RPC que se devolverán desde el servidor. El parámetro *bind_and_activate* controla si *server_bind()* y *server_activate()* son llamados inmediatamente por el constructor; por defecto es verdadero. Establecerlo como false permite que el código manipule la variable de clase *allow_reuse_address* antes de enlazar la dirección. El parámetro *use_built_in_types* se pasa a la función *loads()* y controla qué tipos se procesan cuando se reciben valores de fecha y hora o datos binarios; el valor predeterminado es false.

Distinto en la versión 3.3: Se ha añadido el indicador *use_built_in_types*.

```
class xmlrpc.server.CGIXMLRPCRequestHandler(allow_none=False, encoding=None, use_built_in_types=False)
```

Crea una nueva instancia para gestionar solicitudes XML-RPC en un entorno CGI. Los parámetros *allow_none* y *encoding* se pasan a *xmlrpc.client* y controlan las respuestas XML-RPC que se devolverán desde el servidor. El parámetro *use_built_in_types* se pasa a la función *loads()* y controla qué tipos se procesan cuando se reciben valores de fecha y hora o datos binarios; el valor predeterminado es falso.

Distinto en la versión 3.3: Se ha añadido el indicador *use_built_in_types*.

```
class xmlrpc.server.SimpleXMLRPCRequestHandler
```

Crea una nueva instancia del controlador de solicitudes. Este controlador de solicitudes admite solicitudes POST y modifica el registro para que se respete el parámetro *logRequests* del parámetro constructor *SimpleXMLRPCServer*.

21.27.1 Objetos SimpleXMLRPCServer

La clase *SimpleXMLRPCServer* se basa en *socketserver.TCPServer* y proporciona un medio para crear servidores XML-RPC simples e independientes.

```
SimpleXMLRPCServer.register_function(function=None, name=None)
```

Registra una función que pueda responder a solicitudes XML-RPC. Si se proporciona *name*, este será el nombre del método asociado con *function*, en otro caso se utilizará *function.__name__*. *name* es una cadena de texto, y puede contener caracteres no permitidos en los identificadores de Python, incluido el carácter de punto.

Este método también se puede utilizar como decorador. Cuando se usa como decorador, *name* solo se puede dar como un argumento de palabra clave para registrar *function* bajo *nombre*. Si no se proporciona *name*, se usará *function.__name__*.

Distinto en la versión 3.7: *register_function()* puede ser usado como decorador.

```
SimpleXMLRPCServer.register_instance(instance, allow_dotted_names=False)
```

Registre un objeto que se usa para exponer nombre de métodos que no se han registrado usando *register_function()*. Si *instance* contiene un método *_dispatch()*, este será llamado con el nombre del método solicitado y los parámetros de la solicitud. Su API es *def _dispatch(self, method, params)* (tenga en cuenta que *params* no representa una lista de argumentos variables). Si se invoca a una función subyacente para realizar su tarea, esa función es llamada como *func(*params)*, expandiendo la lista de parámetros. El valor de retorno de *_dispatch()* se retorna al cliente como resultado. Si *instance* no tiene un método *_dispatch()*, se busca un atributo que coincida con el nombre del método solicitado.

Si el argumento opcional *allow_dotted_names* es verdadero y la instancia no tiene un método *_dispatch()*, entonces si el nombre solicitado contiene puntos, cada componente del nombre del método se busca individualmente, con el efecto que produce una búsqueda jerárquica simple. El valor encontrado en esta búsqueda es entonces llamado con los parámetros de la solicitud y el valor de retorno se devuelve al cliente.

Advertencia: Habilitando la opción `allow_dotted_names` permite a los intrusos acceder a las variables globales de su módulo y puede permitir que los intrusos ejecuten código arbitrario en su máquina. Utilice esta opción únicamente en una red cerrada y segura.

`SimpleXMLRPCServer.register_introspection_functions()`

Registre las funciones de introspección XML-RPC `system.listMethods`, `system.methodHelp` y `system.methodSignature`.

`SimpleXMLRPCServer.register_multicall_functions()`

Registre la función de llamada múltiple XML-RPC `system.multicall`.

`SimpleXMLRPCRequestHandler.rpc_paths`

Un valor de atributo que debe ser una tupla que enumere porciones de ruta válidas de la URL para recibir solicitudes XML-RPC. Las solicitudes publicadas en otras rutas darán como resultado un error HTTP 404 «no existe tal página». Si esta tupla está vacía, todas las rutas se considerarán válidas. El valor predeterminado es `('/', '/RPC2')`.

Ejemplo de SimpleXMLRPCServer

Código del servidor:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    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 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

# Run the server's main loop
server.serve_forever()
```

El siguiente código de cliente llamará a los métodos disponibles por el servidor anterior:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3)) # Returns 2**3 = 8
print(s.add(2,3)) # Returns 5
```

(continué en la próxima página)

(proviene de la página anterior)

```
print(s.mul(5,2)) # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

`register_function()` también se puede utilizar como decorador. El ejemplo de servidor anterior puede registrar funciones a modo de decorador:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    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, using
    # register_function as a decorator. *name* can only be given
    # as a keyword argument.
    @server.register_function(name='add')
    def adder_function(x, y):
        return x + y

    # Register a function under function.__name__.
    @server.register_function
    def mul(x, y):
        return x * y

    server.serve_forever()
```

El siguiente ejemplo incluido en el módulo `Lib/xmlrpc/server.py` muestra un servidor que permite nombres con puntos y registra una función de llamada múltiple.

Advertencia: Habilitar la opción `allow_dotted_names` permite a los intrusos acceder a las variables globales de su módulo y puede permitir que los intrusos ejecuten código arbitrario en su máquina. Utilice este ejemplo únicamente dentro de una red cerrada y segura.

```
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
```

(continué en la próxima página)

(proviene de la página anterior)

```
print('Serving XML-RPC on localhost port 8000')
try:
    server.serve_forever()
except KeyboardInterrupt:
    print("\nKeyboard interrupt received, exiting.")
    sys.exit(0)
```

Esta demostración de `ExampleService` se puede invocar desde la línea de comando:

```
python -m xmlrpc.server
```

El cliente que interactúa con el servidor anterior está incluido en `Lib/xmlrpc/client.py`:

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

Este cliente que interactúa con el servidor XMLRPC de demostración se puede invocar como:

```
python -m xmlrpc.client
```

21.27.2 CGIXMLRPCRequestHandler

La clase `CGIXMLRPCRequestHandler` se puede usar para manejar solicitudes XML-RPC enviadas a scripts Python CGI.

`CGIXMLRPCRequestHandler.register_function(function=None, name=None)`

Registra una función que pueda responder a solicitudes XML-RPC. Si se proporciona *name*, este será el nombre del método asociado con *function*, en otro caso se utilizará `function.__name__`. *name* es una cadena de texto, y puede contener caracteres no permitidos en los identificadores de Python, incluido el carácter de punto.

Este método también se puede utilizar como decorador. Cuando se usa como decorador, *name* solo se puede dar como un argumento de palabra clave para registrar *function* bajo *nombre*. Si no se proporciona *name*, se usará `function.__name__`.

Distinto en la versión 3.7: `register_function()` puede ser usado como decorador.

`CGIXMLRPCRequestHandler.register_instance(instance)`

Registra un objeto que se usa para exponer nombres de métodos que no se han registrado usando `register_function()`. Si la instancia contiene un método `_dispatch()`, se llama con el nombre del método solicitado y los parámetros de la solicitud; el valor de retorno se devuelve al cliente como resultado. Si la instancia no tiene un método `_dispatch()`, se busca un atributo que coincida con el nombre del método solicitado; si el nombre del método contiene puntos, cada componente del nombre del método se busca individualmente, con el efecto con el efecto que produce una búsqueda jerárquica simple. El valor encontrado en esta búsqueda es entonces llamado con los parámetros de la solicitud y el valor de retorno se devuelve al cliente.

`CGIXMLRPCRequestHandler.register_introspection_functions()`

Registra las funciones de introspección `system.listMethods`, `system.methodHelp` y `system.methodSignature`.

`CGIXMLRPCRequestHandler.register_multicall_functions()`

Registra la función de llamada múltiple XML-RPC `system.multicall`.

`CGIXMLRPCRequestHandler.handle_request(request_text=None)`

Maneja una solicitud XML-RPC. Si se proporciona `request_text`, deberían ser los datos POST proporcionados por el servidor HTTP, de lo contrario se utilizará el contenido de `stdin`.

Ejemplo:

```
class MyFuncs:
    def mul(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()
```

21.27.3 Documentando el servidor XMLRPC

Estas clases amplían las clases anteriores para proporcionar documentación HTML en respuesta a solicitudes HTTP GET. Los servidores pueden ser independientes, usando `DocXMLRPCServer`, o integrados en un entorno CGI, usando `DocCGIXMLRPCRequestHandler`.

class `xmlrpc.server.DocXMLRPCServer` (*addr*, *requestHandler=DocXMLRPCRequestHandler*,
logRequests=True, *allow_none=False*, *encoding=None*,
bind_and_activate=True, *use_builtin_types=True*)

Crea una nueva instancia de servidor. Todos los parámetros tienen el mismo significado que para `SimpleXMLRPCServer`; `requestHandler` tiene como valor predeterminado `DocXMLRPCRequestHandler`.

Distinto en la versión 3.3: Se ha añadido el indicador `use_builtin_types`.

class `xmlrpc.server.DocCGIXMLRPCRequestHandler`

Crea una nueva instancia para manejar solicitudes XML-RPC en un entorno CGI.

class `xmlrpc.server.DocXMLRPCRequestHandler`

Crea una nueva instancia de controlador de solicitudes. Este controlador de solicitudes admite solicitudes XML-RPC POST, solicitudes GET de documentación y modifica el registro para que se respete el parámetro `logRequests` del parámetro constructor `DocXMLRPCServer`.

21.27.4 Objetos DocXMLRPCServer

La clase `DocXMLRPCServer` se deriva de `SimpleXMLRPCServer` y proporciona un medio para crear servidores XML-RPC autónomos y autodocumentados. Las solicitudes HTTP POST se manejan como llamadas al método XML-RPC. Las solicitudes HTTP GET se manejan generando documentación HTML al estilo pydoc. Esto permite que un servidor proporcione su propia documentación basada en web.

`DocXMLRPCServer.set_server_title(server_title)`

Establezca el título utilizado en la documentación HTML generada. Este título se utilizará dentro del elemento HTML «title».

`DocXMLRPCServer.set_server_name(server_name)`

Establezca el nombre utilizado en la documentación HTML generada. Este nombre aparecerá en la parte superior de la documentación generada dentro de un elemento «h1».

`DocXMLRPCServer.set_server_documentation(server_documentation)`

Establezca la descripción utilizada en la documentación HTML generada. Esta descripción aparecerá como un párrafo, debajo del nombre del servidor, en la documentación.

21.27.5 DocCGIXMLRPCRequestHandler

La clase `DocCGIXMLRPCRequestHandler` se deriva de `CGIXMLRPCRequestHandler` y proporciona un medio para crear scripts CGI XML-RPC autodocumentados. Las solicitudes HTTP POST se manejan como llamadas al método XML-RPC. Las solicitudes HTTP GET se manejan generando documentación HTML al estilo pydoc. Esto permite que un servidor proporcione su propia documentación basada en web.

`DocCGIXMLRPCRequestHandler.set_server_title(server_title)`

Establezca el título utilizado en la documentación HTML generada. Este título se utilizará dentro del elemento HTML «title».

`DocCGIXMLRPCRequestHandler.set_server_name(server_name)`

Establezca el nombre utilizado en la documentación HTML generada. Este nombre aparecerá en la parte superior de la documentación generada dentro de un elemento «h1».

`DocCGIXMLRPCRequestHandler.set_server_documentation(server_documentation)`

Establezca la descripción utilizada en la documentación HTML generada. Esta descripción aparecerá como un párrafo, debajo del nombre del servidor, en la documentación.

21.28 ipaddress — Biblioteca de manipulación IPv4/IPv6

Código fuente: [Lib/ipaddress.py](#)

`ipaddress` proporciona las capacidades para crear, manipular y operar en direcciones y redes IPv4 e IPv6.

Las funciones y clases de este módulo facilitan el control de varias tareas relacionadas con las direcciones IP, incluido comprobar si dos *hosts* están en la misma subred o no, iterar sobre todos los *hosts* de una subred determinada, comprobar si una cadena de caracteres representa o no una dirección IP válida o una definición de red, etc.

Esta es la referencia completa de la API del módulo: para obtener información general y una introducción, véase `ipaddress-howto`.

Nuevo en la versión 3.3.

21.28.1 Funciones de fábrica de conveniencia

El módulo `ipaddress` proporciona funciones de fábrica para crear convenientemente direcciones IP, redes e interfaces:

`ipaddress.ip_address(address)`

Retorna un objeto `IPv4Address` o `IPv6Address` en función de la dirección IP pasada como argumento. Se pueden proporcionar direcciones IPv4 o IPv6; enteros menores que 2^{32} se considerarán IPv4 de forma predeterminada. Se genera un `ValueError` si `address` no representa una dirección IPv4 o IPv6 válida.

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

`ipaddress.ip_network(address, strict=True)`

Retorna un objeto `IPv4Network` o `IPv6Network` en función de la dirección IP pasada como argumento. `address` es una cadena de caracteres o entero que representa la red IP. Se pueden proporcionar redes IPv4 o IPv6; enteros menores que 2^{32} se considerarán IPv4 de forma predeterminada. `strict` se pasa al constructor

de `IPv4Network` o `IPv6Network`. Se genera un `ValueError` si `address` no representa una dirección IPv4 o IPv6 válida, o si la red tiene los bits *host* establecidos.

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

`ipaddress.ip_interface(address)`

Retorna un objeto `IPv4Interface` o `IPv6Interface` en función de la dirección IP pasada como argumento. `address` es una cadena de caracteres o entero que representa la dirección IP. Se pueden proporcionar direcciones IPv4 o IPv6; enteros menores que 2^{32} se considerarán IPv4 de forma predeterminada. Se genera un `ValueError` si `address` no representa una dirección IPv4 o IPv6 válida.

Una desventaja de estas funciones de conveniencia es que la necesidad de manejar ambos formatos IPv4 e IPv6 significa que los mensajes de error proveen información mínima sobre el error preciso, ya que las funciones no saben si se pretendía usar el formato IPv4 o IPv6. Un reporte de error más detallado se puede obtener llamando directamente a los constructores de clase específicos para la versión apropiada.

21.28.2 Direcciones IP

Objetos de dirección

Los objetos `IPv4Address` y `IPv6Address` comparten muchos atributos comunes. Algunos atributos que son sólo significativos para direcciones IPv6 también están implementados para los objetos `IPv4Address`, para que sea más fácil escribir código que maneje ambas versiones de IP correctamente. Los objetos de dirección son *hashable*, por lo que se pueden utilizar como claves en diccionarios.

class `ipaddress.IPv4Address(address)`

Construye una dirección IPv4. Se genera un `AddressValueError` si `address` no es una dirección IPv4 válida.

Lo siguiente constituye una dirección IPv4 válida:

1. A string in decimal-dot notation, consisting of four decimal integers in the inclusive range 0–255, separated by dots (e.g. `192.168.0.1`). Each integer represents an octet (byte) in the address. Leading zeroes are not tolerated to prevent confusion with octal notation.
2. Un entero que cabe en 32 bits.
3. Un entero empaquetado en un objeto *bytes* de longitud 4 (el octeto más significativo primero).

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xc0\xa8\x00\x01')
IPv4Address('192.168.0.1')
```

Distinto en la versión 3.8: Leading zeros are tolerated, even in ambiguous cases that look like octal notation.

Distinto en la versión 3.8.12: Leading zeros are no longer tolerated and are treated as an error. IPv4 address strings are now parsed as strict as glibc `inet_pton()`.

version

El número de versión apropiado: 4 para IPv4, 6 para IPv6.

max_prefixlen

El número total de bits en la representación de la dirección para esta versión: 32 para IPv4, 128 para IPv6.

El prefijo define el número de bits iniciales en una dirección que son comparados para determinar si una dirección es o no parte de una red.

compressed

Nuevo en la versión 3.4.

Distinto en la versión 3.8.20: Fixed some false positives and false negatives, see *is_private* for details.

is_unspecified

True si la dirección no está especificada. Véase [RFC 5735](#) (para IPv4) o [RFC 2373](#) (para IPv6).

is_reserved

True si la dirección está reservada por la IETF.

is_loopback

True si esta es una dirección de *loopback*. Véase [RFC 3330](#) (para IPv4) o [RFC 2373](#) (para IPv6).

is_link_local

True si la dirección está reservada para uso de enlace-local. Véase [RFC 3927](#).

class `ipaddress.IPv6Address` (*address*)

Construye una dirección IPv6. Se genera un *AddressValueError* si *address* no es una dirección IPv6 válida.

Lo siguiente constituye una dirección IPv6 válida:

1. Una cadena de caracteres que consta de 8 grupos de cuatro dígitos hexadecimales, cada grupo representa 16 bits. Los grupos son separados por dos puntos. Esto describe una notación completa (larga). La cadena también puede ser comprimida (notación corta) de varias maneras. Véase [RFC 4291](#) para más detalles. Por ejemplo, "0000:0000:0000:0000:0000:0abc:0007:0def" puede ser comprimida a "::abc:7:def".
2. Un entero que cabe en 128 bits.
3. Un entero empaquetado en un objeto *bytes* de longitud 16, *big-endian*.

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
```

compressed

La forma abreviada de la representación de la dirección, omitiendo los ceros a la izquierda en los grupos y la secuencia más larga de grupos que consisten completamente de ceros colapsada en un sólo grupo vacío.

Este es también el valor retornado por `str(addr)` para direcciones IPv6.

exploded

La forma larga de la representación de la dirección, incluidos todos los ceros iniciales y los grupos que consisten completamente de ceros.

Para los siguientes atributos, véase la documentación correspondiente de la clase *IPv4Address*:

packed**reverse_pointer****version****max_prefixlen****is_multicast****is_private****is_global****is_unspecified****is_reserved****is_loopback****is_link_local**

Nuevo en la versión 3.4: *is_global*

is_site_local

True si la dirección está reservada para el uso local del sitio. Tenga en cuenta que el espacio de direcciones locales del sitio ha quedado en desuso por [RFC 3879](#). Utilice `is_private` para probar si esta dirección está en el espacio de direcciones locales únicas según lo definido por [RFC 4193](#).

ipv4_mapped

Para las direcciones que parecen ser direcciones IPv4 mapeadas (comenzando con `::FFFF/96`), esta propiedad informará la dirección IPv4 incrustada. Para cualquier otra dirección, esta propiedad será `None`.

sixtofour

Para las direcciones que parecen ser direcciones 6to4 (comenzando con `2002::/16`) según lo definido por [RFC 3056](#), esta propiedad reportará la dirección IPv4 incrustada. Para cualquier otra dirección, esta propiedad será `None`.

teredo

Para las direcciones que parecen ser direcciones *Teredo* (comenzando con `2001::/32`) según lo definido en [RFC 4380](#), esta propiedad reportará el par de direcciones IP (`servidor, cliente`) incrustadas. Para cualquier otra dirección, esta propiedad será `None`.

Conversión a cadenas de caracteres y enteros

Para interoperar con interfaces de red como el módulo `socket`, las direcciones se deben convertir en cadenas de caracteres o enteros. Esto se gestiona usando las funciones `str()` e `int()` incorporadas:

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
 '::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

Operadores

Los objetos de dirección admiten algunos operadores. A menos que se indique lo contrario, los operadores solo se pueden aplicar entre objetos compatibles (es decir, IPv4 con IPv4, IPv6 con IPv6).

Operadores de comparación

Los objetos de dirección pueden compararse con el conjunto usual de operadores de comparación. Algunos ejemplos:

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
```

Operadores aritméticos

Los enteros pueden ser sumados o restados de objetos de dirección. Algunos ejemplos:

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4_
↳address
```

21.28.3 Definiciones de red IP

Los objetos *IPv4Network* y *IPv6Network* proveen un mecanismo para definir e inspeccionar definiciones de redes IP. Una definición de red consiste en una máscara y una dirección de red, y como tal define un rango de direcciones IP que son iguales a la dirección de red cuando se enmascaran (*AND* binario) con la máscara. Por ejemplo, una definición de red con la máscara 255.255.255.0 y la dirección de red 192.168.1.0 consiste de las direcciones IP en el rango inclusivo 192.168.1.0 a 192.168.1.255.

Prefijo, máscara de red y máscara de *host*

Hay varias maneras equivalentes de especificar máscaras de red IP. Un *prefijo* es una notación que denota cuántos bits de orden superior se establecen en la máscara de red. Una *máscara de red* es una dirección IP con cierto número de bits de orden superior establecidos. Por lo tanto, el prefijo /24 es equivalente a la máscara de red 255.255.255.0 en IPv4, o `ffff:ff00::` en IPv6. Además, una *máscara del host* es la inversa lógica de una *máscara de red*, y se utiliza a veces (por ejemplo en las listas de control de acceso de Cisco) para denotar una máscara de red. La máscara de host equivalente a /24 en IPv4 es 0.0.0.255.

Objetos de red

Todos los atributos implementados por los objetos de dirección también se implementan mediante objetos de red. Además, los objetos de red implementan atributos adicionales. Todos estos son comunes entre *IPv4Network* y *IPv6Network*, por lo que para evitar la duplicación solo están documentados para *IPv4Network*. Los objetos de red son *hashable*, por lo que se pueden utilizar como claves en diccionarios.

class `ipaddress.IPv4Network` (*address*, *strict=True*)

Construye una definición de red IPv4. *address* puede ser uno de los siguientes:

1. Una cadena de caracteres de una dirección IP y una máscara opcional, separadas por una barra diagonal (/). La dirección IP es la dirección de red, y la máscara puede ser un número único, lo que significa que es un prefijo, o una representación de una dirección IPv4. Si es la última, la máscara se interpreta como una máscara de red si comienza con un campo distinto de cero, o como una máscara de *host* si comienza con un campo igual a cero, con la única excepción de una máscara con todos ceros que es tratada como una máscara de red. Si no se proporciona una máscara, se considera \32.

Por ejemplo, las siguientes especificaciones de *address* son equivalentes: 192.168.1.0/24, 192.168.1.0/255.255.255.0 y 192.168.1.0/0.0.0.255.

2. Un entero que cabe en 32 bits. Este es equivalente a una red de una sola dirección, siendo *address* la dirección de red y /32 la máscara.
3. Un entero empaquetado en un objeto *bytes* de longitud 4, *big-endian*. La interpretación es similar a un entero *address*.

4. Una tupla con dos elementos con una descripción de dirección y una máscara de red, donde la descripción de dirección es una cadena de caracteres, un entero de 32 bits, un entero empaquetado de 4 bytes, o un objeto `IPv4Address` existente; y una máscara de red es un entero que representa la longitud del prefijo (por ejemplo 24) o una cadena de caracteres que representa la máscara de prefijo (por ejemplo 255.255.255.0).

Se genera un `AddressValueError` si `address` no es una dirección IPv4 válida. Se genera un `NetmaskValueError` si la máscara no es válida para un dirección IPv4.

Si `strict` es `True` y los bits de `host` están establecidos en la dirección proporcionada, se genera `ValueError`. De lo contrario, los bits de `host` se enmascaran para determinar la dirección de red adecuada.

A menos que se indique lo contrario, todos los métodos de red que acepten otros objetos de red/dirección generarán `TypeError` si la versión IP del argumento es incompatible con `self`.

Distinto en la versión 3.5: Se agregó la forma de tupla con dos elementos para el parámetro `address` del constructor.

version

max_prefixlen

Consulta en `IPv4Address` la documentación de atributos correspondiente.

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

Estos atributos son verdaderos para la red en su conjunto si son verdaderos tanto para la dirección de red como para la dirección de difusión.

network_address

La dirección de red para la red. La dirección de red y la longitud del prefijo juntas definen de forma única una red.

broadcast_address

La dirección de difusión para la red. Los paquetes enviados a la dirección de difusión deberían ser recibidos por cada `host` en la red.

hostmask

La máscara de `host`, como un objeto `IPv4Address`.

netmask

La máscara de red, como un objeto `IPv4Address`.

with_prefixlen

compressed

exploded

Una representación en cadena de caracteres de la red, con la máscara en notación de prefijo.

`with_prefixlen` y `compressed` son siempre lo mismo que `str(network).exploded` usa la forma completa de la dirección de red.

with_netmask

Una representación en cadena de caracteres de la red, con la máscara en notación de máscara de red.

with_hostmask

Una representación de cadena de caracteres de la red, con la máscara en notación de máscara de `host`.

num_addresses

El número total de direcciones en la red.

prefixlen

Longitud del prefijo de red, en bits.

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the network address itself and the network broadcast address. For networks with a mask length of 31, the network address and network broadcast address are also included in the result. Networks with a mask of 32 will return a list containing the single host address.

```
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
>>> list(ip_network('192.0.2.1/32').hosts())
[IPv4Address('192.0.2.1')]
```

overlaps(other)

True si esta red está parcial o totalmente contenida en *other* u *other* está totalmente contenida en esta red.

address_exclude(network)

Calcula las definiciones de red que resultan de eliminar *network* de esta red. Retorna un iterador de objetos de red. Se genera *ValueError* si *network* no está completamente contenida en esta red.

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```

subnets(prefixlen_diff=1, new_prefix=None)

Las subredes que se unen para crear la definición de red actual, en función de los valores de argumento. *prefixlen_diff* es la cantidad en la que debería aumentarse nuestra longitud de prefijo. *new_prefix* es el nuevo prefijo deseado de las subredes; debe ser más grande que nuestro prefijo. Se debe establecer uno y solo uno de *prefixlen_diff* y *new_prefix*. Retorna un iterador de objetos de red.

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
```

supernet(prefixlen_diff=1, new_prefix=None)

La superred que contiene esta definición de red, dependiendo de los valores de los argumentos. *prefixlen_diff* es la cantidad en la que debería reducirse la longitud de nuestro prefijo. *new_prefix* es el nuevo prefijo deseado de la superred; debe ser más pequeño que nuestro prefijo. Se debe establecer uno y solo uno de *prefixlen_diff* y *new_prefix*. Retorna un único objeto de red.

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

subnet_of (*other*)Retorna True si esta red es una subred de *other*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> b.subnet_of(a)
True
```

Nuevo en la versión 3.7.

supernet_of (*other*)Retorna True si esta red es una superred de *other*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> a.supernet_of(b)
True
```

Nuevo en la versión 3.7.

compare_networks (*other*)Compara esta red con *other*. En esta comparación solo se consideran las direcciones de red; los bits de *host* no. Retorna -1, 0 o 1.

```
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0
```

Obsoleto desde la versión 3.7: Utiliza el mismo algoritmo de ordenación y comparación que «<», «==», y «>»

class `ipaddress.IPv6Network` (*address*, *strict=True*)Construye una definición de red IPv6. *address* puede ser uno de los siguientes:

1. Una cadena de caracteres que consta de una dirección IP y una longitud de prefijo opcional, separadas por una barra diagonal (/). La dirección IP es la dirección de red y la longitud del prefijo debe ser un solo número, el *prefijo*. Si no se proporciona ninguna longitud de prefijo, se considera que es /128.

Ten en cuenta que actualmente no se admiten máscaras de red expandidas actualmente. Eso significa que 2001:db00::0/24 es un argumento válido mientras que 2001:db00::0/ffff:ff00:: no.

2. Un entero que cabe en 128 bits. Este es equivalente a una red de una sola dirección, siendo *address* la dirección de red y /128 la máscara.
3. Un entero empaquetado en un objeto *bytes* de longitud 16, *big-endian*. La interpretación es similar a un entero *address*.
4. Una tupla con dos elementos con una descripción de dirección y una máscara de red, donde la descripción de dirección es una cadena de caracteres, un entero de 128 bits, un entero empaquetado de 16 bytes, o un objeto *IPv6Address* existente; y una máscara de red es un entero que representa la longitud del prefijo.

Se genera un *AddressValueError* si *address* no es una dirección IPv6 válida. Se genera un *NetmaskValueError* si la máscara no es válida para una dirección IPv6.Si *strict* es True y los bits de *host* están establecidos en la dirección proporcionada, se genera *ValueError*. De lo contrario, los bits de *host* se enmascaran para determinar la dirección de red adecuada.

Distinto en la versión 3.5: Se agregó la forma de tupla con dos elementos para el parámetro *address* del constructor.

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the Subnet-Router anycast address. For networks with a mask length of 127, the Subnet-Router anycast address is also included in the result. Networks with a mask of 128 will return a list containing the single host address.

overlaps(*other*)

address_exclude(*network*)

subnets(*prefixlen_diff*=1, *new_prefix*=None)

supernet(*prefixlen_diff*=1, *new_prefix*=None)

subnet_of(*other*)

supernet_of(*other*)

compare_networks(*other*)

Consulta en [IPv4Network](#) la documentación de atributos correspondiente.

is_site_local

Este atributo es verdadero para la red en su conjunto si es verdadero tanto para la dirección de red como para la dirección de difusión.

Operadores

Los objetos de red admiten algunos operadores. A menos que se indique lo contrario, los operadores solo se pueden aplicar entre objetos compatibles (es decir, IPv4 con IPv4, IPv6 con IPv6).

Operadores lógicos

Los objetos de red pueden compararse con el conjunto usual de operadores lógicos. Los objetos de red son ordenados primero por dirección de red, y después por máscara de red.

Iteración

Los objetos de red se pueden iterar para listar todas las direcciones que pertenecen a la red. Para la iteración, se retornan todos los *hosts*, incluyendo *hosts* inutilizables (para *hosts* utilizables, se usa el método `hosts()`). Un ejemplo:

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

Redes como contenedores de direcciones

Los objetos de red pueden actuar como contenedores de direcciones. Algunos ejemplos:

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

21.28.4 Objetos de interfaz

Los objetos de interfaz son *hashable*, por lo que se pueden utilizar como claves en diccionarios.

class `ipaddress.IPv4Interface` (*address*)

Construye una interfaz IPv4. El significado de *address* es el mismo que en el constructor de *IPv4Network*, excepto que las direcciones de *host* arbitrarias son siempre aceptadas.

IPv4Interface es una subclase de *IPv4Address*, así que hereda todos los atributos de esa clase. Adicionalmente, los siguientes atributos están disponibles:

ip

La dirección (*IPv4Address*) sin información de red.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network

La red (*IPv4Network*) a la que pertenece esta interfaz.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen

Una representación en cadena de caracteres de la interfaz con la máscara en notación de prefijo.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask

Una representación en cadena de caracteres de la interfaz con la red como una máscara de red.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask

Una representación de la interfaz con la red como una máscara de *host*.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class `ipaddress.IPv6Interface` (*address*)

Construye una interfaz IPv6. El significado de *address* es el mismo que en el constructor de *IPv6Network*, excepto que las direcciones de *host* arbitrarias son siempre aceptadas.

IPv6Interface es una subclase de *IPv6Address*, así que hereda todos los atributos de esa clase. Adicionalmente, los siguientes atributos están disponibles:

ip

network

with_prefixlen

with_netmask

with_hostmask

Consulta en *IPv4Interface* la documentación de atributos correspondiente.

Operadores

Los objetos de interfaz admiten algunos operadores. A menos que se indique lo contrario, los operadores solo se pueden aplicar entre objetos compatibles (es decir, IPv4 con IPv4, IPv6 con IPv6).

Operadores lógicos

Los objetos de interfaz pueden compararse con el conjunto usual de operadores lógicos.

Para la comparación de igualdad (`==` and `!=`), tanto la dirección IP como la red deben ser iguales para que los objetos sean iguales. Una interfaz no será igual a ningún objeto de dirección o de red.

Para el ordenamiento (`<`, `>`, etc.) las reglas son diferentes. Los objetos de interfaz y de dirección con la misma versión de IP se pueden comparar, y los objetos de dirección siempre se ordenarán antes que los objetos de interfaz. Dos objetos de interfaz se comparan primero por sus redes y, si son iguales, luego por sus direcciones IP.

21.28.5 Otras funciones a nivel de módulo

El módulo también provee las siguientes funciones a nivel de módulo:

`ipaddress.v4_int_to_packed(address)`

Representa una dirección como 4 bytes empaquetados en orden de red (*big-endian*). *address* es una representación en entero de una dirección IPv4. Se genera un `ValueError` si el entero es negativo o demasiado grande para ser una dirección IPv4.

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

`ipaddress.v6_int_to_packed(address)`

Representa una dirección como 16 bytes empaquetados en orden de red (*big-endian*). *address* es una representación en entero de una dirección IPv6. Se genera un `ValueError` si el entero es negativo o demasiado grande para ser una dirección IPv6.

`ipaddress.summarize_address_range(first, last)`

Retorna un iterador del rango de red resumido dadas la primera y la última dirección IP. *first* es la primera `IPv4Address` o `IPv6Address` en el rango y *last* es la última `IPv4Address` o `IPv6Address` en el rango. Se genera un `TypeError` si *first* o *last* no son direcciones IP o no son de la misma versión. Se genera un `ValueError` si *last* no es mayor que *first* o si la versión de *first* no es 4 o 6.

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...     ipaddress.IPv4Address('192.0.2.0'),
...     ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.
→0.2.130/32')]
```

`ipaddress.collapse_addresses(addresses)`

Retorna un iterador de los objetos colapsados `IPv4Network` o `IPv6Network`. *addresses* es un iterador de objetos `IPv4Network` o `IPv6Network`. Se genera un `TypeError` si *addresses* contiene objetos de versiones distintas.

```
>>> [ipaddr for ipaddr in
...     ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),
...     ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

`ipaddress.get_mixed_type_key(obj)`

Retorna una clave adecuada para ordenar entre redes y direcciones. Los objetos de dirección y red no son ordenables por defecto; son fundamentalmente diferentes, así que la expresión:

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

no tiene sentido. Sin embargo, hay veces donde se desearía hacer que *ipaddress* las ordene de cualquier forma. Si se necesita hacer esto, se puede usar esta función como el argumento *key* de *sorted()*.

obj es un objeto de red o de dirección.

21.28.6 Excepciones personalizadas

Para soportar un reporte de errores más específico desde los constructores de clase, el módulo define las siguientes excepciones:

exception `ipaddress.AddressValueError` (*ValueError*)

Cualquier valor de error relacionado a la dirección.

exception `ipaddress.NetmaskValueError` (*ValueError*)

Cualquier valor de error relacionado a la máscara de red.

Los módulos descritos en este capítulo implementan varios algoritmos o interfaces que son útiles principalmente para aplicaciones multimedia. Su disponibilidad depende de la instalación. He aquí una visión general:

22.1 `audioop` — Manipula datos de audio sin procesar

El módulo `audioop` contiene algunas operaciones útiles sobre fragmentos de sonido. Opera en fragmentos de sonido que consisten en muestras de enteros de 8, 16, 24, o 32 bits, guardados en *objetos parecidos a bytes*. Todos los elementos escalares son enteros, a menos que se especifique lo contrario.

Distinto en la versión 3.4: La compatibilidad para muestras de 24-bit fue añadida. Todas las funciones ahora aceptan cualquier *bytes-like object*. La entrada de cadenas de caracteres ahora resulta en un error inmediato.

Este módulo proporciona compatibilidad con las codificaciones a-LAW, u-LAW e Intel/DVI ADPCM.

Algunas de las operaciones más complicadas sólo toman muestras de 16-bit, si no, el tamaño de la entrada (en bytes) siempre es un parámetro de la operación.

El módulo define las siguientes variables y funciones:

exception `audioop.error`

Esta excepción es lanzada en todos los errores, tal como números desconocidos de bytes por entrada, etc.

`audioop.add(fragment1, fragment2, width)`

Retorna un fragmento que es la adición de dos entradas pasadas como parámetros. *width* es la longitud de la muestra en bytes, o 1, 2, 3, o 4. Ambos fragmentos deben tener la misma longitud. Las muestras son truncadas en caso de desbordamiento.

`audioop.adpcm2lin(adpcmfragment, width, state)`

Decodifica un fragmento codificado con Intel/DVI ADPCM en un fragmento lineal. Véase la descripción de `lin2adpcm()` por detalles sobre la codificación ADPCM. Retorna una tupla (`sample`, `newstate`) donde la entrada tiene la longitud especificada en *width*.

`audioop.alaw2lin(fragment, width)`

Convierte los fragmentos de sonido codificados con a-LAW en fragmentos de sonido linealmente codificados. La codificación a-LAW siempre usa muestras de 8 bits, por lo que *width* hace referencia sólo a la longitud de la entrada del fragmento de salida aquí.

`audioop.avg(fragment, width)`

Retorna el promedio de todas las muestras en el fragmento.

`audioop.avgpp(fragment, width)`

Retorna el promedio del valor de pico a pico de todas las muestras en el fragmento. No se hace ningún filtrado, por lo que la utilidad de esta rutina es cuestionable.

`audioop.bias(fragment, width, bias)`

Retorna un fragmento que es el fragmento original con un *bias* añadido a cada muestra. Las muestras se envuelven en caso de desbordamiento.

`audioop.byteswap(fragment, width)`

Intercambia los bytes («Byteswap») de todas las muestras en un fragmento y retorna el fragmento modificado. Convierte muestras *big-endian* en *little-endian* y viceversa.

Nuevo en la versión 3.4.

`audioop.cross(fragment, width)`

Retorna el número de cruces por 0 en el fragmento pasado como un argumento.

`audioop.findfactor(fragment, reference)`

Retorna un factor *F* tal que `rms(add(fragment, mul(reference, -F)))` sea minimal, i.e., retorna el factor con el cual debes multiplicar la *reference* para hacerlo coincidir tanto como sea posible a *fragment*. Los fragmentos deben contener muestras de 2-byte.

El tiempo tomado por esta rutina es proporcional a `len(fragment)`.

`audioop.findfit(fragment, reference)`

Intenta hacer coincidir *reference* tanto bien como sea posible a un *fragment* (que debe ser el fragmento más largo). Esto es (conceptualmente) hecho al tomar segmentos de *fragment*, usando `findfactor()` para computar la mejor coincidencia, y minimizando el resultado. Los fragmentos deben contener muestras de 2-byte. Retorna una tupla (*offset*, *factor*) donde *offset* (entero) es el *offset* en *fragment* donde la coincidencia más óptima empezó y *factor* es el (número flotante) factor según `findfactor()`.

`audioop.findmax(fragment, length)`

Inspecciona *fragment* por un segmento de longitud *length* muestras (¡no bytes!) con la energía máxima, i.e., retorna *i* por el cual `rms(fragment[i*2:(i+length)*2])` es maximal. Los fragmentos deben contener muestras de 2 bytes.

La rutina tarda proporcionalmente a `len(fragment)`.

`audioop.getsample(fragment, width, index)`

Retorna el valor de la muestra *index* del fragmento.

`audioop.lin2adpcm(fragment, width, state)`

Convierte las muestras en codificaciones Intel/DVI ADPCM de 4 bits. La codificación ADPCM es un esquema de codificación adaptativo a través del cual cada número de 4 bits es la diferencia entre una muestra y la siguiente, dividido por un paso (inconsistente). El algoritmo de Intel/DVI ADPCM ha sido seleccionado para su uso por el *IMA*, por lo que bien puede convertirse en un estándar.

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)`

Convierte las muestras en el fragmento de audio en una codificación a-LAW y los retorna como un objeto de bytes. a-LAW es un formato de codificación de audio a través del cual obtienes un rango dinámico de cerca de 13 bits usando sólo muestras de 8 bits. Es usado por el hardware de audio Sun, entre otros.

`audioop.lin2lin(fragment, width, newwidth)`

Convierte muestras entre formatos de 1, 2, 3, y 4 bytes.

Nota: En algunos formatos de audio, como archivos .WAV, las entradas de 16, 24, y 32 bits tienen signo, pero las entradas de 8 bits no tienen signo. Por lo que cuando se convierta en entradas de 8 bits para estas entradas, también necesitas añadir 128 al resultado:

```
new_frames = audioop.lin2lin(frames, old_width, 1)
new_frames = audioop.bias(new_frames, 1, 128)
```

Lo mismo, al revés, tiene que ser aplicado cuando se convierta muestras de 8 bits en muestras de 16, 24, o 32 bits.

`audioop.lin2ulaw(fragment, width)`

Convierte muestras en el fragmento de audio en codificaciones u-LAW y lo retorna como un objeto de bytes. u-LAW es un formato de codificación de audio a través del cual obtienes un rango dinámico de cerca de 14 bits usando sólo muestras de 8 bits. Es usado por el hardware de audio Sun, entre otros.

`audioop.max(fragment, width)`

Retorna el máximo de los valores *absolutos* de las entradas en un fragmento.

`audioop.maxpp(fragment, width)`

Retorna el valor de pico a pico máximo en el fragmento de sonido.

`audioop.minmax(fragment, width)`

Retorna una tupla que consiste de los valores mínimos y máximos de todas las entradas en el fragmento de sonido.

`audioop.mul(fragment, width, factor)`

Retorna un fragmento que tiene todas las entradas en el fragmento original multiplicado por el valor de punto flotante *factor*. Las muestras son truncadas en caso de desbordamiento.

`audioop.ratecv(fragment, width, nchannels, inrate, outrate, state[, weightA[, weightB]])`

Convierte el ratio de fotogramas del fragmento de entrada.

state es una tupla que contiene el estado del convertidor. El convertidor retorna una tupla (*newfragment*, *newstate*), y *newstate* debe ser pasado a la siguiente llamada de `ratecv()`. La llamada inicial debe pasar *None* como el estado.

Los argumentos *weightA* y *weightB* son parámetros para un filtro digital simple y sus valores por defecto son 1 y 0 respectivamente.

`audioop.reverse(fragment, width)`

Invierte las entradas en un fragmento y retorna el fragmento modificado.

`audioop.rms(fragment, width)`

Retorna la media cuadrática del fragmento, i.e. $\sqrt{\sum(S_i^2)/n}$.

Este es una medida del poder en una señal de audio.

`audioop.tomono(fragment, width, lfactor, rfactor)`

Convierte un fragmento estéreo en una fragmento mono. El canal izquierdo es multiplicado por *lfactor* y el derecho por *rfactor* antes de añadir los dos canales para dar una señal mono.

`audioop.tostereo(fragment, width, lfactor, rfactor)`

Genera un fragmento estéreo de un fragmento mono. Cada par de muestras en el fragmento estéreo son computados de la entrada mono, a través del cual las muestras del canal izquierdo son multiplicadas por *lfactor* y del canal derecho por *rfactor*.

`audioop.ulaw2lin(fragment, width)`

Convierte los fragmentos de sonido en codificaciones u-LAW en fragmentos de sonidos linealmente codificados. Las codificaciones u-LAW siempre usan muestras de 8 bits, por lo que *width* hace referencia a la longitud de la muestra del fragmento de salida aquí.

Note que operaciones tales como `mul()` o `max()` no hacen distinción entre fragmentos mono y estéreo, i.e. todas las muestras son tratadas iguales. Si este es un problema, el fragmento estéreo debe ser dividido en dos fragmentos mono primero y recombinado después. Aquí hay un ejemplo de como hacerlo:

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
```

(continué en la próxima página)

(proviene de la página anterior)

```

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)

```

Si usas el codificador ADPCM para construir paquetes de redes y quieres que tu protocolo no tenga estado (*stateless*) (i.e. para ser capaz de tolerar pérdida de paquetes) no sólo debes transmitir los datos pero también el estado. Note que debes enviar el estado inicial (*initial*) (el que pasas a `lin2adpcm()`) junto con el decodificador, no el estado final (como es retornado por el codificador). Si quieres usar un `struct.Struct` para almacenar el estado en binario puedes codificar el primer elemento (el valor predicho) en 16 bits y el segundo (el índice delta) en 8.

Los codificadores ADPCM nunca se han probado en contra de otros codificadores ADPCM, sólo contra ellos mismos. Bien puede ser que malinterpreté los estándares en cuyo caso ellos no serán interoperables con los estándares respectivos.

Las rutinas `find*()` pueden parecer un poco raras a primera vista. Sirven principalmente para hacer echo de la cancelación. Una manera razonablemente rápida para hacerlo es coger la pieza más energética de la muestra de la salida, localizarla en la muestra de la entrada y substraer la muestra de la salida completa de la muestra de entrada:

```

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)

```

22.2 aifc — Lee y escribe archivos AIFF y AIFFC

Código fuente: `Lib/aifc.py`

Este módulo provee soporte para la lectura y escritura de archivos AIFF y AIFF-C. AIFF son las siglas de Formato de Intercambio de Archivos de Audio (*Audio Interchange File Format*), un formato para almacenar muestras de audio digital en un archivo. AIFF-C es una nueva versión del formato que incluye la habilidad de comprimir los datos de audio.

Los archivos de audio tienen una serie de parámetros que describen los datos de audio. La tasa de muestreo o tasa de fotogramas se refiere a la cantidad de veces por segundo que se toman muestras del sonido. El número de canales indica si el audio es mono, estéreo o cuadrafónico. Cada fotograma está compuesto de una muestra por canal. El tamaño de la muestra es el tamaño en bytes de cada muestra. De esta manera, un fotograma está formado por `nchannels * samplesize` bytes, y un segundo de audio está formado por `nchannels * samplesize * framerate` bytes.

Por ejemplo, el audio de calidad de CD tiene un tamaño de muestreo de 2 bytes (16 bits), usa dos canales (estéreo) y tiene una tasa de fotogramas de 44.100 fotogramas/segundo. Esto da como resultado un tamaño del fotograma de 4 bytes (2*2), y un segundo de audio en esta calidad ocupa 2*2*44.100 bytes (176.400 bytes).

El módulo `aifc` define a la siguiente función:

`aifc.open(file, mode=None)`

Abre un archivo AIFF o AIFF-C y retorna una instancia de objeto con los métodos descriptos más abajo. El argumento `file` puede ser tanto una cadena de caracteres nombrando a un archivo como un *file object*. `mode` debe

ser `'r'` o `'rb'` cuando el archivo sea abierto para lectura, o `'w'` o `'wb'` cuando lo sea para escritura. Si este argumento se omite, se usará `file.mode` si es que existe; en caso contrario se usará `'rb'`. Cuando se use para escribir el objeto archivo deberá ser «buscable» (*seekable*), a menos que se sepa por adelantado cuántas muestras se escribirán en total y use `writeframesraw()` and `setnframes()`. La función `open()` se puede usar dentro de una sentencia `with`. Cuando el bloque `with` se complete, se invocará al método `close()`.

Distinto en la versión 3.4: Se agregó soporte para las sentencias `with`.

Los objetos que retorna `open()` cuando un archivo es abierto para lectura contienen los siguientes métodos:

`aifc.getnchannels()`

Retorna el número de canales de audio (1 para mono, 2 para estéreo).

`aifc.getsampwidth()`

Retorna el tamaño en bytes de cada muestra.

`aifc.getframerate()`

Retorna la tasa de muestreo (cantidad de fotogramas de audio por segundo).

`aifc.getnframes()`

Retorna el número de fotogramas de audio en el archivo.

`aifc.getcomptype()`

Retorna un arreglo de bytes de longitud 4 que describe el tipo de compresión usada en el archivo de audio. Para archivos AIFF, el valor que retorna es `b'NONE'`.

`aifc.getcompname()`

Retorna un arreglo de bytes con una descripción legible para humanos del tipo de compresión usada en el archivo de audio. Para archivos AIFF, el valor que retorna es `b'not compressed'` (no comprimido).

`aifc.getparams()`

Retorna una tupla nombrada `namedtuple()` (`nchannels`, `sampwidth`, `framerate`, `nframes`, `comptype`, `compname`), equivalente a la salida de los métodos `get*()`.

`aifc.getmarkers()`

Retorna una lista de los marcadores en el archivo de audio. Un marcador consiste de una tupla de tres elementos. El primero es el identificador de marca (*mark ID*, un número entero); el segundo es la posición de la marca, en fotogramas, desde el comienzo de los datos (un número entero); el tercero es el nombre de la marca (una cadena de caracteres).

`aifc.getmark(id)`

Retorna una tupla tal como se describe en `getmarkers()` para la marca con el *id* dado.

`aifc.readframes(nframes)`

Lee y retorna los *nframes* fotogramas siguientes del archivo de audio. Los datos los retorna como una cadena de caracteres que contiene, por cada fotograma, las muestras sin comprimir de todos los canales.

`aifc.rewind()`

Rebobina el puntero de lectura. La próxima ejecución de `readframes()` comenzará desde el comienzo del archivo.

`aifc.setpos(pos)`

Busca el número de fotograma especificado.

`aifc.tell()`

Retorna el número de fotograma actual.

`aifc.close()`

Cierra el archivo AIFF. Después de invocar este método, el objeto no puede usarse más.

Cuando un archivo se abre para escritura, los objetos que retorna `open()` poseen todos los métodos mencionados más arriba, excepto `readframes()` y `setpos()`. Adicionalmente se incluyen los métodos abajo descriptos. Los métodos `get*()` sólo pueden ser invocados después de haber invocado su correspondiente método `set*()`. Antes de invocar por primera vez `writeframes()` o `writeframesraw()`, todos los parámetros -excepto el número de fotogramas- deben estar completos.

`aifc.aiff()`

Crea un archivo AIFF. Por defecto se crea un archivo AIFF-C, excepto que el nombre del archivo termine en `'.aiff'`, en cuyo caso se creará un archivo AIFF.

`aifc.aifc()`

Crea un archivo AIFF-C. La acción por defecto es que cree un archivo AIFF-C, excepto que el nombre del archivo termine en `'.aiff'`, en cuyo caso se crea por defecto un archivo AIFF.

`aifc.setnchannels(nchannels)`

Especifica el número de canales en el archivo de audio.

`aifc.setsampwidth(width)`

Especifica el tamaño en bytes de las muestras de audio.

`aifc.setframerate(rate)`

Especifica la frecuencia de muestreo en fotogramas por segundo.

`aifc.setnframes(nframes)`

Especifica el número de fotogramas que se escribirán en el archivo de audio. Si este parámetro no es definido, o si no se lo define correctamente, el archivo necesitará soporte de búsqueda (*seeking*).

`aifc.setcomptype(type, name)`

Especifica el tipo de compresión. Si no es especificada, los datos de audio no serán comprimidos. En los archivos AIFF la compresión no está disponible. El parámetro de nombre *name* deberá ser una descripción del tipo de compresión legible por humanos, en forma de un arreglo de bytes. El parámetro de tipo *type* deberá ser un arreglo de bytes de longitud 4. Actualmente se soportan los siguientes tipos de compresión: `b'NONE'`, `b'ULAW'`, `b'ALAW'`, `b'G722'`.

`aifc.setparams(nchannels, sampwidth, framerate, comptype, compname)`

Establece de una vez todos los parámetros mostrados arriba. El argumento es una tupla compuesta por estos parámetros. Esto quiere decir que es posible usar el resultado de una llamada `getparams()` como argumento para `setparams()`.

`aifc.setmark(id, pos, name)`

Agrega una marca con el identificador *id* dado (mayor a 0) y el nombre *name* dado, en la posición *pos* dada. Este método se puede invocar en cualquier momento antes de `close()`.

`aifc.tell()`

Retorna la posición de escritura actual en el archivo de salida. Es útil en combinación con `setmark()`.

`aifc.writeframes(data)`

Escribe los datos al archivo de salida. Este método sólo se puede invocar una vez establecidos los parámetros del archivo de audio.

Distinto en la versión 3.4: Acepta cualquier *bytes-like object*.

`aifc.writeframesraw(data)`

Funciona igual que `writeframes()`, excepto que el encabezado del archivo de audio no es actualizado.

Distinto en la versión 3.4: Acepta cualquier *bytes-like object*.

`aifc.close()`

Cierra el archivo AIFF. El encabezado del archivo se actualiza para reflejar el tamaño real de los datos de audio. Después de invocar a este método, el objeto no puede usarse más.

22.3 sunau — Lectura y escritura de ficheros Sun AU

Código fuente: [Lib/sunau.py](#)

El módulo `sunau` provee una interfaz conveniente para el formato de sonido Sun AU. Note que este módulo es de interfaz compatible con los módulos `aifc` y `wave`.

Un fichero de audio consiste de un encabezado seguido por la información. Los campos del encabezado son:

Campo	Contenido
palabra mágica	Los cuatro bytes <code>.snd</code> .
tamaño del encabezado	Tamaño del encabezado, incluyendo información, en bytes.
tamaño de la información	Tamaño físico de la información, en bytes.
codificación	Indica cómo las muestras de audio están codificadas.
tasa de muestra	La tasa de muestreo.
# de canales	El número de canales en las muestras.
información	Cadena de caracteres ASCII dando una descripción del fichero de audio (rellenada con bytes nulos).

Aparte del campo de información, todos los campos de encabezado tienen 4 bytes de tamaño. Todos ellos son enteros sin signo codificados en orden de bytes *big-endian*.

El módulo `sunau` define las siguientes funciones:

`sunau.open(file, mode)`

Si *file* es una cadena, abre el fichero por ese nombre, de otra forma lo trata como un objeto similar a un fichero buscable. *mode* puede ser cualquiera de

'r' Modo de sólo lectura.

'w' Modo de sólo escritura.

Note que no acepta ficheros de lectura/escritura.

Un *mode* de 'r' retorna un objeto `AU_read`, mientras un *mode* de 'w' o 'wb' retorna un objeto `AU_write`.

`sunau.openfp(file, mode)`

Un sinónimo para `open()`, mantenido para compatibilidad con versiones anteriores.

Deprecated since version 3.7, will be removed in version 3.9.

El módulo `sunau` define la siguiente excepción:

exception `sunau.Error`

Un error generado cuando algo es imposible por especificaciones de Sun AU o deficiencia de implementación.

El módulo `sunau` define los siguientes ítems de información:

`sunau.AUDIO_FILE_MAGIC`

Un entero por cada fichero Sun AU válido comienza con, almacenada en la forma *big-endian*. Esto es la cadena `.snd` interpretada como un entero.

`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`

Valores del campo de codificación para el encabezado AU que son soportados por este módulo.


```
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
```

Valores adicionales conocidos por el campo de codificación del encabezado AU, pero que no son soportados por este módulo.

22.3.1 Objetos AU_read

Objetos AU_read, como se retornan por `open()` arriba, tienen los siguientes métodos:

`AU_read.close()`

Cierra el flujo, y hace que la instancia sea inutilizable. (Esto es llamado automáticamente en la eliminación.)

`AU_read.getnchannels()`

Retorna el número de canales de audio (1 para mono, 2 para estéreo).

`AU_read.getsampwidth()`

Retorna el ancho de muestra en bytes.

`AU_read.getframerate()`

Retorna la frecuencia de muestreo.

`AU_read.getnframes()`

Retorna el número de cuadros por segundo de audio.

`AU_read.getcomptype()`

Retorna el tipo de compresión. Los tipos de compresión soportados son 'ULAW', 'ALAW' y 'NONE'.

`AU_read.getcompname()`

Versión legible por humanos de `getcomptype()`. Los tipos soportados tienen los nombres respectivos 'CCITT G.711 u-law', 'CCITT G.711 A-law' y 'not compressed'.

`AU_read.getparams()`

Retorna un `namedtuple()` (nchannels, sampwidth, framerate, nframes, comptype, compname), equivalente a la salida de los métodos `get*()`.

`AU_read.readframes(n)`

Lee y retorna al menos *n* fotogramas de audio, como un objeto `bytes`. La información será retornada en formato linear. Si la información original está en formato u-LAW, será convertida.

`AU_read.rewind()`

Rebobina el puntero del fichero al comienzo del flujo de audio.

Los siguientes dos métodos definen un término «position» el cual es compatible entre ellos, y es de otra manera dependiente de la implementación.

`AU_read.setpos(pos)`

Establece el puntero del fichero a la posición especificada. Sólo valores retornados desde `tell()` deberían ser utilizados por *pos*.

`AU_read.tell()`

Retorna la posición actual del puntero de fichero. Note que el valor retornado no tiene nada que ver con la posición actual en el fichero.

Las siguientes dos funciones están definidas por compatibilidad con el `aifc`, y no hace nada interesante.

`AU_read.getmarkers()`

Retorna None.

`AU_read.getmark(id)`

Lanza un error.

22.3.2 Objetos `AU_write`

Objetos `AU_write`, como se retornan por `open()` arriba, tienen los siguientes métodos:

`AU_write.setnchannels(n)`

Establece el número de canales.

`AU_write.setsampwidth(n)`

Establece el ancho de muestra (en bytes.)

Distinto en la versión 3.4: Agregado soporte para muestras de 24 bits.

`AU_write.setframerate(n)`

Establece la velocidad de cuadros por segundo.

`AU_write.setnframes(n)`

Establece el número de cuadros por segundo. Esto puede ser cambiado más adelante, cuando y si más cuadros por segundo son escritos.

`AU_write.setcomptype(type, name)`

Establece el tipo de compresión y descripción. Sólo 'NONE' y 'ULAW' son soportados en salida.

`AU_write.setparams(tuple)`

La *tuple* debería ser (nchannels, sampwidth, framerate, nframes, comptype, compname), con valores válidos para los métodos `set*()`. Establece todos los parámetros.

`AU_write.tell()`

Retorna la posición actual en el fichero, con los mismos descargos de responsabilidad para los métodos `AU_read.tell()` y `AU_read.setpos()`.

`AU_write.writeframesraw(data)`

Escribe cuadros por segundo de audio, sin corregir *nframes*.

Distinto en la versión 3.4: Cualquier *bytes-like object* es aceptado ahora.

`AU_write.writeframes(data)`

Escribe cuadros por segundo de audio y se asegura que *nframes* sea correcto.

Distinto en la versión 3.4: Cualquier *bytes-like object* es aceptado ahora.

`AU_write.close()`

Se asegura que *nframes* sea correcto, y cierra el fichero.

Este método es llamado después de la eliminación.

Note que es inválido establecer cualquier parámetro después de llamar `writeframes()` o `writeframesraw()`.

22.4 wave — Leer y escribir archivos WAV

Código fuente: [Lib/wave.py](#)

El módulo `wave` proporciona una interfaz conveniente para el formato de sonido WAV. No es compatible con la compresión/descompresión, pero sí es compatible con mono/estéreo.

El módulo `wave` define la siguiente función y excepción:

`wave.open(file, mode=None)`

Si *file* es una cadena, abra el archivo con ese nombre, de lo contrario trátelo como un objeto similar a un archivo. *mode* puede ser:

'rb' Modo de solo lectura.

'wb' Modo de solo escritura.

Tenga en cuenta que no permite archivos WAV de lectura/escritura.

Un *mode* de `'rb'` retorna un objeto `Wave_read`, mientras que un *mode* de `'wb'` retorna un objeto `Wave_write`. Si se omite *mode* y se pasa un objeto similar a un archivo como *file*, `file.mode` se usa como el valor predeterminado para *mode*.

Si pasa un objeto similar a un archivo, el objeto *wave* no lo cerrará cuando se llame al método `close()`; es responsabilidad del invocador cerrar el objeto de archivo.

La función `open()` se puede utilizar en una declaración `with`. Cuando el bloque `with` se completa, el método `Wave_read.close()` o el método `Wave_write.close()` es invocado.

Distinto en la versión 3.4: Se agregó soporte para archivos no encontrados.

`wave.openfp(file, mode)`

Un sinónimo de `open()`, es mantenido para la compatibilidad con versiones anteriores.

Deprecated since version 3.7, will be removed in version 3.9.

exception `wave.Error`

Error que se produce cuando algo es imposible porque viola la especificación WAV o alcanza una deficiencia de implementación.

22.4.1 Los objetos *Wave_read*

Los objetos *Wave_read*, tal como lo retorna `open()`, tienen los siguientes métodos:

`Wave_read.close()`

Cierra la secuencia si fue abierta por *wave*, y hace que la instancia sea inutilizable. Esto es llamado automáticamente en la colección de objetos.

`Wave_read.getnchannels()`

Retorna el número de canales de audio (1 para mono, 2 para estéreo).

`Wave_read.getsampwidth()`

Retorna el ancho de la muestra en bytes.

`Wave_read.getframerate()`

Retorna la frecuencia del muestreo.

`Wave_read.getnframes()`

Retorna el número de cuadros del audio.

`Wave_read.getcomptype()`

Retorna el tipo de compresión (`'NONE'` es el único tipo admitido).

`Wave_read.getcompname()`

Versión legible para humanos de `getcomptype()`. Generalmente `'not compressed'` significa `'NONE'`.

`Wave_read.getparams()`

Retorna un `namedtuple()` (`nchannels`, `sampwidth`, `framerate`, `nframes`, `comptype`, `compname`), equivalente a la salida de los métodos `get*()`.

`Wave_read.readframes(n)`

Lee y retorna como máximo *n* cuadros de audio, como un objeto *bytes*.

`Wave_read.rewind()`

Rebobina el puntero del archivo hasta el principio de la secuencia de audio.

Los dos métodos siguientes se definen por compatibilidad con el módulo *aifc*, y no hacen nada interesante.

`Wave_read.getmarkers()`

Retorna `None`.

`Wave_read.getmark(id)`

Lanza un error.

Los dos métodos siguientes definen un término «posición» que es compatible entre ellos y, es dependiente de la implementación.

`Wave_read.setpos(pos)`

Establece el puntero del archivo en la posición especificada.

`Wave_read.tell()`

Retorna la posición actual del puntero del archivo.

22.4.2 Los objetos *Wave_write*

Para las secuencias de salida que se pueden buscar, el encabezado de *wave* se actualizará automáticamente para reflejar el número de cuadros realmente escritos. Para secuencias que no se pueden buscar, el valor *nframes* debe ser preciso cuando se escriben los datos del primer cuadro. Se puede lograr un valor *nframes* preciso llamando a *setnframes()* o *setparams()* con el número de cuadros que se escribirán antes de que se llame a *close()* y luego se usa *writeframesraw()* para escribir los datos del cuadro, o llamando a *writeframes()* con todos los datos del cuadro que se escribirán. En el último caso *writeframes()* calculará el número de cuadros en los datos y establecerá *nframes* como consecuencia antes de escribir los datos del cuadro.

Los objetos *Wave_write*, retornados por *open()*, tienen los siguientes métodos:

Distinto en la versión 3.4: Se agregó soporte para archivos no encontrados.

`Wave_write.close()`

Asegúrese de que *nframes* sea correcto y cierre el archivo si fue abierto por *wave*. Este método es invocado en la colección de objetos. Levantará una excepción si la secuencia de salida no se puede buscar y *nframes* no coinciden con el número de cuadros realmente escritos.

`Wave_write.setnchannels(n)`

Configure el número de canales.

`Wave_write.setsampwidth(n)`

Establezca el ancho de la muestra en *n* bytes.

`Wave_write.setframerate(n)`

Establezca la velocidad del cuadro en *n*.

Distinto en la versión 3.2: Una entrada no-entera para este método se redondea al número entero más cercano.

`Wave_write.setnframes(n)`

Establezca el número de cuadros en *n*. Esto se cambiará más adelante si el número de cuadros realmente escritos es diferente (este intento de actualización levantará un error si no se encuentra la secuencia de salida).

`Wave_write.setcomptype(type, name)`

Establece el tipo de compresión y la descripción. Por el momento, solo se admite el tipo de compresión *NONE*, lo que significa que no hay compresión.

`Wave_write.setparams(tuple)`

La *tupla* debe ser (*nchannels*, *sampwidth*, *framerate*, *nframes*, *comptype*, *compname*), con valores válidos para los métodos *set *()*. Establece todos los parámetros.

`Wave_write.tell()`

Retorna la posición actual en el archivo, con el mismo descargo para los métodos *Wave_read.tell()* y *Wave_read.setpos()*.

`Wave_write.writeframesraw(data)`

Escribe cuadros de audio, sin corregir *nframes*.

Distinto en la versión 3.4: Todo *bytes-like object* ahora es aceptado.

`Wave_write.writeframes(data)`

Escribe cuadros de audio y se asegura de que *nframes* sea correcto. Levantará un error si no se puede encontrar la secuencia de salida y si el número total de cuadros que se han escrito después de que se haya escrito *data* no coincide con el valor establecido previamente para *nframes*.

Distinto en la versión 3.4: Todo *bytes-like object* ahora es aceptado.

Tenga en cuenta que no es válido establecer ningún parámetro después de invocar a `writeframes()` o `writeframesraw()`, y cualquier intento de hacerlo levantará `wave.Error`.

22.5 chunk — Lee los datos de los trozos de IFF

Código fuente: [Lib/chunk.py](#)

Este módulo proporciona una interfaz para leer archivos que usan trozos de EA IFF 85.¹ Este formato se utiliza al menos en el formato *Audio Interchange File Format (AIFF/AIFF-C)* y en el formato *Real Media File Format (RMFF)*. El formato de archivo de audio WAVE está estrechamente relacionado y también puede ser leído usando este módulo.

Un trozo tiene la siguiente estructura:

Desplazamiento	Longitud	Contenido
0	4	ID del trozo
4	4	El tamaño del trozo en el orden de bytes big-endian, sin incluir el encabezado
8	<i>n</i>	Bytes de datos, donde <i>n</i> es el tamaño dado en el campo anterior
8 + <i>n</i>	0 o 1	Se necesita un byte de relleno si <i>n</i> es impar y se utiliza la alineación de trozos

La ID es una cadena de 4 bytes que identifica el tipo de trozo.

El campo de tamaño (un valor de 32 bits, codificado utilizando el orden de bytes big-endian) da el tamaño de los datos de los trozos, sin incluir el encabezamiento de 8 bytes.

Normalmente un archivo de tipo IFF consiste en uno o más trozos. El uso propuesto de la clase `Chunk` definido aquí es declarar una instancia al principio de cada trozo y leer de la instancia hasta que llegue al final, después de lo cual se puede declarar una nueva instancia. Al final del archivo, la creación de una nueva instancia fallará con una excepción `EOFError`.

class `chunk.Chunk` (*file*, *align=True*, *bigendian=True*, *inclheader=False*)

Clase que representa un trozo. Se espera que el argumento *file* sea un objeto parecido a un archivo. Una instancia de esta clase está específicamente permitida. El único método que se necesita es `read()`. Si los métodos `seek()` y `tell()` están presentes y no plantean una excepción, también se utilizan. Si estos métodos están presentes y hacen una excepción, se espera que no hayan alterado el objeto. Si el argumento opcional *align* es cierto, se supone que los trozos están alineados en los límites de 2 bytes. Si *align* es falso, se asume que no hay alineación. El valor por defecto es `true`. Si el argumento opcional *bigendian* es falso, se asume que el tamaño de los trozos está en orden little-endian. Esto es necesario para los archivos de audio WAVE. El valor por defecto es `true`. Si el argumento opcional *inclheader* es `true`, el tamaño dado en la cabecera del trozo incluye el tamaño de la cabecera. El valor por defecto es `false`.

Un objeto `Chunk` soporta los siguientes métodos:

getname()

Retorna el nombre (ID) del trozo. Estos son los primeros 4 bytes del trozo.

getsize()

Retorna el tamaño del trozo.

close()

Cierra y salta al final del trozo. Esto no cierra el archivo subyacente.

El resto de los métodos se levantarán `OsError` si se llama después de que el método `close()` haya sido llamado. Antes de *Python* 3.3, solían levantar `IOError`, ahora un alias de `OsError`.

isatty()

Retorna `False`.

¹ «EA IFF 85» *Standard for Interchange Format Files*, Jerry Morrison, Electronic Arts, enero 1985.

seek (*pos*, *whence=0*)

Establece la posición actual del trozo. El argumento *whence* es opcional y por defecto es 0 (posicionamiento absoluto del archivo); otros valores son 1 (búsqueda relativa a la posición actual) y 2 (búsqueda relativa al final del archivo). No hay ningún valor de retorno. Si el archivo subyacente no permite la búsqueda, sólo se permiten las búsquedas hacia adelante.

tell ()

Retorna la posición actual en el trozo.

read (*size=-1*)

Leer como máximo *size* bytes del trozo (menos si la lectura llega al final del trozo antes de obtener *size* bytes). Si el argumento *size* es negativo u omitido, lee todos los datos hasta el final del trozo. Un objeto de bytes vacío se retorna cuando el final del trozo se encuentra inmediatamente.

skip ()

Salta al final del trozo. Todas las llamadas posteriores a `read()` para el trozo retorna b''. Si no estás interesado en el contenido del trozo, este método debe ser llamado para que el archivo apunte al comienzo del siguiente trozo.

22.6 colorsys — Conversiones entre sistemas de color

Código fuente: `Lib/colors.py`

El módulo `colorsys` define conversiones bidireccionales de valores de color entre colores expresados en el espacio de color RGB (sigla en inglés de *Red Green Blue*, en español Rojo Verde Azul) utilizado en monitores de ordenador y otros tres sistemas de coordenadas: YIQ, HLS (sigla en inglés de *Hue Lightness Saturation*, en español Matiz Luminosidad Saturación) y HSV (sigla en inglés de *Hue Saturation Value*, en español Matiz Saturación Valor). Las coordenadas en todos estos espacios de color son números de punto flotante. En el espacio YIQ, la coordenada Y está entre 0 y 1, pero las coordenadas I y Q pueden ser positivas o negativas. En todos los demás espacios, las coordenadas están todas entre 0 y 1.

Ver también:

Puede encontrar más información sobre los espacios de color en <http://poynton.ca/ColorFAQ.html> y <https://www.cambridgeincolour.com/tutorials/color-spaces.htm>.

El módulo `colorsys` define las siguientes funciones:

`colorsys.rgb_to_yiq(r, g, b)`

Convierte el color de las coordenadas RGB en coordenadas YIQ.

`colorsys.yiq_to_rgb(y, i, q)`

Convierte el color de las coordenadas YIQ en coordenadas RGB.

`colorsys.rgb_to_hls(r, g, b)`

Convierte el color de las coordenadas RGB en coordenadas HLS.

`colorsys.hls_to_rgb(h, l, s)`

Convierte el color de las coordenadas HLS en coordenadas RGB.

`colorsys.rgb_to_hsv(r, g, b)`

Convierte el color de las coordenadas RGB en coordenadas HSV.

`colorsys.hsv_to_rgb(h, s, v)`

Convierte el color de las coordenadas HSV en coordenadas RGB.

Ejemplo:

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

22.7 :mod:"imghdr" — Determinar el tipo de imagen

Código fuente: :source:"Lib/imghdr.py"

El módulo `imghdr` determina el tipo de imagen contenida en un archivo o secuencia de bytes.

El módulo `imghdr` define la siguiente función:

`imghdr.what(filename, h=None)`

Comprueba los datos de imagen contenidos en el archivo mencionado en *filename* y retorna una cadena de caracteres que describe el tipo de imagen. Si se proporciona el argumento opcional *h*, se ignora *filename* y se supone que *h* contiene la secuencia de bytes que se va a analizar.

Distinto en la versión 3.6: Acepta un *path-like object*.

Se reconocen los siguientes tipos de imagen, enumerados a continuación con el valor devuelto de `what()`:

Valor	Formato de imagen
'rgb'	Archivos SGI ImgLib
'gif'	Archivos GIF 87a y 89a
'pbm'	Archivos Portable Bitmap
'pgm'	Archivos Portable Graymap
'ppm'	Archivos Portable Pixmap
'tiff'	Archivos TIFF
'rast'	Archivos Sun Raster
'xbm'	Archivos X Bitmap
'jpeg'	Datos JPEG en formatos JFIF o Exif
'bmp'	Archivos BMP
'png'	Portable Network Graphics
'webp'	Archivos WebP
'exr'	Archivos OpenEXR

Nuevo en la versión 3.5: Se añadieron los formatos *exr* y *webp*.

Puede ampliar la lista de tipos de archivo que `imghdr` puede reconocer agregándolos a esta variable:

`imghdr.tests`

Una lista de funciones que realizan las comprobaciones individuales. Cada función toma dos argumentos: la secuencia de bytes y un objeto abierto de tipo archivo. Cuando `what()` es llamada con una secuencia de bytes, el objeto de tipo archivo será `None`.

La función de comprobación debería retornar una cadena de caracteres que describa el tipo de imagen si la comprobación se realizó correctamente o `""Ninguno""` si ha fallado.

Ejemplo:

```
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```

22.8 sndhdr — Determinar el tipo de archivo de sonido

Código fuente: [Lib/sndhdr.py](#)

El `sndhdr` proporciona funciones de utilidad que intentan determinar el tipo de datos de sonido que hay en un archivo. Cuando estas funciones son capaces de determinar qué tipo de datos sonoros se almacenan en un archivo, retornan un `namedtuple()`, que contiene cinco atributos: (`filetype`, `framerate`, `nchannels`, `nframes`, `sampwidth`). El valor de `type` indica el tipo de datos y será una de las cadenas siguientes cadenas: 'aifc', 'aiff', 'au', 'hcom', 'sndr', 'sndt', 'voc', 'wav', '8svx', 'sb', 'ub', o 'ul'. El `sampling_rate` será el valor actual o 0 si es desconocido o difícil de decodificar. De forma similar, `channels` será el número de canales o 0 si no se puede determinar o si el valor es difícil de decodificar. El valor de `frames` será el número de fotogramas o -1. El último elemento de la tupla, `bits_per_sample`, será el tamaño de la muestra en bits, 'A' para A-LAW o 'U' para u-LAW.

`sndhdr.what(filename)`

Determina el tipo de datos de sonido almacenados en el archivo `filename` usando `whathdr()`. Si se tiene éxito, retorna una `namedtuple` como se describe arriba, de lo contrario retorna `None`.

Distinto en la versión 3.5: El resultado cambió de una tupla a una `namedtuple`.

`sndhdr.whathdr(filename)`

Determina el tipo de dato de sonido almacenado en un archivo basado en el encabezado del archivo. El nombre del archivo viene dado por `filename`. Esta función retorna una `namedtuple` como se ha descrito anteriormente en caso de éxito o `None`.

Distinto en la versión 3.5: El resultado cambió de una tupla a una `namedtuple`.

22.9 ossaudiodev — Acceso a dispositivos de audio compatibles con OSS

Este módulo le permite acceder a la interfaz de audio OSS (Open Sound System). OSS está disponible para una amplia gama de Unices comerciales y de código abierto, y es la interfaz de audio estándar para Linux y versiones recientes de FreeBSD.

Distinto en la versión 3.3: Las operaciones en este módulo ahora generan `OSError` donde `IOError` es generado.

Ver también:

Open Sound System Programmer's Guide la documentación oficial de la API de OSS C

El módulo define una gran cantidad de constantes suministradas por el controlador de dispositivo OSS; consulte `<sys/soundcard.h>` en Linux o FreeBSD para obtener un listado.

`ossaudiodev` define las siguientes variables y funciones:

exception `ossaudiodev.OSSAudioError`

Esta excepción se genera en ciertos errores. El argumento es una cadena que describe qué salió mal.

(Si `ossaudiodev` recibe un error de una llamada al sistema como `open()`, `write()`, o `ioctl()`, genera `OSError`. Los errores detectados directamente por `ossaudiodev` dan como resultado `OSSAudioError`.)

(Para compatibilidad con versiones anteriores, la clase de excepción también está disponible como `ossaudiodev.error`.)

`ossaudiodev.open(mode)`

`ossaudiodev.open(device, mode)`

Abra un dispositivo de audio y retorne un objeto de dispositivo de audio OSS. Este objeto admite muchos métodos similares a archivos, como `read()`, `write()`, and `fileno()` (aunque existen diferencias sutiles entre la semántica de lectura/escritura convencional de Unix y la de audio OSS dispositivos). También es compatible con varios métodos específicos de audio; vea a continuación la lista completa de métodos.

device es el nombre de archivo del dispositivo de audio que se utilizará. Si no se especifica, este módulo primero busca en la variable de entorno `AUDIODEV` para un dispositivo a usar. Si no se encuentra, vuelve a `/dev/dsp`.

mode es uno de `'r'` para acceso de solo lectura (grabación), `'w'` para acceso de solo escritura (reproducción) y `'rw'` para ambos. Dado que muchas tarjetas de sonido solo permiten que un proceso tenga la grabadora o el reproductor abiertos a la vez, es una buena idea abrir el dispositivo solo para la actividad necesaria. Además, algunas tarjetas de sonido son semidúplex: se pueden abrir para leer o escribir, pero no ambas a la vez.

Tenga en cuenta la sintaxis de llamada inusual: el *first* argumento es opcional y el segundo es obligatorio. Este es un artefacto histórico para la compatibilidad con el módulo anterior `linuxaudiodev` que `ossaudiodev` reemplaza.

`ossaudiodev.openmixer([device])`

Abra un dispositivo mezclador y retorne un objeto de dispositivo mezclador OSS. *device* es el nombre de archivo del dispositivo mezclador que se utilizará. Si no se especifica, este módulo primero busca en la variable de entorno `MIXERDEV` para usar un dispositivo. Si no se encuentra, vuelve a `/dev/mixer`.

22.9.1 Objetos de dispositivo de audio

Antes de poder escribir o leer desde un dispositivo de audio, debe llamar a tres métodos en el orden correcto:

1. `setfmt()` para configurar el formato de salida
2. `channels()` para configurar el número de canales
3. `speed()` para configurar la frecuencia de muestreo

Alternativamente, puede usar el método `setparameters()` para configurar los tres parámetros de audio a la vez. Esto es más conveniente, pero puede que no sea tan flexible en todos los casos.

Los objetos de dispositivo de audio retornados por `open()` definen los siguientes métodos y atributos (de solo lectura):

`oss_audio_device.close()`

Cierre explícitamente el dispositivo de audio. Cuando haya terminado de escribir o leer desde un dispositivo de audio, debe cerrarlo explícitamente. Un dispositivo cerrado no se puede volver a utilizar.

`oss_audio_device.fileno()`

Retorna el descriptor de archivo asociado con el dispositivo.

`oss_audio_device.read(size)`

Leer *size* bytes de la entrada de audio y retornarlos como una cadena de Python. A diferencia de la mayoría de los controladores de dispositivos Unix, los dispositivos de audio OSS en modo de bloqueo (el predeterminado) bloquearán `read()` hasta que esté disponible toda la cantidad de datos solicitada.

`oss_audio_device.write(data)`

Escriba un *bytes-like object* *data* en el dispositivo de audio y retorne el número de bytes escritos. Si el dispositivo de audio está en modo de bloqueo (el predeterminado), todos los datos siempre se escriben (nuevamente, esto es diferente de la semántica habitual del dispositivo Unix). Si el dispositivo está en modo sin bloqueo, es posible que algunos datos no se escriban—see `writeall()`.

Distinto en la versión 3.5: Ahora se aceptan objetos con permisos de escritura con formato *bytes-like object*.

`oss_audio_device.writeall(data)`

Escriba un *bytes-like object* *data* en el dispositivo de audio: espera hasta que el dispositivo de audio sea capaz de aceptar datos, escribe tantos datos como acepte y repite hasta que *data* se hayan escrito por completo. Si el dispositivo está en modo de bloqueo (el predeterminado), esto tiene el mismo efecto que `write()`; `writeall()` solo es útil en el modo sin bloqueo. No tiene valor de retorno, ya que la cantidad de datos escritos siempre es igual a la cantidad de datos suministrados.

Distinto en la versión 3.5: Ahora se aceptan objetos con permisos de escritura con formato *bytes-like object*.

Distinto en la versión 3.2: Los objetos de dispositivo de audio también admiten el protocolo de gestión de contexto, es decir, se pueden utilizar en una declaración `with`.

Los siguientes métodos se asignan exactamente a una llamada al sistema `ioctl()`. La correspondencia es obvia: por ejemplo, `setfmt()` corresponde al `SNDCTL_DSP_SETFMT` `ioctl`, y `sync()` a `SNDCTL_DSP_SYNC` (esto puede ser útil cuando se consulta la documentación de OSS). Si el subyacente `ioctl()` falla, todos generan `OSError`.

`oss_audio_device.nonblock()`

Ponga el dispositivo en modo sin bloqueo. Una vez en el modo sin bloqueo, no hay forma de volverlo al modo de bloqueo.

`oss_audio_device.getfmts()`

Retorna una máscara de bits de los formatos de salida de audio admitidos por la tarjeta de sonido. Algunos de los formatos admitidos por OSS son:

Formato	Descripción
AFMT_MU_LAW	una codificación logarítmica (utilizada por los archivos Sun <code>.au</code> y <code>/dev/audio</code>)
AFMT_A_LAW	una codificación logarítmica
AFMT_IMA_ADPCM	un formato comprimido 4:1 definido por la Interactive Multimedia Association
AFMT_U8	Audio de 8 bits sin firmar
AFMT_S16_LE	Orden de bytes little-endian firmado, audio de 16 bits (como lo utilizan los procesadores Intel)
AFMT_S16_BE	Orden de bytes big-endian firmado, audio de 16 bits (como lo utilizan 68k, PowerPC, Sparc)
AFMT_S8	Firmado, audio de 8 bits
AFMT_U16_LE	Audio little-endian de 16 bits sin firmar
AFMT_U16_BE	Audio big-endian de 16 bits sin firmar

Consulte la documentación de OSS para obtener una lista completa de los formatos de audio y tenga en cuenta que la mayoría de los dispositivos solo admiten un subconjunto de estos formatos. Algunos dispositivos antiguos solo admiten AFMT_U8; el formato más común utilizado hoy en día es AFMT_S16_LE.

`oss_audio_device.setfmt(format)`

Intente establecer el formato de audio actual en *format*—consulte `getfmts()` para obtener una lista. Retorna el formato de audio en el que se configuró el dispositivo, que puede no ser el formato solicitado. También se puede utilizar para retornar el formato de audio actual. Haga esto pasando un «formato de audio» de AFMT_QUERY.

`oss_audio_device.channels(nchannels)`

Establezca el número de canales de salida en *nchannels*. Un valor de 1 indica sonido monofónico, 2 estereofónico. Algunos dispositivos pueden tener más de 2 canales y algunos dispositivos de gama alta pueden no admitir mono. Retorna el número de canales en los que se configuró el dispositivo.

`oss_audio_device.speed(samplerate)`

Intente establecer la frecuencia de muestreo de audio en *samplerate* muestras por segundo. Retorna la tasa realmente establecida. La mayoría de los dispositivos de sonido no admiten frecuencias de muestreo arbitrarias. Las tarifas comunes son:

Velocidad	Descripción
8000	tasa predeterminada para <code>/dev/audio</code>
11025	grabación de voz
22050	
44100	Audio con calidad de CD (a 16 bits/muestra y 2 canales)
96000	Audio con calidad de DVD (a 24 bits/muestra)

`oss_audio_device.sync()`

Espere hasta que el dispositivo de sonido haya reproducido todos los bytes de su búfer. (Esto sucede implícitamente al finalizar la reproducción).

tamente cuando el dispositivo está cerrado). La documentación de OSS recomienda cerrar y volver a abrir el dispositivo en lugar de usar `sync()`.

`oss_audio_device.reset()`

Deje de reproducir o grabar inmediatamente y retorne el dispositivo a un estado en el que pueda aceptar comandos. La documentación de OSS recomienda cerrar y volver a abrir el dispositivo después de llamar `reset()`.

`oss_audio_device.post()`

Dígale al controlador que es probable que haya una pausa en la salida, lo que hace posible que el dispositivo maneje la pausa de manera más inteligente. Puede usar esto después de reproducir un efecto de sonido puntual, antes de esperar la entrada del usuario o antes de realizar E/S de disco.

Los siguientes métodos de conveniencia combinan varios `ioctl`, o un `ioctl` y algunos cálculos simples.

`oss_audio_device.setparameters(format, nchannels, samplerate[, strict=False])`

Configure los parámetros clave de muestreo de audio (formato de muestra, número de canales y frecuencia de muestreo) en una llamada de método. `format`, `nchannels`, y `samplerate` deben ser como se especifica en los métodos `setfmt()`, `channels()`, y `speed()`. Si `strict` es verdadero, `setparameters()` comprueba si cada parámetro se estableció realmente en el valor solicitado, y genera `OSSAudioError` si no es así. Retorna una tupla (`format`, `nchannels`, `samplerate`) que indica los valores de los parámetros que realmente estableció el controlador del dispositivo (es decir, los mismos que los valores retornados de `setfmt()`, `channels()`, y `speed()`).

Por ejemplo,

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

es equivalente a

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(rate)
```

`oss_audio_device.bufsize()`

Retorna el tamaño del búfer de hardware, en muestras.

`oss_audio_device.obufcount()`

Retorna el número de muestras que están en el búfer de hardware que aún no se han reproducido.

`oss_audio_device.obuffree()`

Retorna el número de muestras que se podrían poner en cola en el búfer de hardware para reproducirse sin bloqueos.

Los objetos de dispositivo de audio también admiten varios atributos de solo lectura:

`oss_audio_device.closed`

Booleano que indica si el dispositivo se ha cerrado.

`oss_audio_device.name`

Cadena que contiene el nombre del archivo del dispositivo.

`oss_audio_device.mode`

El modo de E/S para el archivo, ya sea "r", "rw", o "w".

22.9.2 Objetos del dispositivo mezclador

El objeto del mezclador proporciona dos métodos similares a archivos:

`oss_mixer_device.close()`

Este método cierra el archivo del dispositivo mezclador abierto. Cualquier otro intento de usar el mezclador después de que este archivo esté cerrado generará un *OSError*.

`oss_mixer_device.fileno()`

Retorna el número de identificador de archivo del archivo del dispositivo mezclador abierto.

Distinto en la versión 3.2: Los objetos Mixer también admiten el protocolo de gestión de contexto.

Los métodos restantes son específicos para la mezcla de audio:

`oss_mixer_device.controls()`

Este método retorna una máscara de bits que especifica los controles del mezclador disponibles («Control» es un «canal» mezclable específico, como `SOUND_MIXER_PCM` o `SOUND_MIXER_SYNTH`). Esta máscara de bits indica un subconjunto de todos los controles de mezclador disponibles — las constantes `SOUND_MIXER_*` definidas a nivel de módulo. Para determinar si, por ejemplo, el objeto mezclador actual admite un mezclador PCM, utilice el siguiente código Python:

```
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

Para la mayoría de los propósitos, los controles `SOUND_MIXER_VOLUME` (volumen maestro) y `SOUND_MIXER_PCM` deberían ser suficientes — pero el código que usa el mezclador debe ser flexible cuando se trata de elegir los controles del mezclador. En el Ultrasonido Gravis, por ejemplo, `SOUND_MIXER_VOLUME` no existe.

`oss_mixer_device.stereocontrols()`

Retorna una máscara de bits que indica los controles del mezclador estéreo. Si se establece un bit, el control correspondiente es estéreo; si no está configurado, el control es monofónico o no es compatible con el mezclador (úselo en combinación con `controls()` para determinar cuál).

Consulte el ejemplo de código de la función `controls()` para ver un ejemplo de cómo obtener datos de una máscara de bits.

`oss_mixer_device.recontrols()`

Retorna una máscara de bits que especifica los controles del mezclador que se pueden usar para grabar. Consulte el ejemplo de código de `controls()` para ver un ejemplo de lectura desde una máscara de bits.

`oss_mixer_device.get(control)`

Retorna el volumen de un control de mezcla determinado. El volumen retornado es una tupla de 2 (`left_volume, right_volume`). Los volúmenes se especifican como números del 0 (silencioso) al 100 (volumen completo). Si el control es monofónico, todavía se retorna una tupla de 2, pero ambos volúmenes son iguales.

Genera *OSSAudioError* si se especifica un control no válido, o *OSError* si se especifica un control no admitido.

`oss_mixer_device.set(control, (left, right))`

Establece el volumen para un control de mezclador dado en (`left, right`). `left` y `right` deben ser enteros y estar entre 0 (silencio) y 100 (volumen completo). En caso de éxito, el nuevo volumen se retorna como 2 tuplas. Tenga en cuenta que puede que no sea exactamente el mismo que el volumen especificado, debido a la resolución limitada de algunos mezcladores de tarjetas de sonido.

Aumenta *OSSAudioError* si se especificó un control de mezcla no válido, o si los volúmenes especificados estaban fuera de rango.

`oss_mixer_device.get_recsrc()`

Este método retorna una máscara de bits que indica qué control(es) se están utilizando actualmente como fuente de grabación.

`oss_mixer_device.set_recsrc` (*bitmask*)

Llame a esta función para especificar una fuente de grabación. Retorna una máscara de bits que indica la nueva fuente de grabación (o fuentes) si tiene éxito; plantea *OSError* si se especificó una fuente no válida. Para configurar la fuente de grabación actual a la entrada del micrófono:

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```

Los módulos descritos en este capítulo te ayudan a escribir software que es independiente del idioma y lugar al proporcionar mecanismos para seleccionar un idioma a ser usado en mensajes de programas o adaptar la salida para que coincida con las convenciones locales.

La lista de módulos descritos en este capítulo es:

23.1 `gettext` — Servicios de internacionalización multilingües

Código fuente: [Lib/gettext.py](#)

El módulo `gettext` proporciona servicios de internacionalización (I18N) y localización (L10N) para sus módulos y aplicaciones Python. Admite tanto la API del catálogo de mensajes GNU **gettext** como una API basada en clases de nivel superior que puede ser más apropiada para los archivos Python. La interfaz que se describe a continuación le permite escribir sus mensajes de módulo y aplicación en un idioma natural, y proporcionar un catálogo de mensajes traducidos para ejecutar en diferentes idiomas naturales.

También se dan algunas sugerencias para localizar sus módulos y aplicaciones Python.

23.1.1 GNU API `gettext`

El módulo `gettext` define la siguiente API, que es muy similar al programa GNU **gettext** API. Si usa esta API, afectará la traducción de toda su aplicación a nivel mundial. A menudo, esto es lo que desea si su aplicación es monolingüe, y la elección del idioma depende de la configuración regional de su usuario. Si está localizando un módulo de Python, o si su aplicación necesita cambiar idiomas sobre la marcha, probablemente desee utilizar la API basada en clases.

`gettext.bindtextdomain` (*domain*, *localedir*=None)

Enlaza (*bind*) el *domain* al directorio de configuración regional *localedir*. Más concretamente, `gettext` buscará archivos binarios `.mo` para el dominio dado usando la ruta (en Unix): `localedir/language/LC_MESSAGES / domain.mo`, donde se busca *language* en las variables de entorno `LANGUAGE`, `LC_ALL`, `LC_MESSAGES` y `LANG` respectivamente.

Si *localedir* se omite o es *None*, se retorna el enlace actual para *domain*.¹

`gettext.bind_textdomain_codeset (domain, codeset=None)`

Enlaza (*bind*) el *domain* al *codeset*, cambiando la codificación de las cadenas de bytes retornadas por las funciones `gettext()`, `ldgettext()`, `lgettext()` y `ldngettext()`. Si se omite *codeset*, se retorna el enlace (*binding*) actual.

Deprecated since version 3.8, will be removed in version 3.10.

`gettext.textdomain (domain=None)`

Cambia o consulta el dominio global actual. Si *domain* es *None*, se retorna el dominio global actual; de lo contrario, el dominio global se establece en *domain*, que se retorna.

`gettext.gettext (message)`

Retorna la traducción localizada de *message*, en función del dominio global actual, el idioma y el directorio de configuración regional. Esta función generalmente tiene un alias como `_()` en el espacio de nombres local (ver ejemplos a continuación).

`gettext.dgettext (domain, message)`

Como `gettext()`, pero busque el mensaje en el *domain* especificado.

`gettext.ngettext (singular, plural, n)`

Como `gettext()`, pero considere formas plurales. Si se encuentra una traducción, aplique la fórmula plural a *n* y retorna el mensaje resultante (algunos idiomas tienen más de dos formas plurales). Si no se encuentra ninguna traducción, retorna *singular* if *n* es 1; retorna *plural* de lo contrario.

La fórmula Plural se toma del encabezado del catálogo. Es una expresión C o Python que tiene una variable libre *n*; la expresión se evalúa como el índice del plural en el catálogo. Consulte [la documentación de GNU gettext](#) para conocer la sintaxis precisa que se utilizará en archivos `.po` y las fórmulas para un variedad de idiomas.

`gettext.dngettext (domain, singular, plural, n)`

Como `ngettext()`, pero busque el mensaje en el *domain* especificado.

`gettext.pgettext (context, message)`

`gettext.dpgettext (domain, context, message)`

`gettext.npgettext (context, singular, plural, n)`

`gettext.dnpgettext (domain, context, singular, plural, n)`

Similar a las funciones correspondientes sin la *p* en el prefijo (es decir, `gettext()`, `dgettext()`, `ngettext()`, `dngettext()`), pero la traducción está restringida al mensaje dado *context*.

Nuevo en la versión 3.8.

`gettext.lgettext (message)`

`gettext.ldgettext (domain, message)`

`gettext.lngettext (singular, plural, n)`

`gettext.ldngettext (domain, singular, plural, n)`

Equivalente a las funciones correspondientes sin el prefijo *l* (`gettext()`, `dgettext()`, `ngettext()` y `dngettext()`), pero la traducción se retorna como una cadena de bytes codificada en la codificación del sistema preferida si no se estableció explícitamente otra codificación con `bind_textdomain_codeset()`.

Advertencia: Estas funciones deben evitarse en Python 3, ya que retornan bytes codificados. Es mucho mejor usar alternativas que retornan cadenas Unicode, ya que la mayoría de las aplicaciones de Python querrán manipular el texto legible por humanos como cadenas en lugar de bytes. Además, es posible que

¹ El directorio de configuración regional predeterminado depende del sistema; por ejemplo, en RedHat Linux es `/usr/share/locale`, pero en Solaris es `/usr/lib/locale`. El módulo `gettext` no intenta admitir estos valores predeterminados dependientes del sistema; en cambio, su valor predeterminado es `sys.base_prefix/share/locale` (ver `sys.base_prefix`). Por esta razón, siempre es mejor llamar a `bindtextdomain()` con una ruta absoluta explícita al inicio de su aplicación.

obtenga excepciones inesperadas relacionadas con Unicode si hay problemas de codificación con las cadenas traducidas.

Deprecated since version 3.8, will be removed in version 3.10.

Tenga en cuenta que GNU **gettext** también define un método `dcgettext()`, pero esto no se consideró útil, por lo que actualmente no está implementado.

Aquí hay un ejemplo del uso típico de esta API:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

23.1.2 API basada en clases

La API basada en clases del módulo `gettext` le brinda más flexibilidad y mayor comodidad que la API GNU **gettext**. Es la forma recomendada de localizar sus aplicaciones y módulos de Python. `gettext` define una clase `GNUTranslations` que implementa el análisis de archivos de formato GNU `.mo`, y tiene métodos para retornar cadenas. Las instancias de esta clase también pueden instalarse en el espacio de nombres incorporado como la función `_()`.

`gettext.find(domain, loaledir=None, languages=None, all=False)`

Esta función implementa el algoritmo de búsqueda de archivos estándar `.mo`. Toma un *domain*, idéntico al que toma `textdomain()`. Opcional *loaledir* es como en `bindtextdomain()`. *languages* opcionales es una lista de cadenas, donde cada cadena es un código de idioma.

Si no se proporciona *loaledir*, se utiliza el directorio de configuración regional predeterminado del sistema.² Si no se proporcionan *languages*, se buscan las siguientes variables de entorno: `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, y `LANG`. El primero que retorna un valor no vacío se usa para la variable *languages*. Las variables de entorno deben contener una lista de idiomas separados por dos puntos, que se dividirá en dos puntos para producir la lista esperada de cadenas de código de idioma.

`find()` luego expande y normaliza los idiomas, y luego los itera, buscando un archivo existente construido con estos componentes:

`loaledir/language/LC_MESSAGES/domain.mo`

El primer nombre de archivo que existe es retornado por `find()`. Si no se encuentra dicho archivo, se retorna `None`. Si se proporciona *all*, retorna una lista de todos los nombres de archivo, en el orden en que aparecen en la lista de idiomas o las variables de entorno.

`gettext.translation(domain, loaledir=None, languages=None, class_=None, fallback=False, codeset=None)`

Retorna una instancia de `*Translations` basada en *domain*, *loaledir* y *languages*, que primero se pasan a `find()` para obtener una lista del archivo asociado de rutas `.mo`. Instancias con idéntico nombres de archivo `.mo` se almacenan en caché. La clase real instanciada es *class_* si se proporciona, de lo contrario `GNUTranslations`. El constructor de la clase debe tomar un solo argumento *file object*. Si se proporciona, *codeset* cambiará el juego de caracteres utilizado para codificar cadenas traducidas en los métodos `lgettext()` y `lngettext()`.

Si se encuentran varios archivos, los archivos posteriores se utilizan como retrocesos para los anteriores. Para permitir la configuración de la reserva, `copy.copy()` se utiliza para clonar cada objeto de traducción del caché; los datos de la instancia real aún se comparten con el caché.

Si no se encuentra el archivo `.mo`, esta función genera `OSError` si *fallback* es falso (que es el valor predeterminado) y retorna una instancia de `NullTranslations` si *fallback* es verdadero.

² Vea la nota al pie de página para `bindtextdomain()` arriba.

Distinto en la versión 3.3: *IOError* solía aparecer en lugar de *OSError*.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro *codeset*.

`gettext.install(domain, localedir=None, codeset=None, names=None)`

Esto instala la función `_()` en el espacio de nombres incorporado de Python, basado en *domain*, *localedir* y *codeset* que se pasan a la función `translation()`.

Para el parámetro *names*, consulte la descripción del método del objeto de traducción `install()`.

Como se ve a continuación, generalmente marca las cadenas en su aplicación que son candidatas para la traducción, envolviéndolas en una llamada a la función `_()`, como esta:

```
print(_('This string will be translated.'))
```

Para mayor comodidad, desea que la función `_()` se instale en el espacio de nombres integrado de Python, para que sea fácilmente accesible en todos los módulos de su aplicación.

Deprecated since version 3.8, will be removed in version 3.10: El parámetro *codeset*.

La clase `NullTranslations`

Las clases de traducción son las que realmente implementan la traducción de cadenas de mensajes del archivo fuente original a cadenas de mensajes traducidas. La clase base utilizada por todas las clases de traducción es `NullTranslations`; Esto proporciona la interfaz básica que puede utilizar para escribir sus propias clases de traducción especializadas. Estos son los métodos de `NullTranslations`:

`class gettext.NullTranslations(fp=None)`

Toma un término opcional *file object fp*, que es ignorado por la clase base. Inicializa las variables de instancia «protegidas» `_info` y `_charset` que se establecen mediante clases derivadas, así como `_fallback`, que se establece a través de `add_fallback()`. Luego llama a `self._parse(fp)` if *fp* no es `None`.

`_parse(fp)`

No-op 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)`

Agrega *fallback* como el objeto de respaldo para el objeto de traducción actual. Un objeto de traducción debe consultar la reserva si no puede proporcionar una traducción para un mensaje dado.

`gettext(message)`

Si se ha establecido una reserva, reenvíe `gettext()` a la reserva. De lo contrario, retorna *message*. Anulado en clases derivadas.

`ngettext(singular, plural, n)`

Si se ha establecido una reserva, reenvíe `ngettext()` a la reserva. De lo contrario, retorna *singular* si *n* es 1; volver *plural* de lo contrario. Anulado en clases derivadas.

`pgettext(context, message)`

Si se ha establecido una reserva, reenvía `pgettext()` a la reserva. De lo contrario, retorna el mensaje traducido. Anulado en clases derivadas.

Nuevo en la versión 3.8.

`npgettext(context, singular, plural, n)`

Si se ha establecido una reserva, reenvía `npgettext()` a la reserva. De lo contrario, retorna el mensaje traducido. Anulado en clases derivadas.

Nuevo en la versión 3.8.

`lgettext(message)`

`lngettext(singular, plural, n)`

Equivalente a `gettext()` y `ngettext()`, pero la traducción se retorna como una cadena de bytes

codificada en la codificación del sistema preferida si no se estableció explícitamente ninguna codificación con `set_output_charset()`. Anulado en clases derivadas.

Advertencia: Estos métodos deben evitarse en Python 3. Consulte la advertencia para la función `gettext()`.

Deprecated since version 3.8, will be removed in version 3.10.

info()

Retorna la variable «protected» `_info`, un diccionario que contiene los metadatos encontrados en el archivo del catálogo de mensajes.

charset()

Retorna la codificación del archivo de catálogo de mensajes.

output_charset()

Retorna la codificación utilizada para retornar mensajes traducidos en `gettext()` y `gettext()`.

Deprecated since version 3.8, will be removed in version 3.10.

set_output_charset(charset)

Cambiar la codificación utilizada para retornar mensajes traducidos.

Deprecated since version 3.8, will be removed in version 3.10.

install(names=None)

Este método instala `gettext()` en el espacio de nombres incorporado, vinculándolo a `_`.

Si se proporciona el parámetro `names`, debe ser una secuencia que contenga los nombres de las funciones que desea instalar en el espacio de nombres incorporado además de `_()`. Los nombres admitidos son 'gettext', 'ngettext', 'pgettext', 'npgettext', 'lgettext', y 'lngettext'.

Tenga en cuenta que esta es solo una forma, aunque la más conveniente, para que la función `_()` esté disponible para su aplicación. Debido a que afecta a toda la aplicación globalmente, y específicamente al espacio de nombres incorporado, los módulos localizados nunca deberían instalarse `_()`. En su lugar, deberían usar este código para hacer que `_()` esté disponible para su módulo:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

Esto pone `_()` solo en el espacio de nombres global del módulo y, por lo tanto, solo afecta las llamadas dentro de este módulo.

Distinto en la versión 3.8: Se agregó 'pgettext' y 'npgettext'.

La clase GNUTranslations

El módulo `gettext` proporciona una clase adicional derivada de `NullTranslations`: `GNUTranslations`. Esta clase anula `_parse()` para permitir la lectura de formato GNU `gettext` archivos `.mo` en formato big-endian y little-endian.

`GNUTranslations` analiza los metadatos opcionales del catálogo de traducción. Es una convención con GNU `gettext` para incluir metadatos como la traducción de la cadena vacía. Estos metadatos están en estilos pares **RFC 822**key: value, y deben contener la clave `Project-Id-Version`. Si se encuentra la clave `Content-Type`, entonces la propiedad `charset` se usa para inicializar la variable de instancia «protected» `_charset`, con el valor predeterminado `None` si no se encuentra. Si se especifica la codificación del juego de caracteres, todos los identificadores de mensajes y las cadenas de mensajes leídos del catálogo se convierten a Unicode utilizando esta codificación, de lo contrario se asume ASCII.

Dado que los identificadores de mensaje también se leen como cadenas Unicode, todos los métodos `*gettext()` asumirán los identificadores de mensaje como cadenas Unicode, no cadenas de bytes.

The entire set of key/value pairs are placed into a dictionary and set as the «protected» `_info` instance variable.

Si el número mágico del archivo `.mo` no es válido, el número de versión principal es inesperado, o si se producen otros problemas al leer el archivo, se crea una instancia de `GNUTranslations` puede generar `OSError`.

class `gettext.GNUTranslations`

Los siguientes métodos se anulan desde la implementación de la clase base:

gettext (*message*)

Busca el *message* id en el catálogo y retorna la cadena de caracteres de mensaje correspondiente, como una cadena Unicode. Si no hay una entrada en el catálogo para el *message* id, y se ha establecido una retroceso (*fallback*), la búsqueda se reenvía al método de retroceso `gettext()`. De lo contrario, se retorna el *message* id.

ngettext (*singular*, *plural*, *n*)

Realiza una búsqueda de formas plurales de una identificación de mensaje. *singular* se usa como la identificación del mensaje para fines de búsqueda en el catálogo, mientras que *n* se usa para determinar qué forma plural usar. La cadena del mensaje retornado es una cadena de caracteres Unicode.

Si la identificación del mensaje no se encuentra en el catálogo y se especifica una reserva, la solicitud se reenvía al método de reserva `ngettext()`. De lo contrario, cuando *n* es 1 se retorna *singular*, y *plural* se retorna en todos los demás casos.

Aquí hay un ejemplo:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

pgettext (*context*, *message*)

Busca el *context* y *message* id en el catálogo y retorna la cadena de de caracteres mensaje correspondiente, como una cadena de caracteres Unicode. Si no hay ninguna entrada en el catálogo para el *message* id y *context*, y se ha establecido un retroceso (*fallback*), la búsqueda se reenvía al método de retroceso `pgettext()`. De lo contrario, se retorna el *message* id.

Nuevo en la versión 3.8.

npgettext (*context*, *singular*, *plural*, *n*)

Realiza una búsqueda de formas plurales de una identificación de mensaje. *singular* se usa como la identificación del mensaje para fines de búsqueda en el catálogo, mientras que *n* se usa para determinar qué forma plural usar.

Si la identificación del mensaje para *context* no se encuentra en el catálogo y se especifica una reserva, la solicitud se reenvía al método de reserva `npgettext()`. De lo contrario, cuando *n* *es 1 se retorna **singular*, y *plural* se retorna en todos los demás casos.

Nuevo en la versión 3.8.

lgettext (*message*)

lgettext (*singular*, *plural*, *n*)

Equivalente a `gettext()` y `ngettext()`, pero la traducción se retorna como una cadena de bytes codificada en la codificación del sistema preferida si no se estableció explícitamente ninguna codificación con `set_output_charset()`.

Advertencia: Estos métodos deben evitarse en Python 3. Consulte la advertencia para la función `lgettext()`.

Deprecated since version 3.8, will be removed in version 3.10.

Soporte de catálogo de mensajes de Solaris

El sistema operativo Solaris define su propio formato de archivo binario `.mo`, pero como no se puede encontrar documentación sobre este formato, no es compatible en este momento.

El constructor del catálogo

GNOME usa una versión del módulo `gettext` de James Henstridge, pero esta versión tiene una API ligeramente diferente. Su uso documentado fue:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```

Para la compatibilidad con este módulo anterior, la función `Catalog()` es un alias para la función `translation()` descrita anteriormente.

Una diferencia entre este módulo y el de Henstridge: sus objetos de catálogo admitían el acceso a través de una API de mapeo, pero esto parece estar sin usar y, por lo tanto, actualmente no es compatible.

23.1.3 Internacionalizando sus programas y módulos

La internacionalización (I18N) se refiere a la operación mediante la cual un programa conoce varios idiomas. La localización (L10N) se refiere a la adaptación de su programa, una vez internacionalizado, al idioma local y los hábitos culturales. Para proporcionar mensajes multilingües para sus programas de Python, debe seguir los siguientes pasos:

1. prepare su programa o módulo marcando especialmente cadenas traducibles
2. ejecuta un conjunto de herramientas sobre tus archivos marcados para generar catálogos de mensajes sin procesar
3. crear traducciones específicas del idioma de los catálogos de mensajes
4. use el módulo `gettext` para que las cadenas de caracteres de mensajes se traduzcan correctamente

Para preparar su código para I18N, debe mirar todas las cadenas en sus archivos. Cualquier cadena que deba traducirse debe marcarse envolviéndola en `_('...')` — es decir, una llamada a la función `_()`. Por ejemplo:

```
filename = 'mylog.txt'
message = _('writing a log message')
with open(filename, 'w') as fp:
    fp.write(message)
```

En este ejemplo, la cadena de caracteres `'writing a log message'` está marcada como candidata para la traducción, mientras que las cadenas `'mylog.txt'` y `'w'` no lo están.

Existen algunas herramientas para extraer las cadenas destinadas a la traducción. El programa GNU original `gettext` solo admitía el código fuente C o C++, pero su versión extendida `xgettext` escanea el código escrito en varios idiomas, incluido Python, para encontrar cadenas marcadas como traducibles. `Babel` es una biblioteca de internacionalización de Python que incluye un script `pybabel` para extraer y compilar catálogos de mensajes. El programa de *François Pinard* llamado `xpot` hace un trabajo similar y está disponible como parte de su paquete `po-utils`.

(Python también incluye versiones de Python puro de estos programas, llamadas `pygettext.py` y `msgfmt.py`; algunas distribuciones de Python las instalarán por usted. `pygettext.py` es similar a `xgettext`, pero solo entiende el código fuente de Python y no puede manejar otros lenguajes de programación como C o C++. `pygettext.py` admite una interfaz de línea de comandos similar a `xgettext`; para detalles sobre su uso, ejecute `pygettext.py --help`. `msgfmt.py` es binario compatible con GNU `msgfmt`. Con estos dos programas, es posible que no necesite el paquete GNU `gettext` para internacionalizar sus aplicaciones Python.)

xgettext, **pygettext**, y herramientas similares generan archivos `.po` que son catálogos de mensajes. Son archivos estructurados legibles por humanos que contienen cada cadena marcada en el código fuente, junto con un marcador de posición para las versiones traducidas de estas cadenas.

Las copias de estos archivos `.po` se entregan a los traductores humanos individuales que escriben traducciones para cada lenguaje natural compatible. Envían las versiones completas específicas del idioma como un archivo `<nombre-idioma>.po` que se compila en un archivo binario `.mo` de lectura mecánica utilizando el programa **msgfmt**. Los archivos `.mo` son utilizados por el módulo `gettext` para el procesamiento de traducción real en tiempo de ejecución.

Como use el módulo `gettext` en su código depende de si está internacionalizando un solo módulo o toda su aplicación. Las siguientes dos secciones discutirán cada caso.

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.

Digamos que su módulo se llama «spam» y los diversos archivos `.mo` de traducciones del lenguaje natural del módulo residen en `/usr/share/locale` en formato GNU **gettext**. Esto es lo que pondría en la parte superior de su módulo:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

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.

En el caso simple, entonces, solo necesita agregar el siguiente bit de código al archivo del controlador principal de su aplicación:

```
import gettext
gettext.install('myapplication')
```

Si necesita establecer el directorio de configuración regional, puede pasarlo a la función `install()`:

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

Cambiar idiomas sobre la marcha

Si su programa necesita admitir muchos idiomas al mismo tiempo, es posible que desee crear varias instancias de traducción y luego cambiar entre ellas explícitamente, de esta manera:

```
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
```

(continué en la próxima página)

(proviene de la página anterior)

```
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

Traducciones diferidas

En la mayoría de las situaciones de codificación, las cadenas se traducen donde se codifican. Sin embargo, ocasionalmente, debe marcar cadenas para la traducción, pero diferir la traducción real hasta más tarde. Un ejemplo clásico es:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]

# ...
for a in animals:
    print(a)
```

Aquí, desea marcar las cadenas en la lista de `animals` como traducibles, pero en realidad no desea traducirlas hasta que se impriman.

Aquí hay una manera de manejar esta situación:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

Esto funciona porque la definición ficticia de `_()` simplemente retorna la cadena sin cambios. Y esta definición ficticia anulará temporalmente cualquier definición de `_()` en el espacio de nombres incorporado (hasta el comando `del`). Sin embargo, tenga cuidado si tiene una definición previa de `_()` en el espacio de nombres local.

Tenga en cuenta que el segundo uso de `_()` no identificará «a» como traducible al programa **gettext**, porque el parámetro no es un literal de cadena.

Otra forma de manejar esto es con el siguiente ejemplo:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

En este caso, está marcando cadenas traducibles con la función `N_()`, que no entrará en conflicto con ninguna definición de `_()`. Sin embargo, deberá enseñar a su programa de extracción de mensajes a buscar cadenas traducibles marcadas con `N_()`. **xgettext**, **pygettext**, `pybabel extract`, y **xpot** todos soportan esto mediante el uso de `-k` interruptor de línea de comando. La elección de `N_()` aquí es totalmente arbitraria; podría haber sido igual de fácil `MarkThisStringForTranslation()`.

23.1.4 Agradecimientos

Las siguientes personas contribuyeron con código, comentarios, sugerencias de diseño, implementaciones anteriores y una valiosa experiencia para la creación de este módulo:

- *Peter Funk*
- *James Henstridge*
- *Juan David Ibáñez Palomar*
- *Marc-André Lemburg*
- *Martin von Löwis*
- *François Pinard*
- *Barry Warsaw*
- *Gustavo Niemeyer*

Notas al pie

23.2 locale — Servicios de internacionalización

Source code: [Lib/locale.py](#)

El módulo `locale` abre el acceso a la base de datos y la funcionalidad de POSIX locale. El mecanismo POSIX locale permite a los programadores tratar ciertos problemas culturales en una aplicación, sin requerir que el programador conozca todos los detalles de cada país donde se ejecuta el software.

El módulo `locale` se implementa en la parte superior del módulo `_locale`, que a su vez utiliza una implementación de configuración regional ANSI C si está disponible.

El módulo `locale` define las siguientes excepciones y funciones:

exception `locale.Error`

Cuando el `locale` pasado a `setlocale()` no es reconocido, se lanza una excepción.

`locale.setlocale(category, locale=None)`

Si `locale` está dado y no es `None`, `setlocale()` modifica la configuración de localización para `category`. Las categorías disponibles se enumeran en la descripción de los datos que figura a continuación. `locale` puede ser una cadena de caracteres, o una iterable de dos cadenas de caracteres (código de idioma y codificación). Si es iterable, se convierte a un nombre de configuración regional usando el motor de *aliasing* de la configuración regional. Una cadena de caracteres vacía especifica la configuración predeterminada del usuario. Si la modificación de la localización falla, se genera la excepción `Error`. Si tiene éxito, se retorna la nueva configuración de localización.

Si se omite `locale` o es `None`, se retorna la configuración actual para `category`.

`setlocale()` no es seguro para subprocessos en la mayoría de los sistemas. Las aplicaciones normalmente comienzan con una llamada de

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

Esto establece la configuración regional de todas las categorías en la configuración predeterminada del usuario (normalmente especificada en la variable de entorno `LANG`). Si la configuración regional no se cambia a partir de entonces, el uso de multiprocesamiento, no debería causar problemas.

`locale.localeconv()`

retorna la base de datos de las convenciones locales como diccionario. Este diccionario tiene las siguientes cadenas de caracteres como claves:

Categoría	Clave	Significado
<i>LC_NUMERIC</i>	'decimal_point'	Carácter de punto decimal.
	'agrupación'	Secuencia de números que especifica qué posiciones relativas se espera el 'miles_sep'. Si la secuencia termina con <i>CHAR_MAX</i> , no se realiza ninguna agrupación adicional. Si la secuencia termina con un 0, el último tamaño de grupo se usa repetidamente.
	'thousands_sep'	Carácter utilizado entre grupos.
<i>LC_MONETARY</i>	'int_curr_symbol'	Símbolo de moneda internacional.
	'currency_symbol'	Símbolo de moneda local.
	'p_cs_precedes/n_cs_precedes'	Si el símbolo de moneda precede al valor (para valores positivos o negativos).
	'p_sep_by_space/n_sep_by_space'	Si el símbolo de moneda está separado del valor por un espacio (para valores positivos o negativos).
	'mon_decimal_point'	Punto decimal utilizado para valores monetarios.
	'frac_digits'	Número de dígitos fraccionarios utilizados en el formateo local de valores monetarios.
	'int_frac_digits'	Número de dígitos fraccionarios utilizados en el formateo internacional de valores monetarios.
	'mon_thousands_sep'	Separador de grupo utilizado para valores monetarios.
	'mon_grouping'	Equivalente a 'grouping', utilizada para valores monetarios.
	'positive_sign'	Símbolo utilizado para anotar un valor monetario positivo.
	'negative_sign'	Símbolo utilizado para anotar un valor monetario negativo.
	'p_sign_posn/n_sign_posn'	La posición del signo (para respuestas positivas, valores negativos), ver abajo.

Todos los valores numéricos se pueden establecer a *CHAR_MAX* para indicar que no hay ningún valor especificado en esta configuración regional.

Los valores posibles para 'p_sign_posn' y "n_sign_posn" se dan a continuación.

Valor	Explicación
0	Moneda y valor están rodeados por paréntesis.
1	El signo debe preceder al valor y al símbolo de moneda.
2	El signo debe seguir el valor y el símbolo de moneda.
3	El signo debe preceder inmediatamente al valor.
4	El signo debe seguir inmediatamente al valor.
CHAR_MAX	No se especifica nada en esta configuración regional.

La función establece temporalmente el `LC_CTYPE` de configuración regional al `LC_NUMERIC` de la configuración regional o el `LC_MONETARY` de la configuración local si las configuraciones locales son diferentes y las cadenas de caracteres numéricas o monetarias no son ASCII. Este cambio temporal afecta a otros hilos.

Distinto en la versión 3.7: La función ahora establece temporalmente el `LC_CTYPE` de la configuración local al `LC_NUMERIC` en algunos casos.

`locale.nl_langinfo(option)`

retorna información específica de la configuración regional como una cadena de caracteres. Esta función no está disponible en todos los sistemas, y el conjunto de opciones posibles también puede variar entre plataformas. Los posibles valores de argumento son números, para los cuales las constantes simbólicas están disponibles en el módulo de configuración regional.

La función `nl_langinfo()` acepta una de las siguientes claves. La mayoría de las descripciones están tomadas de la descripción correspondiente en la biblioteca C de GNU.

`locale.CODESET`

Obtiene una cadena de caracteres con el nombre de la codificación de caracteres utilizada en la configuración regional seleccionada.

`locale.D_T_FMT`

Obtiene una cadena de caracteres que se puede utilizar como cadena de caracteres de formato para `time.strftime()` para representar la fecha y la hora de una manera específica de la configuración regional.

`locale.D_FMT`

Obtiene una cadena de caracteres que se puede utilizar como cadena de formato para `time.strftime()` para representar una fecha de una manera específica de la configuración regional.

`locale.T_FMT`

Obtiene una cadena de caracteres que se puede utilizar como cadena de formato para `time.strftime()` para representar un tiempo de una manera específica de la configuración regional.

`locale.T_FMT_AMPM`

Obtiene una cadena de caracteres de formato para `time.strftime()` para representar el tiempo en el formato am/pm.

`DAY_1 ... DAY_7`

Consigue el nombre del n-ésimo día de la semana.

Nota: Esto sigue la convención estadounidense de `DAY_1` siendo domingo, no la convención internacional (ISO 8601) que el lunes es el primer día de la semana.

`ABDAY_1 ... ABDAY_7`

Obtener el nombre abreviado del n-ésimo día de la semana.

`MON_1 ... MON_12`

Obtener el nombre del n-ésimo mes.

`ABMON_1 ... ABMON_12`

Obtener el nombre abreviado del enésimo mes.

`locale.RADIXCHAR`

Obtener el carácter radical (punto decimal, coma decimal, etc.).

`locale.THOUSEP`

Obtener el carácter separador de miles (grupos de tres dígitos).

`locale.YESEXPR`

Obtener una expresión regular que se pueda usar con la función `regex` para reconocer una respuesta positiva a una pregunta de sí / no.

Nota: La expresión está en la sintaxis adecuada para la función `regex()` de la biblioteca C, que podría diferir de la sintaxis utilizada en `re`.

`locale.NOEXPR`

Obtener una expresión regular que se puede usar con la función `regex(3)` para reconocer una respuesta negativa a una pregunta de sí/no.

`locale.CRNCYSTR`

Obtener el símbolo de moneda, precedido por «-» si el símbolo debe aparecer antes del valor, «+» si el símbolo debe aparecer después del valor, o «.» si el símbolo debe reemplazar el carácter raíz.

`locale.ERA`

Obtener una cadena de caracteres que represente la era utilizada en la configuración regional actual.

La mayoría de las configuraciones regionales no definen este valor. Un ejemplo de una localidad que sí define este valor es la japonesa. En Japón, la representación tradicional de fechas incluye el nombre de la época correspondiente al reinado del entonces emperador.

Normalmente no debería ser necesario utilizar este valor directamente. Especificar el modificador E en sus cadenas de caracteres de formato hace que la función `time.strftime()` use esta información. El formato de la cadena de caracteres retornada no está especificado, y por lo tanto no debe asumir conocimiento de él en diferentes sistemas.

`locale.ERA_D_T_FMT`

Obtener una cadena de caracteres de formato para `time.strftime()` para representar la fecha y la hora de una manera específica de la era.

`locale.ERA_D_FMT`

Obtener una cadena de caracteres de formato para `time.strftime()` para representar una fecha de una manera basada en una era específica.

`locale.ERA_T_FMT`

Obtener una cadena de caracteres de formato para `time.strftime()` para representar una hora de una manera basada en una era específica.

`locale.ALT_DIGITS`

Obtener una representación de hasta 100 valores utilizados para representar los valores 0 a 99.

`locale.getdefaultlocale([envvars])`

Intenta determinar la configuración regional por defecto y los retorna como una tupla del formulario (código de idioma, codificación).

De acuerdo con POSIX, un programa que no ha llamado a `setlocale(LC_ALL, '')` se ejecuta utilizando la configuración regional portátil “C”. Llamar a `setlocale(LC_ALL, '')` le permite usar la configuración regional predeterminada definida por la variable `LANG`. Dado que no queremos interferir con la configuración de configuración regional actual, emulamos el comportamiento de la manera descrita anteriormente.

Para mantener la compatibilidad con otras plataformas, no sólo se prueba la variable `LANG` sino una lista de variables dadas como parámetro `envvars`. Se utilizará la primera que se encuentre definida. `envvars` por defecto a la ruta de búsqueda utilizada en GNU gettext; siempre debe contener el nombre de la variable `'LANG'`. La ruta de búsqueda GNU gettext contiene `'LC_ALL'`, `'LC_CTYPE'`, `'LANG'` y `'LANGUAGE'`, en ese orden.

Excepto por el código `'C'`, el código de lenguaje corresponde a **RFC 1766**. *language code* y *encoding* pueden ser `None` si sus valores no pueden determinarse.

`locale.getlocale (category=LC_CTYPE)`

retorna la configuración actual para la categoría de configuración regional dada como secuencia que contiene *language code*, *coding*. *category* puede ser uno de los valores `LC_` * excepto `LC_ALL`. Por defecto es `LC_CTYPE`.

Excepto por el código 'C', el código de lenguaje corresponde a [RFC 1766](#). *language code* y *encoding* pueden ser `None` si sus valores no pueden determinarse.

`locale.getpreferredencoding (do_setlocale=True)`

retorna la codificación utilizada para los datos de texto, según las preferencias del usuario. Las preferencias del usuario se expresan de manera diferente en diferentes sistemas, y puede que no estén disponibles programáticamente en algunos sistemas, por lo que esta función solo retorna una suposición.

En algunos sistemas, es necesario invocar `setlocale()` para obtener las preferencias del usuario, por lo que esta función no es segura para subprocesos. Si invocar `setlocale` no es necesario o deseado, `do_setlocale` se debe ajustar a `False`.

En Android o en el modo UTF-8 (`-X utf8` opción), siempre retorna 'UTF-8', se ignoran la configuración regional y el argumento `do_setlocale`.

Distinto en la versión 3.7: La función ahora siempre retorna UTF-8 en Android o si el modo UTF-8 está habilitado.

`locale.normalize (localename)`

retorna un código de configuración regional normalizado para el nombre configuración regional dado. El código de configuración regional retornado está formateado para usarse con `setlocale()`. Si la normalización falla, el nombre original se retorna sin cambios.

Si no se conoce la codificación dada, la función por defecto codifica el código de configuración regional como `setlocale()`.

`locale.resetlocale (category=LC_ALL)`

Establece la configuración regional de *category* a la configuración predeterminada.

La configuración predeterminada se determina llamando a `getdefaultlocale()`. *category* por defecto a `LC_ALL`.

`locale.strcoll (string1, string2)`

Compara dos cadenas según la configuración actual `LC_COLLATE`. Como cualquier otra función de comparación, retorna un valor negativo o positivo, o 0, dependiendo de si *string1* compila antes o después de *string2* o es igual a él.

`locale.strxfrm (string)`

Transforma una cadena en una que se puede utilizar en comparaciones de localización. Por ejemplo, `strxfrm(s1) < strxfrm(s2)` es equivalente a `strcoll(s1, s2) < 0`. Esta función se puede utilizar cuando la misma cadena se compara repetidamente, p. ej., al agrupar una secuencia de cadenas.

`locale.format_string (format, val, grouping=False, monetary=False)`

Formatea un número *val* según la configuración actual `LC_NUMERIC`. El formato sigue las convenciones del operador `%`. Para los valores en coma flotante, el punto decimal se modifica si procede. Si *grouping* es verdadero, también se tiene en cuenta el agrupamiento.

Si *monetary* es verdadero, la conversión utiliza separador monetario de miles y cadenas de caracteres de agrupación.

Procesa especificadores de formato como en `format % val`, pero tiene en cuenta los ajustes de configuración regional actuales.

Distinto en la versión 3.7: Se añadió el parámetro de la palabra clave *monetary*.

`locale.format (format, val, grouping=False, monetary=False)`

Tenga en cuenta que esta función funciona como `format_string()` pero sólo funcionará para exactamente un especificador `%char`. Por ejemplo, `'%f'` y `'%.0f'` son especificadores válidos, pero `'%f KiB'` no lo es.

Para cadenas de caracteres de formato completas, use `format_string()`.

Obsoleto desde la versión 3.7: Use `format_string()` en su lugar.

`locale.currency(val, symbol=True, grouping=False, international=False)`
Formatea un número *val* según la configuración actual `LC_MONETARY`.

La cadena de caracteres retornada incluye el símbolo de moneda si *symbol* es verdadero, que es el valor predeterminado. Si *grouping* es verdadero (que no es el valor predeterminado), la agrupación se realiza con el valor. Si *international* es verdadero (que no es el valor predeterminado), se utiliza el símbolo de moneda internacional.

Tenga en cuenta que esta función no funcionará con la configuración regional “C”, por lo que primero debe establecer una configuración regional (*locale*) a través de `setlocale()`.

`locale.str(float)`
Formatea un número de punto flotante usando el mismo formato que la función integrada `str(float)`, pero toma en cuenta el punto decimal.

`locale.delocalize(string)`
Convierte una cadena de caracteres en una cadena de números normalizada, siguiendo la configuración `LC_NUMERIC`.
Nuevo en la versión 3.5.

`locale atof(string)`
Convierte una cadena de caracteres a un número de punto flotante, siguiendo la configuración `LC_NUMERIC`.

`locale atoi(string)`
Convierte una cadena de caracteres a un entero, siguiendo las convenciones `LC_NUMERIC`.

`locale.LC_CTYPE`
Configuración regional de localización para las funciones de tipo caracter. Dependiendo de la configuración de esta categoría, las funciones del módulo *string* que se ocupan de casos cambian su comportamiento.

`locale.LC_COLLATE`
Categoría de configuración regional para ordenar cadenas de caracteres. Las funciones `strcoll()` y `strxfrm()` del módulo *locale* están afectadas.

`locale.LC_TIME`
Categoría de configuración regional para el formateo de hora. La función `time.strftime()` sigue estas convenciones.

`locale.LC_MONETARY`
Categoría de configuración regional para el formateo de valores monetarios. Las opciones disponibles están disponibles en la función `localeconv()`.

`locale.LC_MESSAGES`
Categoría de configuración regional para visualización de mensajes. Python actualmente no admite mensajes de configuración regional específicos de la aplicación. Los mensajes mostrados por el sistema operativo, como los retornados por `os.strerror()` podrían verse afectados por esta categoría.

`locale.LC_NUMERIC`
Categoría de configuración regional para formateo de numeros. Las funciones `format()`, `atoi()`, `atof()` y `str()` del módulo *locale* están afectados por esa categoría. Todas las demás operaciones de formato numérico no están afectadas.

`locale.LC_ALL`
Combinación de todos los ajustes de configuración regional. Si se utiliza este indicador cuando se cambia la localización, se intenta establecer la localización para todas las categorías. Si eso falla para cualquier categoría, ninguna categoría se cambia en absoluto. Cuando la configuración regional se recupera usando este indicador, se retorna una cadena de caracteres que indica la configuración para todas las categorías. Esta cadena de caracteres se puede usar más tarde para restaurar la configuración.

`locale.CHAR_MAX`
Esta es una constante simbólica utilizada para diferentes valores retornados por `localeconv()`.

Ejemplo:

```
>>> 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\xfe4n', '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
```

23.2.1 Segundo plano, detalles, indicaciones, consejos y advertencias

El estándar C define la configuración regional como una propiedad de todo el programa que puede ser relativamente costosa de cambiar. Además de eso, alguna implementación se rompe de tal manera que los cambios de configuración local frecuentes pueden causar volcados de núcleo. Esto hace que la configuración regional sea algo dolorosa de usar correctamente.

Inicialmente, cuando se inicia un programa, la configuración regional es la configuración regional C, no importa cuál sea la configuración regional preferida por el usuario. Hay una excepción: la categoría `LC_CTYPE` se cambia al inicio para establecer la codificación de la configuración regional actual en la codificación de configuración regional preferida del usuario. El programa debe decir explícitamente que quiere la configuración regional preferida del usuario para otras categorías llamando a `setlocale(LC_ALL, '')`.

Generalmente es una mala idea llamar `setlocale()` en alguna rutina de biblioteca, ya que como efecto secundario afecta a todo el programa. Guardar y restaurar es casi igual de malo: es caro y afecta a otros hilos que se ejecutan antes de que los ajustes se hayan restaurado.

Si, al codificar un módulo para uso general, se necesita una versión configuración regional independiente de una operación que se ve afectada por la localización (como ciertos formatos utilizados con `time.strftime()`), tendrá que encontrar una manera de hacerlo sin usar la rutina de biblioteca estándar. Aún mejor es convencerse a sí mismo de que el uso de la configuración regional está bien. Sólo como último recurso debe documentar que su módulo no es compatible con la configuración regional no-C.

La única manera de realizar operaciones numéricas según la configuración regional es utilizar las funciones especiales definidas por este módulo: `atof()`, `atoi()`, `formato()`, `str()`.

No hay manera de realizar conversiones de casos y clasificaciones de caracteres de acuerdo a la configuración regional. Para cadenas de texto (Unicode) estas se hacen de acuerdo con el valor de carácter solamente, mientras que para cadenas de bytes, las conversiones y clasificaciones se hacen de acuerdo con el valor ASCII del byte, y bytes cuyo bit alto se establece (es decir, bytes no ASCII) nunca se convierten o se consideran parte de una clase de caracteres como letra o espacio en blanco.

23.2.2 Para escritores de extensión y programas que incrustan Python

Los módulos de extensión nunca deben llamar a `setlocale()`, excepto para averiguar cuál es la configuración regional actual. Pero dado que el valor de retorno sólo se puede utilizar portablemente para restaurarlo, eso no es muy útil (excepto quizás para averiguar si la localización es o no es C).

Cuando el código de Python utiliza el módulo `locale` para cambiar la configuración regional, esto también afecta a la aplicación de incrustación. Si la aplicación de incrustación no quiere que esto suceda, debe eliminar el módulo de extensión `_locale` (que hace todo el trabajo) de la tabla de módulos incorporados en el archivo de configuración `config.c`, y asegúrese de que el módulo `_locale` no es accesible como una biblioteca compartida.

23.2.3 Acceso a los catálogos de mensajes

```
locale.gettext(msg)
locale.dgettext(domain, msg)
locale.dcgettext(domain, msg, category)
locale.textdomain(domain)
locale.bindtextdomain(domain, dir)
```

El módulo de configuración regional expone la interfaz *gettext* de la biblioteca C en sistemas que proporcionan esta interfaz. Consta de las funciones `gettext()`, `dgettext()`, `dcgettext()`, `textdomain()`, `binddomaintext()`, y `bind_textdomain_codeset()`. Estas son similares a las mismas funciones en el módulo *gettext*, pero usan el formato binario de la biblioteca C para catálogos de mensajes, y los algoritmos de búsqueda de la biblioteca C para localizar catálogos de mensajes.

Las aplicaciones de Python normalmente no deberían tener que invocar estas funciones, y deberían usar *gettext* en su lugar. Una excepción conocida a esta regla son las aplicaciones que enlazan con bibliotecas C adicionales que invocan internamente `gettext()` o `dcgettext()`. Para estas aplicaciones, puede ser necesario vincular el dominio de texto, para que las bibliotecas puedan localizar adecuadamente sus catálogos de mensajes.

Frameworks de programa

Los módulos descritos en este capítulo son *frameworks* que dictarán en gran medida la estructura de su programa. Actualmente, los módulos descritos aquí están orientados para escribir en las interfaces de línea de comandos.

La lista completa de módulos descritos en este capítulo es:

24.1 `turtle` — Gráficos con *Turtle*

Código fuente: [Lib/turtle.py](#)

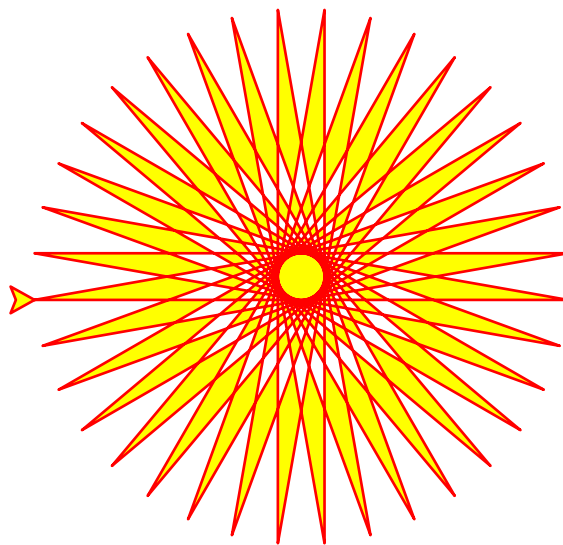
24.1.1 Introducción

Gráficas Turtle es una forma muy habitual de introducción a la programación para niñas y niños. Era parte original del lenguaje de programación Logo, desarrollado por Wally Feurzeig, Seymour Papert y Cynthia Solomon en 1967.

Imagina una tortuga robot que empieza en las coordenadas (0, 0) en un plano x-y. Después de un `import turtle`, dele el comando `turtle.forward(15)`, y se mueve (¡en la pantalla!) 15 píxeles en la dirección en la que se encuentra, dibujando una línea mientras se mueve. Dele el comando `turtle.right(25)`, y rotará en el lugar, 25 grados, en sentido horario.

Turtle star

Turtle puede dibujar figuras intrincadas usando programas que repiten movimientos simples.



```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

Al combinar estos comandos y otros similares, se pueden dibujar figuras intrincadas y formas.

El módulo `turtle` es una reimplementación extendida del mismo módulo de la distribución estándar Python hasta la versión 2.5.

Trata de mantener los méritos del viejo módulo y ser (casi) 100% compatible con él. Esto implica en primer lugar, habilitar al programador que está aprendiendo, el uso de todos los comandos, clases y métodos de forma interactiva cuando usa el módulo desde el IDLE ejecutado con la opción `-n`.

El módulo `turtle` provee las primitivas gráficas, tanto en orientación procedimental como orientada a objetos. Como usa el módulo `tkinter` para las gráficas subyacentes, necesita tener instalada una versión de Python con soporte TK.

La interface orientada a objetos usa esencialmente clases dos+dos:

1. La clase `TurtleScreen` define una ventana gráfica como base para las tortugas dibujantes. Su constructor necesita una clase `tkinter.Canvas` o una a `ScrolledCanvas` como argumento. Se debe usar cuando `turtle` es usado como parte de una aplicación.

La función `Screen()` devuelve un objeto *singleton* de la subclase `TurtleScreen`. Esta función debe utilizarse cuando `turtle` se usa como una herramienta independiente para hacer gráficos. Siendo un objeto *singleton*, no es posible que tenga herencias de su clase.

Todos los métodos de `TurtleScreen/Screen` también existen como funciones. Por ejemplo, como parte de la interface orientada a procedimientos.

2. `RawTurtle` (alias: `RawPen`) Define los objetos Turtle con los cuales dibujar con la clase `TurtleScreen`. Su constructor necesita como argumento un `Canvas`, `ScrolledCanvas` o `TurtleScreen`, así el objeto `RawTurtle` sabe donde dibujar.

Derivada de `RawTurtle` está la subclase `Turtle` (alias: `Pen`), que dibuja en «la» instancia `Screen` que se

crea automáticamente, si no está presente.

Todos los métodos de *RawTurtle/Turtle* también existen como funciones. Por ejemplo, como parte de la interface orientada a procedimientos.

La interface procedimental provee funciones que son derivadas de los métodos de las clases *Screen* y *Turtle*. Tienen los mismos nombres que los métodos correspondientes. Un objeto *Screen* es creado automáticamente cada vez que una función derivada de un método *Screen* es llamado. Un objeto *Turtle* (innombrado) se crea automáticamente cada vez que se llama a una función derivada de un método *Turtle*.

Para usar varias tortugas en una pantalla se tiene que usar la interface orientada a objetos.

Nota: En la siguiente documentación se proporciona la listas de argumentos para las funciones. Los métodos, por su puesto, tienen el argumento principal adicional *self* que se omite aquí.

24.1.2 Reseña de los métodos disponibles para Turtle y Screen

Métodos Turtle

Movimiento de Turtle

Mover y dibujar

```
forward() | fd()
backward() | bk() | back()
right() | rt()
left() | lt()
goto() | setpos() | setposition()
setx()
sety()
setheading() | seth()
home()
circle()
dot()
stamp()
clearstamp()
clearstamps()
undo()
speed()
```

Mostrar el estado de la tortuga

```
position() | pos()
towards()
xcor()
ycor()
heading()
distance()
```

Ajuste y unidades de medida

```
degrees()
radians()
```

Control del lápiz

Estado de dibujo

```
pendown() | pd() | down()
penup() | pu() | up()
pensize() | width()
pen()
isdown()
```

Control del color

```
color()
pencolor()
fillcolor()
```

Relleno

```
filling()
begin_fill()
end_fill()
```

Más controles de dibujo

```
reset()
clear()
write()
```

Estado de la Tortuga

Visibilidad

```
showturtle() | st()
hideturtle() | ht()
isvisible()
```

Apariencia

```
shape()
resizemode()
shapeseize() | turtlesize()
shearfactor()
settiltangle()
tiltangle()
tilt()
shapetransform()
get_shapepoly()
```

Usando eventos

```
onclick()
onrelease()
ondrag()
```

Métodos especiales de *Turtle*

```
begin_poly()
end_poly()
get_poly()
clone()
getturtle() | getpen()
getscreen()
setundobuffer()
undobufferentries()
```

Métodos de TurtleScreen/Screen

Control de ventana

```
bgcolor()  
bgpic()  
clear() | clearscreen()  
reset() | resetscreen()  
screensize()  
setworldcoordinates()
```

Control de animación

```
delay()  
tracer()  
update()
```

Usando eventos de pantalla

```
listen()  
onkey() | onkeyrelease()  
onkeypress()  
onclick() | onscreenclick()  
ontimer()  
mainloop() | done()
```

Configuración y métodos especiales

```
mode()  
colormode()  
getcanvas()  
getshapes()  
register_shape() | addshape()  
turtles()  
window_height()  
window_width()
```

Métodos de entrada

```
textinput()  
numinput()
```

Métodos específicos para Screen

```
bye()  
exitonclick()  
setup()  
title()
```

24.1.3 Métodos de *RawTurtle/Turtle* Y sus correspondientes funciones

Casi todos los ejemplos de esta sección se refieren a una instancia *Turtle* llamada `turtle`.

Movimiento de Turtle

`turtle.forward(distance)`
`turtle.fd(distance)`

Parámetros `distance` – un número (entero o flotante)

Mover hacia adelante la tortuga la *distance* especificada, en la dirección en la que la tortuga apunta.

```
>>> 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)`

Parámetros `distance` – un número

Mover hacia atrás la tortuga la *distance* especificada, opuesta a la dirección en que la tortuga apunta. No cambia la dirección de la tortuga.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

`turtle.right(angle)`
`turtle.rt(angle)`

Parámetros `angle` – un número (entero o flotante)

Gira la tortuga a la derecha tomando los *angle* como unidad de medida. (La unidad de medida por defecto son los grado, pero se puede configurar a través de las funciones `degrees()` y `radians()`.) La orientación de los ángulos depende del modo en que está la tortuga, ver `mode()`.

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

`turtle.left(angle)`
`turtle.lt(angle)`

Parámetros `angle` – un número (entero o flotante)

Gira la tortuga a la izquierda tomando los *ángulos* como unidad de medida. (La unidad de medida por defecto son los grado, pero se puede configurar a través de las funciones `degrees()` y `radians()`.) La orientación de los ángulos depende del modo en que está la tortuga, ver `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)
```

Parámetros

- **x** – un número o un par/vector de números
- **y** – un número o None

Si **y** es None, **x** debe ser un par de coordenadas o un *Vec2D* (ejemplo: según lo devuelto por *pos()*).

Mueve la tortuga a una posición absoluta. Si el lápiz está bajo, traza una línea. No cambia la orientación de la tortuga.

```
>>> tp = turtle.pos()
>>> tp
(0.00, 0.00)
>>> 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)
```

Parámetros x – un número (entero o flotante)

Establece la primera coordenada de la tortuga a **x**, deja la segunda coordenada sin cambios.

```
>>> turtle.position()
(0.00, 240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00, 240.00)
```

```
turtle.sety(y)
```

Parámetros y – un número (entero o flotante)

Establece la segunda coordenada de la tortuga a **y**, deja la primera coordenada sin cambios.

```
>>> turtle.position()
(0.00, 40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00, -10.00)
```

```
turtle.setheading(to_angle)
turtle.seth(to_angle)
```

Parámetros to_angle – un número (entero o flotante)

Establece la orientación de la tortuga a *to_angle*. Aquí hay algunas direcciones comunes en grados:

modo estándar	modo logo
0 - este	0 - norte
90 - norte	90 - este
180 - oeste	180 - sur
270 - sur	270 - oeste

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

`turtle.home()`

Mueve la tortuga al origen – coordenadas (0,0) – y establece la orientación a la orientación original (que depende del modo, ver `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)`

Parámetros

- **radius** – un número
- **extent** – un número (o None)
- **steps** – un entero (o None)

Dibuja un círculo con el *radius* (radio) dado. El centro es *radius* unidades a la izquierda de la tortuga; *extent* – un ángulo – determina que parte del círculo se dibuja. Si no se pasa *extent*, dibuja el círculo entero. Si *extent* no es un círculo completo, un punto final del arco es la posición actual de lápiz. Dibuja el arco en dirección antihoraria si *radius* es positivo, si no en dirección horaria. Finalmente la dirección de la tortuga es modificada por el aumento de *extent*.

Como el círculo se aproxima a un polígono regular inscripto, *steps* determina el número de pasos a usar. Si no se da, será calculado automáticamente. Puede ser usado para dibujar polígonos regulares.

```
>>> 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)`

Parámetros

- **size** – un entero ≥ 1 (si se da)
- **color** – un *colorstring* o una tupla numérica de color

Dibuja un punto circular con diámetro *size*, usando *color*. Si *size* no se da, el máximo de pensize+4 y 2*pensize es usado.

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

`turtle.stamp()`

Estampa una copia de la forma de la tortuga en el lienzo en la posición actual. Devuelve un `stamp_id` por cada estampa, que puede ser usado para borrarlo al llamar `clearstamp(stamp_id)`.

```
>>> turtle.color("blue")
>>> turtle.stamp()
11
>>> turtle.fd(50)
```

`turtle.clearstamp(stampid)`

Parámetros `stampid` – un entero, debe devolver el valor de la llamada previa de la función `stamp()` call

Borra la estampa con el *stampid* dado.

```
>>> 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)`

Parámetros `n` – un entero (o None)

Borra todas o las primeros/últimos *n* estampas de la tortuga. Si *n* es None, borra todas las estampas, Si $n > 0$ borra la primera estampa *n*, sino y $n < 0$ borra la última estampa *n*.

```
>>> 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()`

Deshace (repetidamente) la(s) última(s) acción(es) de la tortuga. El número de acciones a deshacer es determinado por el tamaño del *undobuffer*.

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

`turtle.speed(speed=None)`

Parámetros `speed` – un entero en el rango 0..10 o un *speedstring* (ver abajo)

Establecer la velocidad de la tortuga a un valor entero en el rango 0..10. Si no se pasa ningún argumento, vuelve a la velocidad actual.

Si el parámetro de entrada es un número mayor que 10 o menor que 0.5, la velocidad se establece a 0. La frase acerca de la velocidad es mapeada a valores de velocidad de la siguiente manera:

- «fastest»: 0
- «fast»: 10
- «normal»: 6
- «slow»: 3
- «slowest»: 1

Velocidades de 1 a 10 generan una animación cada vez más rápida al dibujar las líneas y en el giro de la tortuga.

Atención: `speed = 0` implica que *no* habrá ninguna animación. Los métodos *forward/back* harán que la tortuga salte, de la misma manera que *left/right* hará que la tortuga gire instantáneamente.

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

Mostrar el estado de la tortuga

`turtle.position()`

`turtle.pos()`

Devuelve la posición actual de la tortuga (x,y) (como un vector *Vec2D*)

```
>>> turtle.pos()
(440.00,-0.00)
```

`turtle.towards(x, y=None)`

Parámetros

- **x** – un número o par de vectores numéricos o una instancia de la tortuga
- **y** – un número si x es un número, si no 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()`Devuelve la coordenada x de la tortuga.

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28, 76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

`turtle.ycor()`Devuelve la coordenada y de la tortuga.

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00, 86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

`turtle.heading()`Devuelve la orientación actual de la tortuga (el valor depende del modo de la tortuga, ver `mode()`).

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

`turtle.distance(x, y=None)`**Parámetros**

- **x** – un número o par de vectores numéricos o una instancia de la tortuga
- **y** – un número si x es un número, si no `None`

Devuelve la distancia desde la tortuga al vector (x,y) dado, otra instancia de la tortuga, el valor es unidad pasos de tortuga.

```
>>> turtle.home()
>>> turtle.distance(30, 40)
50.0
>>> turtle.distance((30, 40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

Configuración de las medidas`turtle.degrees (fullcircle=360.0)`**Parámetros** `fullcircle` – un número

Establece la unidad de medida del ángulo, por ejemplo establece el número de «grados» para un círculo completo. El valor por defecto es 36 grados.

```
>>> turtle.home()
>>> turtle.left(90)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> 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()Establece la unidad de medida del ángulo a radianes. Equivalente a `degrees(2*math.pi)`.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

Control del lápiz

Estado de dibujo

turtle.pendown()
turtle.pd()
turtle.down()

Baja el lápiz – dibuja mientras se mueve.

turtle.penup()
turtle.pu()
turtle.up()

Levanta el lápiz – no dibuja mientras se mueve.

turtle.pensize() (*width=None*)
turtle.width() (*width=None*)**Parámetros width** – un número positivoEstablece el grosor de la línea a *width* o lo devuelve. Si *resizemode* se establece a «auto» y *turtleshape* es un polígono, ese polígono es dibujado con el mismo grosor de línea. Si no se dan argumentos, devuelve el grosor del lápiz actual.

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)    # from here on lines of width 10 are drawn
```

turtle.pen() (*pen=None, **pendict*)**Parámetros**

- **pen** – un diccionario con algunos o todos las claves listadas debajo
- **pendict** – uno o más argumentos-palabras claves con las claves listadas debajo como palabras claves

Devuelve o establece los atributos del lápiz en un «diccionario-lápiz» con el siguiente para de valores/claves:

- «shown»: True/False

- «pendown»: True/False
- «pencolor»: color-string or color-tuple
- «fillcolor»: color-string or color-tuple
- «pensize»: número positivo
- «speed»: número en el rango 0..10
- «resizemode»: «auto» or «user» or «noresize»
- «stretchfactor»: (número positivo, número positivo)
- «outline»: número positivo
- «tilt»: número

Este diccionario puede usarse como argumento de una llamada subsecuente a la función `pen()` para restaurar el estado anterior del lápiz. Más aún, uno o más de estos atributos pueden darse como argumentos claves. Esto puede usarse para establecer diferentes atributos del lápiz en una sola definición.

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

`turtle.isdown()`

Devuelve True si el lápiz está abajo, si está arriba False.

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

Control del color

`turtle.pencolor(*args)`

Devuelve o establece el color del lápiz.

Se permiten cuatro formatos de entrada:

`pencolor()` Devuelve el color del lápiz actual como una palabra específica de algún color o como una tupla (ver ejemplo). Puede ser usado como una entrada para otra llamada de `color/pencolor/fillcolor`.

`pencolor(colorstring)` Establece el color del lápiz a *colorstring*, que es una palabra que especifica un color Tk, tales como "red", "yellow", o "#33cc8c".

`pencolor(r, g, b)` Establece el color del lápiz representado como una tupla de *r*, *g*, y *b*. Cada valor *r*, *g*, y *b* debe ser un valor entero en el rango 0..colormode, donde colormode es 1.0 o 255 (ver `colormode()`).

`pencolor(r, g, b)` Establece el color del lápiz al color RGB representado por *r*, *g*, y *b*. Cada valor *r*, *g*, y *b* debe estar en el rango 0..colormode.

Si *turtleshape* es un polígono, la línea del polígono es dibujado con el nuevo color de lápiz elegido.

```
>>> 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()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

turtle.fillcolor(*args)

Return or set the fillcolor.

Se permiten cuatro formatos de entrada:

fillcolor() Devuelve el valor actual de *fillcolor* como una palabra, posiblemente en formato de tupla (ver ejemplo). Puede ser usado como entrada de otra llamada *color/pencolor/fillcolor*.

fillcolor(colorstring) Establece *fillcolor* a *colorstring*, que es un color TK especificado como una palabra (en inglés), tales como «red», «yellow», o «#33cc8c».

fillcolor((r, g, b)) Establece *fillcolor* al color RGB representado por la tupla *r*, *g*, y *b*. Cada uno de los valores *r*, *g*, y *b* debe estar en el rango 0..*colormode*, donde *colormode* es 1.0 o 255 (ver *colormode()*).

fillcolor(r, g, b) Establece *fillcolor* al color RGB representado por *r*, *g*, y *b*. Cada uno de los valores *r*, *g* y *b* debe ser un valor en el rango 0..*colormode*.

Si *turtleshape* es un polígono, el interior de ese polígono es dibujado con el color establecido en *fillcolor*.

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor((50, 193, 143)) # Integers, not floats
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.color(*args)

Retorna o establece *pencolor* (el color del lápiz) y *fillcolor* (el color de relleno).

Se permiten varios formatos de entrada. Usan de 0 a 3 argumentos como se muestra a continuación:

color() Devuelve el valor actual de *pencolor* y el valor actual de *fillcolor* como un par de colores especificados como palabras o tuplas, como devuelven las funciones *pencolor()* y *fillcolor()*.

color(colorstring), color((r, g, b)), color(r, g, b) Entradas como en *pencolor()*, establece al valor dado tanto, *fillcolor* como *pencolor*.

color(colorstring1, colorstring2), color((r1, g1, b1), (r2, g2, b2))

Equivalente a *pencolor(colorstring1)* y *fillcolor(colorstring2)* y análogamente si se usa el otro formato de entrada.

Si *turtleshape* es un polígono, la línea y el interior de ese polígono e dibujado con los nuevos colores que se establecieron.

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

Ver también: Método *Screen* `colormode()`.

Relleno

`turtle.filling()`

Devuelve *fillstate* (True si está lleno, sino False).

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

`turtle.begin_fill()`

Para ser llamada justo antes de dibujar una forma a rellenar.

`turtle.end_fill()`

Rellena la forma dibujada después de última llamada a la función *begin_fill()*.

Superponer o no, regiones de polígonos auto-intersectados o múltiples formas, estas son rellenadas dependiendo de los gráficos del sistema operativo, tipo de superposición y número de superposiciones. Por ejemplo, la flecha de la tortuga de arriba, puede ser toda amarilla o tener algunas regiones blancas.

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

Más controles de dibujo

`turtle.reset()`

Borra el dibujo de la tortuga de la pantalla, centra la tortuga y establece las variables a los valores por defecto.

```
>>> 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()`

Borra el dibujo de la tortuga de la pantalla. No mueve la tortuga. El estado y posición de la tortuga así como los todos los dibujos de las otras tortugas no son afectados.

`turtle.write(arg, move=False, align="left", font=("Arial", 8, "normal"))`

Parámetros

- **arg** – objeto que se escribirá en *TurtleScreen*
- **move** – True/False
- **align** – una de las frases «left», «center» o «right»
- **font** – un trio (nombre de fuente, tamaño de fuente, tipo de fuente)

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)
```

Estado de la Tortuga

Visibilidad

`turtle.hideturtle()`

`turtle.ht()`

Hace invisible a la tortuga, Es una buena idea hacer eso mientras está haciendo dibujos complejos, ya que esconder a la tortuga acelera dibujo de manera observable.

```
>>> turtle.hideturtle()
```

`turtle.showturtle()`

`turtle.st()`

Hace visible la tortuga.

```
>>> turtle.showturtle()
```

`turtle.isvisible()`

Devuelve True si la tortuga se muestra, False si está oculta.

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

Apariencia

`turtle.shape(name=None)`

Parámetros **name** – una cadena de caracteres que es un nombre de forma válido

Establece la forma de la tortuga al *name* que se establece o, si no se establece un nombre, devuelve el nombre actual de su forma. La forma *name* debe existir en el diccionario de formas de *TurtleScreen*. Inicialmente están las siguientes formas poligonales: «arrow», «turtle», «circle», «square», «triangle», «classic». Para aprender como trabajar con estas formas ver los métodos de Screen [register_shape\(\)](#).

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

`turtle.resizemode(rmode=None)`

Parámetros `rmode` – una de las cadenas «*auto*», «*user*», «*noresize*»

Establece *resizemode* a alguno de los valores: «*auto*», «*user*», «*noresize*». Si *mode* no se aporta, devuelve el actual *resizemode*. Distintos *resizemode* tienen los siguientes efectos:

- «*auto*»: adapta la apariencia de la tortuga al correspondiente valor del lápiz.
- «*user*»: adapta la apariencia de la tortuga de acuerdo a los valores de *stretchfactor* y *outlinewidth* (contorno), que se establece con la función *shapeseize()*.
- «*noresize*»: no se adapta la apariencia de la tortuga.

`resizemode("user")` is called by *shapeseize()* 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)`

Parámetros

- **`stretch_wid`** – número positivo
- **`stretch_len`** – número positivo
- **`outline`** – número positivo

Devuelve o establece los atributos del lápiz los factores x/y de estiramiento y contorno. Establece *resizemode* a «*user*» si y solo si *resizemode* está definido como «*user*», la tortuga será verás estirada acorde a sus factores de estiramiento: *stretch_wid* es el factor de estiramiento perpendicular a su orientación, *stretch_len* es su factor de estiramiento en dirección a su orientación, *outline* determina el grosor de contorno de la forma.

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

`turtle.shearfactor(shear=None)`

Parámetros `shear` – número (opcional)

Establece o devuelve el valor actual del estiramiento. Estira la forma de la tortuga de acuerdo a la inclinación del factor de corte, que es la tangente del ángulo de corte. No cambia el rumbo de la tortuga (dirección del movimiento). Si no se da un valor de inclinación: devuelve el factor actual, por ejemplo la tangente del ángulo de inclinación, por el cual se cortan las líneas paralelas al rumbo de la tortuga.

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5, 2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

`turtle.tilt(angle)`

Parámetros `angle` – un número

Rota la forma de la tortuga en *ángulo* desde su ángulo de inclinación actual, pero no cambia el rumbo de la tortuga (dirección del movimiento).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

`turtle.settiltangle(angle)`

Parámetros *angle* – un número

Rota la forma de la tortuga apuntando en la dirección especificada por el *ángulo*, independientemente de su ángulo de dirección actual. No cambia el rumbo de la tortuga (dirección de movimiento).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```

Obsoleto desde la versión 3.1.

`turtle.tiltangle(angle=None)`

Parámetros *angle* – un número (opcional)

Establece o devuelve el ángulo de inclinación actual. Si se otorga un ángulo, rota la forma de la tortuga para apuntar en la dirección del ángulo especificado, independientemente de su actual ángulo de inclinación. No cambia el rumbo de la tortuga (dirección del movimiento). Si no se da el ángulo: devuelve el ángulo de inclinación actual, por ejemplo: el ángulo entre la orientación de la forma de la tortuga y el rumbo de la tortuga (su dirección de movimiento).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

`turtle.shapetransform(t11=None, t12=None, t21=None, t22=None)`

Parámetros

- **t11** – un número (opcional)
- **t12** – un número (opcional)
- **t21** – un número (opcional)
- **t22** – un número (opcional)

Establece o devuelve la matriz de transformación actual de la forma de la tortuga.

If none of the matrix elements are given, return the transformation matrix as a tuple of 4 elements. Otherwise set the given elements and transform the turtleshape according to the matrix consisting of first row t11, t12 and second row t21, t22. The determinant $t11 * t22 - t12 * t21$ must not be zero, otherwise an error is raised. Modify stretchfactor, shearfactor and tiltangle according to the given matrix.


```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

`turtle.get_shapepoly()`

Devuelve el polígono de la forma actual como tupla de pares de coordenadas. Esto puede ser usado para definir una nueva forma o componentes de una forma compuesta.

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

Usando eventos

`turtle.onclick(fun, btn=1, add=None)`

Parámetros

- **fun** – una función con dos argumentos que se invocará con las coordenadas del punto en el que se hizo clic en el lienzo
- **btn** – número del botón del mouse, el valor predeterminado es 1 (botón izquierdo del mouse)
- **add** – True o False – si es True, se agrega un nuevo enlace, de lo contrario reemplazará el enlace anterior

Enlaza *acciones divertidas* a eventos de click en la tortuga. Si la *accion* es None, las acciones asociadas son borradas. Ejemplo para la tortuga anónima, en la forma procedimental:

```
>>> 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)`

Parámetros

- **fun** – una función con dos argumentos que se invocará con las coordenadas del punto en el que se hizo clic en el lienzo
- **btn** – número del botón del mouse, el valor predeterminado es 1 (botón izquierdo del mouse)
- **add** – True o False – si es True, se agrega un nuevo enlace, de lo contrario reemplazará el enlace anterior

Enlaza *acciones divertidas* a eventos del tipo soltar botón del mouse en la tortuga. SI la *acción* es None, las acciones asociadas son borradas.

```
>>> class MyTurtle(Turtle):
...     def glow(self, x, y):
...         self.fillcolor("red")
...     def unglow(self, x, y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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*)

Parámetros

- **fun** – una función con dos argumentos que se invocará con las coordenadas del punto en el que se hizo clic en el lienzo
- **btn** – número del botón del mouse, el valor predeterminado es 1 (botón izquierdo del mouse)
- **add** – True o False – si es True, se agrega un nuevo enlace, de lo contrario reemplazará el enlace anterior

Enlaza *acciones divertidas* a eventos en los movimientos del mouse. Si la *acción* es None, las acciones asociadas son borradas.

Observación: cada secuencia de los eventos de movimiento del mouse en una tortuga es precedida por un evento de click del mouse en esa tortuga.

```
>>> turtle.ondrag(turtle.goto)
```

Subsecuentemente, clickear y arrastrar la Tortuga la moverá a través de la pantalla produciendo dibujos a mano alzada (si el lápiz está abajo).

Métodos especiales de *Turtle*

`turtle.begin_poly` ()

Comienza a grabar los vértices de un polígono. La posición actual de la tortuga es el primer vértice del polígono.

`turtle.end_poly` ()

Deja de grabar los vértices de un polígono. La posición actual de la tortuga es el último vértice del polígono. Esto se conectará con el primer vértice.

`turtle.get_poly` ()

Devuelve el último polígono grabado.

```
>>> 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` ()

Crea y devuelve un clon de la tortuga con la misma posición, dirección y propiedades de la tortuga.

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

`turtle.getturtle` ()

`turtle.getpen` ()

Devuelve el objeto Tortuga en sí. El único uso razonable: es como una función para devolver la «tortuga anónima»:

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

`turtle.getscreen()`

Devuelve el objeto *TurtleScreen* sobre el cual la tortuga está dibujando. Los métodos *TurtleScreen* luego pueden ser llamados para ese objeto.

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

`turtle.setundobuffer(size)`

Parámetros *size* – un entero o `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()`

Devuelve el número de entradas en el buffer para deshacer acciones.

```
>>> while undobufferentries():
...     undo()
```

Formas compuestas

Para usar formas complejas con la tortuga, que consiste en varios polígonos de diferentes colores, deberá usar la clase de ayuda *Shape* explícitamente como se describe debajo:

1. Crear una objeto de forma vacía del tipo *compound*.
2. Agregar todos los componentes deseados a este objeto, usando el método `addcomponent()`.

Por ejemplo:

```
>>> 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. Ahora agregar la forma a la lista de formas de la pantalla y úsela:

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

Nota: La clase *Shape* es usada internamente por el método `register_shape()` en maneras diferentes. El programador deberá lidiar con la clase *Shape* solo cuando use formas compuestas como las que se mostraron arriba!

24.1.4 Métodos de *TurtleScreen/Screen* y sus correspondientes funciones

La mayoría de los ejemplos en esta sección se refieren a la instancia de *TurtleScreen* llamada `screen`.

Control de ventana

`turtle.bgcolor(*args)`

Parámetros `args` – una cadena de color o tres números en el rango 0..*colormode* o una tupla de 3 de esos números

Establece o devuelve el color de fondo de *TurtleScreen*.

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

`turtle.bgpic(picname=None)`

Parámetros `picname` – una cadena, nombre o archivo gif o "nopic", o None

Establece la imagen de fondo o devuelve el nombre de la imagen de fondo actual. Si *picname* es un nombre de archivo, establece la imagen correspondiente como fondo. Si *picname* es "nopic", borra la imagen de fondo, si hay alguna presente. Si *picname* es None, devuelve el nombre de archivo de la imagen de fondo actual.

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

`turtle.clear()`

`turtle.clearscreen()`

Borra todos los dibujos y todas las tortugas de la pantalla de la tortuga. Reinicia la ahora vacía pantalla de la tortuga a su estado inicial: fondo blanco, sin imagen de fondo, sin enlaces a eventos o seguimientos.

Nota: Este método de *TurtleScreen* está disponible como una función global solo bajo el nombre `clearscreen`. La función global `clear` es otra, derivada del método de *Turtle* `clear`.

`turtle.reset()`

`turtle.resetscreen()`

Reinicia todas las tortugas de la pantalla a su estado inicial.

Nota: Este método *TurtleScreen* esta disponible como una función global solo bajo el nombre `resetscreen`. La función global `reset` es otra, derivada del método *Turtle* `reset`.

`turtle.screensize(canvwidth=None, canvheight=None, bg=None)`

Parámetros

- **canvwidth** – entero positivo, nueva anchura del lienzo en pixeles
- **canvheight** – entero positivo, nueva altura del lienzo en pixeles
- **bg** – *colorstrng* o tupla de color, nuevo color de fondo

Si no se dan argumentos, devuelve el actual (ancho y alto del lienzo). Sino redimensiona el lienzo en el que la tortuga está dibujando. No altera la ventana de dibujo. Para ver las partes ocultas del lienzo, use la barra de desplazamiento. Con este método, se pueden hacer visibles aquellas partes de un dibujo que antes estaban fuera del lienzo.

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000, 1500)
>>> screen.screensize()
(2000, 1500)
```

ej. buscar una tortuga que se escapó por error ;-)

`turtle.setworldcoordinates (llx, lly, urx, ury)`

Parámetros

- **llx** – un número, coordenada x de la esquina inferior izquierda del lienzo
- **lly** – un número, coordenada y de la esquina inferior izquierda del lienzo
- **urx** – un número, coordenada x de la esquina superior derecha del lienzo
- **ury** – un número, coordenada z de la esquina superior derecha del lienzo

Configura coordenadas definidas por el usuario y cambia al modo *world* si es necesario. Esto realiza un `screen.reset()`. Si el modo *world* ya está activo, todos los dibujos se re dibujan de acuerdo a las nuevas coordenadas.

ATENCIÓN: en los sistemas de coordenadas definidos por el usuario, los ángulos pueden aparecer distorsionados.

```
>>> 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
```

Control de animación

`turtle.delay (delay=None)`

Parámetros delay – entero positivo

Establece o retorna el *retraso* del dibujo en mili segundos. (Este es aproximadamente, el tiempo de intervalo entre dos actualizaciones consecutivas del lienzo). Mientras más largo sea el retraso, más lenta la animación.

Argumento opcional:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

`turtle.tracer (n=None, delay=None)`

Parámetros

- **n** – entero no negativo
- **delay** – entero no negativo

Activa o desactiva la animación de la tortuga y establece el retraso para la actualización de los dibujos. Si se da n , solo cada n -ésima actualización regular de pantalla se realiza realmente. (Puede usarse para acelerar los dibujos de gráficos complejos). Cuando es llamada sin argumentos, devuelve el valor de n guardado actualmente. El segundo argumento establece el valor de retraso (ver `delay()`).

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

`turtle.update()`

Realiza una actualización de la pantalla de la tortuga. Para ser usada cuando *tracer* está deshabilitada.

Ver también el método *RawTurtle/Turtle* `speed()`.

Usando eventos de pantalla

`turtle.listen(xdummy=None, ydummy=None)`

Establece el foco en la pantalla de la tortuga (para recoger eventos de teclado). Los argumentos *dummy* se proveen en orden de ser capaces de pasar `listen()` a los métodos *onclick*.

`turtle.onkey(fun, key)`

`turtle.onkeyrelease(fun, key)`

Parámetros

- **fun** – una función sin argumentos o `None`
- **key** – una cadena de caracteres: tecla (por ejemplo, «a») o una acción del teclado (por ejemplo, «space»)

Vincula *fun* a un evento de liberación de una tecla. Si *fun* es `None`, los eventos vinculados son removidos. Aclaración: para poder registrar eventos de teclado, *TurtleScreen* tiene que tener el foco. (ver el método `listen()`.)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

`turtle.onkeypress(fun, key=None)`

Parámetros

- **fun** – una función sin argumentos o `None`
- **key** – una cadena de caracteres: tecla (por ejemplo, «a») o una acción del teclado (por ejemplo, «space»)

Vincula *fun* a un evento de pulsado de una tecla, o a cualquier evento de pulsado de tecla si no se da una. Aclaración: para poder registrar eventos de teclado, *TurtleScreen* tiene que tener el foco. (ver el método `listen()`.)

```
>>> def f():
...     fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

`turtle.onclick(fun, btn=1, add=None)`

`turtle.onscreenclick(fun, btn=1, add=None)`

Parámetros

- **fun** – una función con dos argumentos que se invocará con las coordenadas del punto en el que se hizo clic en el lienzo
- **btn** – número del botón del mouse, el valor predeterminado es 1 (botón izquierdo del mouse)
- **add** – True o False – si es True, se agrega un nuevo enlace, de lo contrario reemplazará el enlace anterior

Vincula *fun* a eventos de click del mouse en esta pantalla. Si *fun* es None, los vínculos existentes son removidos.

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
```

Nota: Este método *TurtleScreen* está disponible como una función global solo bajo el nombre *onscreenclick*. La función global *onclick* es otra derivada del método Turtle *onclick*.

`turtle.ontimer(fun, t=0)`

Parámetros

- **fun** – una función sin argumentos
- **t** – un número ≥ 0

Instala un temporizador que llama a *fun* cada *t* milisegundos.

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()    ### makes the turtle march around
>>> running = False
```

`turtle.mainloop()`

`turtle.done()`

Comienza un bucle de evento - llamando a la función *mainloop* del *Tkinter*. Debe ser la última declaración en un programa gráfico de turtle. No debe ser usado si algún script es corrido dentro del IDLE en modo -n (Sin subprocesso) - para uso interactivo de gráficos turtle.:

```
>>> screen.mainloop()
```

Métodos de entrada

`turtle.textinput(title, prompt)`

Parámetros

- **title** – cadena de caracteres
- **prompt** – cadena de caracteres

Abre una ventana de diálogo para ingresar una cadena de caracteres. El parámetro *title* es el título de la ventana de diálogo, *prompt* es un texto que usualmente describe que información se debe ingresar. Devuelve la cadena ingresada. Si el diálogo es cancelado, devuelve *None*.

```
>>> screen.textinput("NIM", "Name of first player:")
```

`turtle.numinput` (*title*, *prompt*, *default=None*, *minval=None*, *maxval=None*)

Parámetros

- **title** – cadena de caracteres
- **prompt** – cadena de caracteres
- **default** – número (opcional)
- **minval** – número (opcional)
- **maxval** – número (opcional)

Abre una ventana de diálogo para el ingreso de un número. *title* es el título de la ventana de diálogo, *prompt* es un texto que usualmente describe qué información numérica ingresar. *default*: valor por defecto, *minval*: mínimo valor aceptado, *maxval*: máximo valor aceptado. Si se aportan estos parámetros, el número a ingresar debe estar en el rango *minval*..*maxval*. Si no, se da una pista y el diálogo permanece abierto para su corrección. Devuelve el número ingresado. Si el diálogo es cancelado, devuelve `None`.

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

Configuración y métodos especiales

`turtle.mode` (*mode=None*)

Parámetros **mode** – una de las cadenas «*standard*», «*logo*» o «*world*»

Establece el mode de la tortuga («*standard*», «*logo*» o «*world*») y realiza un reinicio. Si no se da «*mode*», retorna el modo actual.

El modo «*standard*» es compatible con el antiguo *turtle*. El modo «*logo*» es compatible con la mayoría de los gráficos de tortuga Logo. El modo «*world*» usa coordenadas de mundo definidas por el usuario. **Atención:** en este modo los ángulos aparecen distorsionados si la relación de unidad *x*/*y* no es igual a 1.

Modo	Rumbo inicial de la tortuga	ángulos positivos
« <i>standard</i> »	hacia la derecha (este)	sentido antihorario
« <i>logo</i> »	hacia arriba (norte)	sentido horario

```
>>> mode("logo")    # resets turtle heading to north
>>> mode()
'logo'
```

`turtle.colormode` (*cmode=None*)

Parámetros **cmode** – uno de los valores 1.0 o 255

Devuelve el *colormode* o lo establece a 1.0 o 255. Subsecuentemente, los valores triples de color *r*, *g*, *b* tienen que estar en el rango 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()`

Devuelve el lienzo de este *TurtleScreen*. Útil para conocedores que saben que hace con un *TKinter Canvas*.

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

`turtle.getshapes()`

Devuelve la lista de nombres de todas las formas de la tortuga actualmente disponibles.

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

`turtle.register_shape(name, shape=None)`

`turtle.addshape(name, shape=None)`

Hay tres formas distintas de llamar a esta función:

- (1) *name* es el nombre de un archivo gif y *shape* es *None*: instala la imagen correspondiente.

```
>>> screen.register_shape("turtle.gif")
```

Nota: Las imágenes de tipo *shapes* no rotan cuando la tortuga gira, así que ellas ¡no muestran el rumbo de la tortuga!

- (2) *name* es una cadena de caracteres arbitraria y *shape* es una tupla de pares de coordenadas: Instala la forma poligonal correspondiente.

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

- (3) *name* es una cadena de caracteres arbitraria y *shape* es un objeto *Shape* (compuesto): Instala la correspondiente forma compuesta.

Agregar una forma de tortuga a la lista de formas de *TurtleScreen*. Solo las formas registradas de esta manera pueden ser usadas invocando el comando `shape(shapename)`.

`turtle.turtles()`

Devuelve la lista de tortugas en la pantalla.

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

`turtle.window_height()`

Devuelve la altura de la ventana de la tortuga.

```
>>> screen.window_height()
480
```

`turtle.window_width()`

Devuelve el ancho de la ventana de la tortuga.

```
>>> screen.window_width()
640
```

Métodos específicos de *Screen*, no heredados de *TurtleScreen*

`turtle.bye()`

Apaga la ventana gráfica de la tortuga.

`turtle.exitonclick()`

Bind `bye()` method to mouse clicks on the Screen.

Si el valor «*using_IDLE*» en la configuración del diccionario es `False` (valor por defecto), también ingresa *mainloop*. Observaciones: si se usa la *IDLE* con el modificador (no subproceso) `-n`, este valor debe establecerse a `True` en `turtle.cfg`. En este caso el propio *mainloop* de la *IDLE* está activa también para script cliente.

`turtle.setup` (*width*=_CFG["width"], *height*=_CFG["height"], *startx*=_CFG["leftright"],
starty=_CFG["topbottom"])

Establece el tamaño y la posición de la ventana principal. Los valores por defecto de los argumentos son guardados en el diccionario de configuración y puede ser cambiado a través del archivo `turtle.cfg`.

Parámetros

- **width** – si es un entero, el tamaño en pixeles, si es un número flotante, una fracción de la pantalla; el valor por defecto es 50% de la pantalla
- **height** – si es un entero, la altura en pixeles, si es un número flotante, una fracción de la pantalla; el valor por defecto es 75% de la pantalla
- **startx** – si es positivo, punto de inicio en pixeles desde la esquina izquierda de la pantalla, si es negativo desde la esquina derecha de la pantalla, si es `None`, centra la ventana horizontalmente
- **starty** – si es positivo, punto de inicio en pixeles desde la parte superior de la pantalla, si es negativo desde la parte inferior de la pantalla, si es `None`, centra la ventana verticalmente

```
>>> 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*)

Parámetros *titlestring* – una cadena de caracteres que se muestra en la barra de título de la ventana gráfica de la tortuga

Establece el título de la ventana de la tortuga a *titlestring*.

```
>>> screen.title("Welcome to the turtle zoo!")
```

24.1.5 Clases públicas

class `turtle.RawTurtle` (*canvas*)

class `turtle.RawPen` (*canvas*)

Parámetros *canvas* – una clase `tkinter.Canvas`, una *ScrolledCanvas* o una clase *TurtleScreen*

Crea una tortuga. La tortuga tiene todos los métodos descritos anteriormente como «métodos de «*Turtle*/*RawTurtle*».

class `turtle.Turtle`

Subclase de *RawTurtle*, tiene la misma interface pero dibuja sobre una objeto por defecto *Screen* creado automáticamente cuando es necesario por primera vez.

class `turtle.TurtleScreen` (*cv*)

Parámetros *cv* – un `tkinter.Canvas`

Provee métodos orientados a la pantalla como `setbg()` etc. descritos anteriormente.

class `turtle.Screen`

Subclase de *TurtleScreen*, con *cuatro métodos agregados*.

class `turtle.ScrolledCanvas` (*master*)

Parámetros *master* – algunos *widgets Tkinter* para contener el *ScrollCanvas*, por ejemplo un lienzo *Tkinter* con barras de desplazamiento agregadas

Usado por la clase *Screen*, que proporciona automáticamente un *ScrolledCanvas* como espacio de trabajo para las tortugas.

class `turtle.Shape` (*type_*, *data*)

Parámetros *type_* – una de las cadenas de caracteres «*polygon*», «*image*», «*compound*»

Estructura de datos que modela las formas. El par (*type_*, *data*)‘ debe seguir estas especificaciones:

<i>type_</i>	<i>data</i>
« <i>polygon</i> »	una tupla para el polígono, por ejemplo: una tupla de par de coordenadas
« <i>image</i> »	una imagen (en esta forma solo se usa ¡internamente!)
« <i>compound</i> »	None (una forma compuesta tiene que ser construida usando el método <i>addcomponent()</i>)

addcomponent (*poly*, *fill*, *outline=None*)

Parámetros

- **poly** – un polígono, por ejemplo una tupla de pares de números
- **fill** – un color con el que el polígono será llenado
- **outline** – un color con el que se delineará el polígono (se de ingresa)

Ejemplo:

```
>>> 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()
```

Ver *Formas compuestas*.

class `turtle.Vec2D` (*x*, *y*)

Una clase de vectores bidimensionales, usado como clase de ayuda para implementar gráficos de tortuga. Puede ser útil para los programas de gráficos de tortugas también. Derivado de la tupla, ¡por lo que un vector es una tupla!

Proporciona (para *a*, *b* vectores, *k* números)

- *a* + *b* suma de vectores
- *a* - *b* resta de vectores
- *a* * *b* producto interno
- *k* * *a* y *a* * *k* multiplicación con escalar
- *abs(a)* valor absoluto de *a*
- *a.rotate(angle)* rotación

24.1.6 Ayuda y configuración

Cómo usar la ayuda

Los métodos públicos de las clases `Screen` y `Turtle` están ampliamente documentados a través de cadenas de documentación. Por lo tanto, estos se pueden usar como ayuda en línea a través de las instalaciones de ayuda de Python:

- Cuando se usa IDLE, la información sobre herramientas muestra las firmas y las primeras líneas de las cadenas de documentos de las llamadas de función/método escritas.
- Llamar a `help()` en métodos o funciones muestra los 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()
```

- Los docstrings de las funciones que se derivan de los métodos tienen una forma modificada:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    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)
```

(continué en la próxima página)

(proviene de la página anterior)

```

Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()

```

Estos docstrings modificados se crean automáticamente junto con las definiciones de función que se derivan de los métodos en el momento de la importación.

Traducción de cadenas de documentos a diferentes idiomas

Existe una utilidad para crear un diccionario cuyas claves son los nombres de los métodos y cuyos valores son los docstrings de los métodos públicos de las clases `Screen` y `Turtle`.

`turtle.write_docstringdict(filename="turtle_docstringdict")`

Parámetros `filename` – una cadena de caracteres, utilizada como nombre de archivo

Crea y escribe un diccionario-docstring en un script de Python con el nombre de archivo dado. Esta función tiene que ser llamada explícitamente (no es utilizada por las clases de gráficos de tortugas). El diccionario de docstrings se escribirá en el script de Python `filename.py`. Está destinado a servir como plantilla para la traducción de las cadenas de documentos a diferentes idiomas.

Si usted (o sus estudiantes) desea utilizar `turtle` con ayuda en línea en su idioma nativo, debe traducir los docstrings y guardar el archivo resultante como, por ejemplo, `turtle_docstringdict_german.py`.

Si tiene una entrada adecuada en su archivo `turtle.cfg`, este diccionario se leerá en el momento de la importación y reemplazará los docstrings originales en inglés.

En el momento de escribir este artículo, existen diccionarios de docstrings en alemán e italiano. (Solicitudes por favor a glingl@aon.at.)

Cómo configurar Screen and Turtles

La configuración predeterminada incorporada imita la apariencia y el comportamiento del antiguo módulo de tortuga para mantener la mejor compatibilidad posible con él.

Si desea utilizar una configuración diferente que refleje mejor las características de este módulo o que se adapte mejor a sus necesidades, por ejemplo, para su uso en un aula, puede preparar un archivo de configuración `turtle.cfg` que se leerá en el momento de la importación y modificará la configuración de acuerdo con su configuración.

La configuración incorporada correspondería al siguiente `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

```

(continué en la próxima página)

(proviene de la página anterior)

```

fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False

```

Breve explicación de las entradas seleccionadas:

- Las primeras cuatro líneas corresponden a los argumentos del método `Screen.setup()`.
- Las líneas 5 y 6 corresponden a los argumentos del método `Screen.screensize()`.
- *shape* puede ser cualquiera de las formas integradas, por ejemplo: `arrow`, `turtle`, etc. Para obtener más información, pruebe con `help(shape)`.
- Si no desea usar color de relleno (es decir, hacer que la tortuga sea transparente), debe escribir `fillcolor = ""` (pero todas las cadenas no vacías no deben tener comillas en el archivo `cfg`).
- Si desea reflejar el estado de la tortuga, debe usar `resizemode = auto`.
- Si establece, por ejemplo, `language = italian` el `docstringdict_turtle_docstringdict_italian.py` se cargará en el momento de la importación (si está presente en la ruta de importación, por ejemplo, en el mismo directorio que `turtle`).
- Las entradas `exampleturtle` y `examplescreen` definen los nombres de estos objetos a medida que aparecen en las cadenas de documentos. La transformación de método-docstrings en función-docstrings eliminará estos nombres de las docstrings.
- `using_IDLE`: establezca esto en `True` si trabaja regularmente con IDLE y su interruptor `-n` («sin subproceso»). Esto evitará que `exitonclick()` ingrese al bucle principal.

Puede haber un archivo `turtle.cfg` en el directorio donde se almacena `turtle` y uno adicional en el directorio de trabajo actual. Este último anulará la configuración del primero.

El directorio `Lib/turtledemo` contiene un archivo `turtle.cfg`. Puede estudiarlo como un ejemplo y ver sus efectos al ejecutar las demostraciones (preferiblemente no desde el visor de demostraciones).

24.1.7 turtledemo — Scripts de demostración

El paquete `turtledemo` incluye un conjunto de scripts de demostración. Estos scripts se pueden ejecutar y visualizar utilizando el visor de demostración suministrado de la siguiente manera:

```
python -m turtledemo
```

Alternativamente, puede ejecutar los scripts de demostración individualmente. Por ejemplo,

```
python -m turtledemo.bytedesign
```

El directorio del paquete `turtledemo` contiene:

- Un visor de demostración `__main__.py` que se puede utilizar para ver el código fuente de los scripts y ejecutarlos al mismo tiempo.
- Múltiples scripts que demuestran diferentes características del módulo `turtle`. Se puede acceder a los ejemplos a través del menú de ejemplos. También se pueden ejecutar de forma independiente.
- Un archivo `turtle.cfg` que sirve como ejemplo de cómo escribir y usar dichos archivos.

Los scripts de demostración son:

Nombre	Descripción	Características
bytedesign	patrón de gráficos de tortuga clásica compleja	<code>tracer()</code> , retrasar (<i>delay</i>), <code>update()</code>
caos	gráficos dinámicos de Verhulst, muestra que los cálculos de la computadora pueden generar resultados a veces contra las expectativas del sentido común	coordenadas mundiales
reloj	reloj analógico que muestra la hora de su computadora	tortugas como manecillas de reloj, temporizador
colormixer	experimento con r, g, b	<code>ondrag()</code>
bosque	3 árboles de ancho primero	aleatorización
fractalcurves	Curvas Hilbert & Koch	recursión
lindenmayer	etnomatemáticas (kolams indios)	Sistema-L
minimal_hanoi	Torres de Hanoi	Tortugas rectangulares como discos de Hanoi (<i>shape</i> , tamaño de forma)
nim	juega el clásico juego de nim con tres montones de palos contra la computadora.	tortugas como nimsticks, impulsado por eventos (mouse, teclado)
pintar	programa de dibujo super minimalista	<code>onclick()</code>
paz	elemental	turtle: apariencia y animación
penrose	embaldosado aperiódico con cometas y dardos	<code>stamp()</code>
planet_and_moon	simulación de sistema gravitacional	formas compuestas, <code>Vec2D</code>
round_dance	tortugas bailarinas que giran por parejas en dirección opuesta	formas compuestas, clonar tamaño de forma, <i>tilt</i> , <code>get_shapepoly</code> , <code>update</code>
sorting_animate	demonstración visual de diferentes métodos de ordenamiento	alineación simple, aleatorización
árbol	un primer árbol de amplitud (gráfico, usando generadores)	<code>clone()</code>
two_canvases	diseño simple	tortugas en dos lienzos (<i>two_canvases</i>)
wikipedia	un patrón del artículo de wikipedia sobre gráficos de tortuga (<i>turtle</i>)	<code>clone()</code> , <code>undo()</code>
yinyang	otro ejemplo elemental	<code>circle()</code>

¡Diviértete!

24.1.8 Cambios desde Python 2.6

- Los métodos `Turtle.tracer()`, `Turtle.window_width()` y `Turtle.window_height()` han sido eliminados. Los métodos con estos nombres y funciones ahora están disponibles solo como métodos de `Screen`. Las funciones derivadas de estos permanecen disponibles. (De hecho, ya en Python 2.6 estos métodos eran simplemente duplicaciones de los métodos correspondientes `TurtleScreen/Screen`).
- El método `Turtle.fill()` ha sido eliminado. El comportamiento de `begin_fill()` y `end_fill()` ha cambiado ligeramente: ahora cada proceso de llenado debe completarse con una llamada `end_fill()`.
- Se ha añadido un método `Turtle.filling()`. Retorna un valor booleano: `True` si hay un proceso de llenado en curso, `False` en caso contrario. Este comportamiento corresponde a una llamada `fill()` sin argumentos en Python 2.6.

24.1.9 Cambios desde Python 3.0

- Se han añadido los métodos `Turtle.shearfactor()`, `Turtle.shapetransform()` y `Turtle.get_shapepoly()`. Por lo tanto, ahora está disponible la gama completa de transformaciones lineales regulares para transformar formas de tortugas. `Turtle.tiltangle()` se ha mejorado en funcionalidad: ahora se puede usar para obtener o establecer el tiltangle. `Turtle.settiltangle()` ha quedado obsoleto.
- El método `Screen.onkeypress()` se ha agregado como complemento a `Screen.onkey()` que, de hecho, une las acciones al evento `keyrelease`. En consecuencia, este último tiene un alias: `Screen.onkeyrelease()`.
- Se ha añadido el método `Screen.mainloop()`. Entonces, cuando se trabaja solo con objetos `Screen` y `Turtle`, ya no se debe importar adicionalmente `mainloop()`.
- Se han añadido dos métodos de entrada `Screen.textinput()` y `Screen.numinput()`. Estos cuadros de diálogo de entrada emergentes y retornan cadenas y números respectivamente.
- Se han agregado dos scripts de ejemplo `tdemo_nim.py` y `tdemo_round_dance.py` al directorio `Lib/turtledemo`.

24.2 cmd — Soporte para intérpretes orientados a línea de comandos

Código fuente: [Lib/cmd.py](#)

La clase `Cmd` proporciona un *framework* simple para escribir intérpretes de comandos orientados a línea. A menudo son útiles para agentes de prueba, herramientas administrativas y prototipos que luego se incluirán en una interfaz más sofisticada.

class `cmd.Cmd` (*completekey='tab', stdin=None, stdout=None*)

Una instancia `Cmd` o subclase de la instancia es un *framework* intérprete orientado a líneas. No hay una buena razón para crear instancias `Cmd`; mas bien, es útil como una super clase de una clase intérprete que usted define con el fin de heredar métodos `Cmd` y encapsular métodos de acción.

El argumento opcional *completekey* es el nombre *readline* de una llave de finalización; por defecto es `Tab`. Si *completekey* no es `None` y *readline* está disponible, el comando de finalización es hecho automáticamente.

Los argumentos opcionales *stdin* y *stdout* especifican la entrada y salida de los objetos archivo que la instancia `Cmd` o la instancia subclase usará para la entrada y salida. Si no está especificado, se predeterminarán a `sys.stdin` y `sys.stdout`.

Si desea un *stdin* dado a ser usado, asegúrese de establecer las instancias atributo `use_rawinput` a `False`, de lo contrario *stdin* será ignorado.

24.2.1 Objetos Cmd

Una instancia `Cmd` tiene los siguientes métodos:

`Cmd.cmdloop` (*intro=None*)

Emita repetidamente una solicitud, acepte la entrada, analice un prefijo inicial de la entrada recibida y envíe a los métodos de acción, pasándoles el resto de la línea como argumento.

El argumento opcional es un *banner* o cadena de caracteres introductoria que se tramitará antes del primer mensaje (esto anula el atributo de clase *intro*).

Si el módulo *readline* es cargado, la entrada heredará automáticamente **bash**-como edición historia-lista (e.g. `Control-P` retorna al último comando, `Control-N` adelanta al siguiente, `Control-F` mueve el

cursor a la derecha no destructivamente, `Control-B` mueve el cursor a la izquierda no destructivamente, etc.).

Un fin-de-archivo en la entrada es retornado como cadena de caracteres `'EOF'`.

Una instancia del intérprete reconocerá un nombre de comando `foo` si y solo si tiene un método `do_foo()`. Como un caso especial, una línea comenzando con el carácter `'?'` es enviada al método `do_help()`. Como otro caso especial, una línea comenzando con el carácter `'!'` es enviada al método `do_shell()` (Si dicho método está definido).

Este método retornará cuando el método `postcmd()` retorna un valor verdadero. El argumento `stop` a `postcmd()` es el valor de retorno de los comandos correspondientes al método `do_*`.

Si completar está habilitado, comandos de completar se realizarán automáticamente y la finalización de los comandos `args` es realizada llamando `complete_foo()` con los argumentos `text`, `line`, `begidx`, y `endidx`. `text` es el prefijo de la cadena de caracteres que estamos intentando emparejar: todos los emparejamientos retornados deben comenzar con él. `line` es la línea de entrada actual con el espacio en blanco inicial eliminado, `begidx` y `endidx` son los índices iniciales y finales del texto prefijo, que podrían usarse para proporcionar un completar diferente dependiendo de la posición en la que se encuentre el argumento.

Todas las subclases de `Cmd` heredan un `do_help()` predefinido. Este método, llamado con el argumento `'bar'`, invoca el método correspondiente `help_bar()`, y si eso no está presente, imprime el *docstring* de `do_bar()`, si está disponible. Sin argumentos, `do_help()` enumera todos los temas de ayuda disponibles (Es decir, todos los comandos con los métodos correspondientes `help_*` o los comandos que tienen *docstrings*), y también enumera cualquier comando no documentado.

`Cmd.onecmd(str)`

Interprete el argumento como si hubiera sido escrito en respuesta a la solicitud. Esto puede ser anulado, pero normalmente no debería ser necesario; vea los métodos `precmd()` y `postcmd()` para enlaces de ejecución útiles. El valor de retorno es una bandera que indica si la interpretación de los comandos por parte del intérprete debe detenerse. Si hay un método `do_*` para el comando `str`, se retorna el valor de retorno de ese método; de lo contrario, se retorna el valor de retorno del método `default()`.

`Cmd.emptyline()`

Método llamado cuando se ingresa una línea vacía en respuesta a la solicitud. Si este método no se anula, repite el último comando no vacío ingresado.

`Cmd.default(line)`

Método llamado en una línea de entrada cuando no se reconoce el prefijo del comando. Si este método no se anula, imprime un mensaje de error y retorna.

`Cmd.completedefault(text, line, begidx, endidx)`

Método llamado para completar una línea de entrada cuando no hay un comando específico, método `complete_*` está disponible. Por defecto, retorna una lista vacía.

`Cmd.precmd(line)`

Método de enlace ejecutado justo antes de que se interprete la línea de comando `line`, pero después de que se genere y emita la solicitud de entrada. Este método es un trozo en `Cmd`; existe para ser anulado por subclases. El valor de retorno es utilizado como el comando que se ejecutará mediante el método `onecmd()`; la implementación `precmd()` puede reescribir el comando o simplemente retornar `line` sin cambios.

`Cmd.postcmd(stop, line)`

Método de enlace ejecutado justo después de que finaliza un despacho de comando. Este método es un trozo en `Cmd`; existe para ser anulado por subclases. `line` es la línea de comando que se ejecutó, y `stop` es una bandera que indica si la ejecución finalizará después de la llamada a `postcmd()`; este será el valor de retorno del método `onecmd()`. El valor de retorno de este método será usado como el nuevo valor para la bandera interna que corresponde a `stop`; retornando falso hará que la interpretación continúe.

`Cmd.preloop()`

Método de enlace ejecutado una vez cuando `cmdloop()` es llamado. Este método es un trozo en `Cmd`; existe para ser anulado por subclases.

`Cmd.postloop()`

Método de enlace ejecutado una vez cuando `cmdloop()` está a punto de retornar. Este método es un trozo en `Cmd`; existe para ser anulado por subclases.

Instancias de subclases `Cmd` tienen algunas variables de instancia públicas:

`Cmd.prompt`

El aviso emitido para solicitar entradas.

`Cmd.identchars`

La cadena de caracteres aceptada para el prefijo del comando.

`Cmd.lastcmd`

El último prefijo de comando no vacío visto.

`Cmd.cmdqueue`

Una lista de líneas de entrada puestas en cola. La lista `cmdqueue` es verificada en `cmdloop()` cuando una nueva entrada es necesitada; Si es no vacía, sus elementos serán procesados en orden, como si se ingresara en la solicitud.

`Cmd.intro`

Una cadena para emitir como introducción o *banner*. Puede ser anulado dando un argumento al método `cmdloop()`.

`Cmd.doc_header`

El encabezado a tramitar si la salida de ayuda tiene una sección para comandos documentados.

`Cmd.misc_header`

El encabezado a tramitar si la salida de ayuda tiene una sección para temas de ayuda misceláneos (Es decir, hay métodos `help_*()` sin los métodos correspondientes `do_*()`).

`Cmd.undoc_header`

El encabezado a tramitar si la salida de ayuda tiene una sección para comandos no documentados (Es decir, hay métodos `do_*()` sin los métodos correspondientes `help_*()`).

`Cmd.ruler`

El carácter utilizado para dibujar líneas separadoras debajo de los encabezados mensajes-ayuda. Si está vacío, no se dibuja una línea regla. El valor predeterminado es `'= '`.

`Cmd.use_rawinput`

Una bandera, por defecto verdadera. Si es verdadera, `cmdloop()` usa `input()` para mostrar un mensaje y leer el siguiente comando; si es falsa, `sys.stdout.write()` y `sys.stdin.readline()` son usados. (Esto significa que importando `readline`, en sistemas que lo soportan, el intérprete soportará automáticamente **Emacs**-basados y comandos-historial de teclado.)

24.2.2 Ejemplo Cmd

El módulo `cmd` es principalmente útil para construir shells personalizados que permiten al usuario trabajar con un programa de forma interactiva.

Esta sección presenta un ejemplo simple de cómo construir un *shell* alrededor de algunos de los comandos en el módulo `turtle`.

Comandos `turtle` básicos como `forward()` son añadidos a la subclase `Cmd` con un método llamado `do_forward()`. El argumento es convertido a un número y enviado al módulo `turtle`. El *docstring* se utiliza en la utilidad de ayuda proporcionada por el *shell*.

El ejemplo también incluye un registro básico y facilidad de reproducción implementado con el método `precmd()` el cuál es el responsable de convertir la entrada a minúscula y escribir los comandos en un archivo. El método `do_playback()` lee el archivo y añade los comandos grabados al `cmdqueue` para un *playback*: inmediato:

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
    intro = 'Welcome to the turtle shell.  Type help or ? to list commands.\n'
    prompt = '(turtle) '
    file = None
```

(continué en la próxima página)

(proviene de la página anterior)

```

# ----- basic turtle commands -----
def do_forward(self, arg):
    'Move the turtle forward by the specified distance: FORWARD 10'
    forward(*parse(arg))
def do_right(self, arg):
    'Turn turtle right by given number of degrees: RIGHT 20'
    right(*parse(arg))
def do_left(self, arg):
    'Turn turtle left by given number of degrees: LEFT 90'
    left(*parse(arg))
def do_goto(self, arg):
    'Move turtle to an absolute position with changing orientation. GOTO 100,
↪200'
    goto(*parse(arg))
def do_home(self, arg):
    'Return turtle to the home position: HOME'
    home()
def do_circle(self, arg):
    'Draw circle with given radius an options extent and steps: CIRCLE 50'
    circle(*parse(arg))
def do_position(self, arg):
    'Print the current turtle position: POSITION'
    print('Current position is %d %d\n' % position())
def do_heading(self, arg):
    'Print the current turtle heading in degrees: HEADING'
    print('Current heading is %d\n' % (heading(),))
def do_color(self, arg):
    'Set the color: COLOR BLUE'
    color(arg.lower())
def do_undo(self, arg):
    'Undo (repeatedly) the last turtle action(s): UNDO'
def do_reset(self, arg):
    'Clear the screen and return turtle to center: RESET'
    reset()
def do_bye(self, arg):
    'Stop recording, close the turtle window, and exit: BYE'
    print('Thank you for using Turtle')
    self.close()
    bye()
    return True

# ----- record and playback -----
def do_record(self, arg):
    'Save future commands to filename: RECORD rose.cmd'
    self.file = open(arg, 'w')
def do_playback(self, arg):
    'Playback commands from a file: PLAYBACK rose.cmd'
    self.close()
    with open(arg) as f:
        self.cmdqueue.extend(f.read().splitlines())
def precmd(self, line):
    line = line.lower()
    if self.file and 'playback' not in line:
        print(line, file=self.file)
    return line
def close(self):
    if self.file:
        self.file.close()
        self.file = None

```

(continué en la próxima página)

(proviene de la página anterior)

```
def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()
```

Aquí hay una sesión de muestra con la *turtle shell* mostrando las funciones de ayuda, de ayuda, usando líneas en blanco para repetir comandos, un registro simple y facilidad de reproducción:

```
Welcome to the turtle shell.  Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
=====
bye      color      goto      home      playback  record    right
circle  forward  heading  left      position  reset     undo

(turtle) help forward
Move the turtle forward by the specified distance:  FORWARD 10
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0
```

(continué en la próxima página)

(proviene de la página anterior)

```
Current heading is 0

Current heading is 180

(turtle) bye
Thank you for using Turtle
```

24.3 shlex — Análisis léxico simple

Código fuente: [Lib/shlex.py](#)

La clase `shlex` facilita la escritura de analizadores léxicos para sintaxis simples que se parecen al intérprete de comandos de Unix. Esto será a menudo útil para escribir pequeños lenguajes, (por ejemplo, en archivos de control para aplicaciones Python) o para analizar cadenas de texto citadas.

El módulo `shlex` define las siguientes funciones:

`shlex.split(s, comments=False, posix=True)`

Divide la cadena de caracteres `s` usando una sintaxis similar a la de un intérprete de comandos. Si `comments` es `False` (por defecto), el análisis de los comentarios en la cadena de caracteres dada sera deshabilitada (estableciendo el atributo `commenters` de la instancia `shlex` a la cadena de caracteres vacía). Esta función trabaja en modo POSIX por defecto, pero utiliza el modo non-POSIX si el argumento `posix` es falso.

Nota: Como la función `split()` inicia una instancia `shlex`, al pasar `None` por `s` se leerá la cadena de caracteres para separarse de la entrada estándar.

`shlex.join(split_command)`

Concatena los tokens de la lista `split_command` y retorna una cadena. Esta función es la inversa de `split()`.

```
>>> from shlex import join
>>> print(join(['echo', '-n', 'Multiple words']))
echo -n 'Multiple words'
```

El valor retornado se escapa del intérprete de comandos para protegerlo contra vulnerabilidades de inyección (consulte `quote()`).

Nuevo en la versión 3.8.

`shlex.quote(s)`

Retorna una versión con escape del intérprete de comandos de la cadena `s`. El valor retornado es una cadena que se puede usar de forma segura como un token en un intérprete de línea de comandos, para los casos en los que no se puede usar una lista.

Este idioma sería inseguro:

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command) # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

`quote()` te permite tapar el agujero de seguridad:

```
>>> from shlex import quote
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l '''somefile; rm -rf ~'''''
```

La cita es compatible con los intérpretes de comandos UNIX y con `split()`:

```
>>> from shlex import split
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

Nuevo en la versión 3.3.

El módulo `shlex` define las siguientes clases:

class `shlex.shlex` (*instream=None, infile=None, posix=False, punctuation_chars=False*)

Una instancia o instancia de la subclase `shlex` es un objeto de analizador léxico. El argumento de inicialización, si está presente, especifica de dónde leer caracteres. Debe ser un objeto similar a un archivo/stream con los métodos `read()` y `readline()` o una cadena de caracteres. Si no se da ningún argumento, la entrada se tomará de `sys.stdin`. El segundo argumento opcional es una cadena de nombre de archivo, que establece el valor inicial del atributo `infile`. Si el argumento `instream` es omitido o es igual a `sys.stdin`, este segundo argumento tiene como valor predeterminado «stdin». El argumento `posix` define el modo operativo: cuando `posix` no es `true` (predeterminado), la instancia `shlex` funcionará en modo de compatibilidad. Cuando se opera en modo POSIX, `shlex` intentará estar lo más cerca posible de las reglas de análisis de el intérprete de comandos POSIX. El argumento `punctuation_chars` proporciona una manera de hacer que el comportamiento sea aún más cercano a la forma en que se analizan los intérpretes de comandos reales. Esto puede tomar una serie de valores: el valor predeterminado, `False`, conserva el comportamiento visto en Python 3.5 y versiones anteriores. Si se establece en `True`, se cambia el análisis de los caracteres `() ; <> | &`: cualquier ejecución de estos caracteres (caracteres considerados de puntuación) se retorna como un único token. Si se establece en una cadena de caracteres no vacía, esos caracteres se utilizarán como caracteres de puntuación. Los caracteres del atributo `wordchars` que aparecen en `punctuation_chars` se eliminarán de `wordchars`. Consulte [Compatibilidad mejorada con intérprete de comandos](#) para obtener más información. `punctuation_chars` solo se puede establecer en la creación de la instancia `shlex` y no se puede modificar más adelante.

Distinto en la versión 3.6: Se añadió el parámetro `punctuation_chars`.

Ver también:

Módulo `configparser` Analizador de archivos de configuración similares a los archivos `.ini` de Windows.

24.3.1 objetos `shlex`

Una instancia `shlex` tiene los siguientes métodos:

`shlex.get_token()`

Retorna un token. Si los tokens han sido apilados usando `push_token()`, saca una ficha de la pila. De lo contrario, lee uno de la secuencia de entrada. Si la lectura encuentra un fin de archivo inmediato, `eof` se retorna (la cadena de caracteres vacía (`' '`) en modo no-POSIX, y `None` en modo POSIX).

`shlex.push_token(str)`

Coloca el argumento en la pila de tokens.

`shlex.read_token()`

Lee un token sin procesar. Ignora la pila de retroceso y no interpreta las peticiones de la fuente. (Este no es normalmente un punto de entrada útil, y está documentado aquí sólo para completarlo.)

`shlex.sourcehook(filename)`

Cuando `shlex` detecta una petición de fuente (ver `source` abajo) este método recibe el siguiente token

como argumento, y se espera que retorne una tupla que consista en un nombre de archivo y un archivo abierto como objeto.

Normalmente, este método primero elimina las comillas del argumento. Si el resultado es un nombre de ruta absoluto, o no había ninguna solicitud de origen anterior en vigor, o el origen anterior era una secuencia (como `sys.stdin`), el resultado se deja solo. De lo contrario, si el resultado es un nombre de ruta relativo, la parte del directorio del nombre del archivo se pone inmediatamente antes en la pila de inclusión de origen (este comportamiento es similar a la forma en que el preprocesador de C controla `#include "file.h"`).

El resultado de las manipulaciones se trata como un nombre de archivo y se retorna como el primer componente de la tupla, con `open()` llamado en él para producir el segundo componente. (Nota: esto es lo contrario del orden de los argumentos en la inicialización de instancia!)

Este enlace se expone para que pueda usarlo para implementar rutas de búsqueda de directorios, adición de extensiones de archivo y otros trucos de espacios de nombres. No hay ningún enlace “close” correspondiente, pero una instancia de `shlex` llamará el método `close()` de la secuencia de entrada de origen cuando retorna EOF.

Para un control más explícito del apilamiento de código fuente, utilice los métodos `push_source()` y `pop_source()`.

`shlex.push_source(newstream, newfile=None)`

Inserta una secuencia de fuente de entrada en la pila de entrada. Si se especifica el argumento de nombre de archivo, más adelante estará disponible para su uso en mensajes de error. Este es el mismo método utilizado internamente por el método `sourcehook()`.

`shlex.pop_source()`

Saca la última fuente de entrada de la pila de entrada. Este es el mismo método que el analizador léxico utiliza cuando llega al EOF en una secuencia de entrada apilada.

`shlex.error_leader(infile=None, lineno=None)`

Este método genera un mensaje de error líder en el formato de una etiqueta de error del compilador de Unix C; el formato es `"%s", line %d: '`, donde el `%s` es reemplazado con el nombre del archivo fuente actual y el `%d` con el número de línea de entrada actual (los argumentos opcionales pueden ser usado para sobrescribir estos).

Esta conveniencia se proporciona para animar a los usuarios de `shlex` a generar mensajes de error en el formato estándar y analizable que entienden Emacs y otras herramientas de Unix.

Las instancias de las subclases `shlex` tienen algunas variables de instancia pública que controlan el análisis léxico o pueden ser usadas para la depuración:

`shlex.commenters`

La cadena de caracteres que son reconocidos como comentarios de principiantes. Todos los caracteres desde el comentario principiante hasta el final de la línea son ignorados. Incluye sólo `'#'` por defecto.

`shlex.wordchars`

La cadena de caracteres que se acumulará en tokens de varios caracteres. Por defecto, incluye todos los alfanuméricos ASCII y el subrayado. En el modo POSIX, los caracteres acentuados del conjunto Latin-1 también están incluidos. Si `punctuation_chars` no está vacío, los caracteres `~-./*?=&`, que pueden aparecer en las especificaciones del nombre de archivo y en los parámetros de la línea de comandos, también se incluirán en este atributo, y cualquier carácter que aparezca en `punctuation_chars` será eliminado de `wordchars` si están presentes allí. Si `whitespace_split` se establece en `True`, esto no tendrá ningún efecto.

`shlex.whitespace`

Caracteres que serán considerados como espacio en blanco y saltados. El espacio blanco limita los tokens. Por defecto, incluye espacio, tabulación, salto de línea y retorno de carro.

`shlex.escape`

Caracteres que serán considerados como de escape. Esto sólo se usará en el modo POSIX, e incluye sólo `'\'` por defecto.

`shlex.quotes`

Los caracteres que serán considerados como citas de la cadena. El token se acumula hasta que la misma cita se

encuentra de nuevo (así, los diferentes tipos de citas se protegen entre sí como en el intérprete de comandos.) Por defecto, incluye ASCII comillas simples y dobles.

`shlex.escapedquotes`

Caracteres en *quotes* que interpretarán los caracteres de escape definidos en *escape*. Esto sólo se usa en el modo POSIX, e incluye sólo ' ' ' ' por defecto.

`shlex.whitespace_split`

Si es `True`, los tokens sólo se dividirán en espacios en blanco. Esto es útil, por ejemplo, para analizar las líneas de comando con *shlex*, obteniendo los tokens de forma similar a los argumentos de el intérprete de comandos. Cuando se utiliza en combinación con `punctuation_chars`, los tokens se dividirán en espacios en blanco además de esos caracteres.

Distinto en la versión 3.8: El atributo `punctuation_chars` se hizo compatible con el atributo *whitespace_split*.

`shlex.infile`

El nombre del archivo de entrada actual, como se estableció inicialmente al momento de la instanciación de la clase o apilado por solicitudes de fuente posteriores. Puede ser útil examinar esto cuando se construyan mensajes de error.

`shlex.instream`

El flujo de entrada del cual esta instancia *shlex* está leyendo caracteres.

`shlex.source`

Este atributo es `None` por defecto. Si le asignas una cadena, esa cadena será reconocida como una solicitud de inclusión a nivel léxico similar a la palabra clave *source* en varios intérpretes de comandos. Es decir, el token inmediatamente siguiente se abrirá como un nombre de archivo y la entrada se tomará de ese flujo hasta EOF, en cuyo momento se llamará al método *close()* de ese flujo y la fuente de entrada se convertirá de nuevo en el flujo de entrada original. Las peticiones de origen pueden ser apiladas a cualquier número de niveles de profundidad.

`shlex.debug`

Si este atributo es numérico y 1 o más, una instancia *shlex* imprimirá una salida de progreso verboso en su comportamiento. Si necesitas usar esto, puedes leer el código fuente del módulo para conocer los detalles.

`shlex.lineno`

Numero de linea de fuente (conteo de las nuevas lineas vistas hasta ahora mas uno).

`shlex.token`

El buffer de tokens. Puede ser útil examinarlo cuando se capturan excepciones.

`shlex.eof`

Token usado para determinar el final del archivo. Esto se ajustará a la cadena de caracteres vacía (' '), en el modo no-POSIX, y a `None` en el modo POSIX.

`shlex.punctuation_chars`

Una propiedad de sólo lectura. Caracteres que serán considerados como puntuación. Las series de caracteres de puntuación se retornarán como un único token. Sin embargo, tenga en cuenta que no se realizará ninguna comprobación de validez semántica: por ejemplo, “>>” podría ser retornado como un token, aunque no sea reconocido como tal por los intérpretes de comandos.

Nuevo en la versión 3.6.

24.3.2 Reglas de análisis

Cuando se opera en modo no-POSIX, *shlex* intentará obedecer las siguientes reglas.

- Los caracteres entre comillas no son reconocidos dentro de las palabras (Do "Not" Separate es analizado como la única palabra Do "Not" Separate);
- Los caracteres de escape no son reconocidos;
- El encerrar los caracteres entre comillas preserva el valor literal de todos los caracteres dentro de las comillas;
- Las comillas finales separan las palabras ("Do" Separate es analizado como "Do" y Separate);
- Si `whitespace_split` es `False`, cualquier carácter que no sea declarado como un carácter de palabra, espacio en blanco, o una cita será retornado como un token de un solo carácter. Si es `True`, *shlex* sólo dividirá las palabras en espacios en blanco;
- EOF es señalado con una cadena de caracteres vacía ('');
- No es posible analizar cadenas de caracteres vacías, incluso si se citan.

Cuando se opera en el modo POSIX, *shlex* intentará obedecer a las siguientes reglas de análisis.

- Las comillas se eliminan y no separan las palabras ("Do" "Not" "Separate" se analiza como la sola palabra DoNotSeparate);
- Los caracteres de escape no citados (por ejemplo '\') conservan el valor literal del siguiente carácter que continua;
- Encerrar caracteres entre comillas que no forman parte de *escapedquotes* (por ejemplo, '"') conservan el valor literal de todos los caracteres dentro de las comillas;
- Encerrar caracteres entre comillas que forman parte de *escapedquotes* (por ejemplo, '"') conserva el valor literal de todos los caracteres dentro de las comillas, con la excepción de los caracteres mencionados en *escape*. Los caracteres de escape conservan su significado especial solo cuando van seguidos de la comilla en uso, o el propio carácter de escape. De lo contrario, el carácter de escape sera considerado un carácter normal.
- EOF es señalado con un valor *None*;
- Se permiten cadenas de caracteres vacías entrecomilladas ('').

24.3.3 Compatibilidad mejorada con intérprete de comandos

Nuevo en la versión 3.6.

La clase *shlex* provee compatibilidad con el análisis realizado por los intérprete de comandos comunes de Unix como `bash`, `dash` y `sh`. Para aprovechar esta compatibilidad, especifica el argumento `punctuation_chars` en el constructor. Esto por defecto es `False`, el cual conserva el comportamiento pre-3.6. Sin embargo, si se establece como `True`, entonces se cambia el análisis de los caracteres `();<>|&`: cualquier ejecución de estos caracteres se retorna como un único token. Mientras que esto se queda corto en un analizador completo para los intérprete de comandos (que estaría fuera del alcance de la biblioteca estándar, dada la multiplicidad de intérpretes de comandos que hay), te permite realizar el procesamiento de las líneas de comandos más fácilmente de lo que podrías hacerlo de otra manera. Para ilustrarlo, puede ver la diferencia en el siguiente fragmento:

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> s = shlex.shlex(text, posix=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b;', 'c', '&&', 'd', '||', 'e;', 'f', '>'abc;', '(def', 'ghi)']
>>> s = shlex.shlex(text, posix=True, punctuation_chars=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', 'abc', ';', '(', 'def', 'ghi', ')']
```

Por supuesto, se retornarán tokens que no son válidos para los intérpretes de comandos y deberá implementar sus propias comprobaciones de errores en los tokens retornados.

En lugar de pasar `True` como valor para el parámetro `punctuation_chars`, puede pasar una cadena con caracteres específicos, que se usará para determinar qué caracteres constituyen puntuación. Por ejemplo:

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

Nota: Cuando es especificado `punctuation_chars`, el atributo `wordchars` se aumenta con los caracteres `~-./*?=`. Esto se debe a que estos caracteres pueden aparecer en los nombres de archivo (incluidos los comodines) y en los argumentos de la línea de comandos (por ejemplo, `--color=auto`). Por lo tanto:

```
>>> import shlex
>>> s = shlex.shlex('~ /a && b-c --color=auto || d *.py?',
...               punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

Sin embargo, para que coincida con el intérprete de comandos lo más cerca posible, se recomienda utilizar siempre `posix` y `whitespace_split` cuando se utiliza `punctuation_chars`, el cual negará por completo `wordchars`.

Para obtener el mejor efecto, `punctuation_chars` debe establecerse junto con `posix=True`. (Tenga en cuenta que `posix=False` es el valor predeterminado para `shlex`.)

Interfaces gráficas de usuario con Tk

Tk/Tcl ha sido durante mucho tiempo una parte integral de Python. Proporciona un conjunto de herramientas robusto e independiente de la plataforma para administrar ventanas. Disponible para desarrolladores a través del paquete *tkinter* y sus extensiones, los módulos *tkinter.tix* y *tkinter.ttk*.

El paquete *tkinter* es una capa delgada orientada a objetos encima de Tcl/Tk. Para usar el módulo *tkinter*, no tiene que escribir ningún código Tcl, pero deberá consultar la documentación de Tk, y ocasionalmente, la documentación de Tcl. El módulo *tkinter* es un conjunto de funciones que envuelven las implementaciones widgets Tk como clases de Python. Además, el módulo interno *_tkinter* proporciona un mecanismo robusto para que los hilos Python y Tcl interactúen.

Las ventajas de los módulos *tkinter* son su velocidad y que generalmente se suministra de forma nativa con Python. Aunque su documentación es deficiente, existen otros recursos: referencias, tutoriales, libros y otros. El módulo *tkinter* también es famoso por su aspecto desactualizado, el cual se ha mejorado mucho desde Tk 8.5. Sin embargo, hay muchas otras bibliotecas GUI que pueden interesarle. Para obtener más información sobre las alternativas, consulte la sección *Otros paquetes de interfaz gráfica de usuario*.

25.1 *tkinter* — Interface de Python para Tcl/Tk

Código fuente: `Lib/tkinter/__init__.py`

El paquete *tkinter* («interfaz Tk») es la interfaz por defecto de Python para el toolkit de la GUI Tk. Tanto Tk como *tkinter* están disponibles en la mayoría de las plataformas Unix, así como en sistemas Windows (Tk en sí no es parte de Python, es mantenido por ActiveState).

Ejecutar `python -m tkinter` desde la línea de comandos debería abrir una ventana que demuestre una interfaz Tk simple para saber si *tkinter* está instalado correctamente en su sistema. También muestra qué versión de Tcl/Tk está instalada para que pueda leer la documentación de Tcl/Tk específica de esa versión.

Ver también:

Documentación Tkinter:

Recursos de Python Tkinter La Guía de temas de Python Tkinter proporciona una gran cantidad de información sobre cómo usar Tk desde Python y enlaces a otras fuentes de información sobre Tk.

TKDocs Extensive tutorial plus friendlier widget pages for some of the widgets.

Tkinter 8.5 reference: a GUI for Python Material online de referencia.

Documentos de Tkinter de effbot Referencia en línea para Tkinter producida por effbot.org.

Programming Python Libro de Mark Lutz, que cubre Tkinter excelentemente bien.

Modern Tkinter for Busy Python Developers Libro de Mark Roseman sobre la construcción de interfaces gráficas de usuario atractivas y modernas con Python y Tkinter.

Python and Tkinter Programming Libro de John Grayson (ISBN 1-884777-81-3).

Documentación de Tcl/Tk:

Comandos Tk La mayoría de los comandos están disponibles como `tkinter` o `tkinter.ttk`. Cambie a “8.6” para que coincida con la versión de su instalación Tcl/Tk.

Páginas recientes del manual de Tcl/Tk Manual reciente de Tcl/Tk en www.tcl.tk.

Página de Inicio de ActiveState Tcl El desarrollo de Tk/Tcl se está llevando a cabo en gran medida en ActiveState.

Tcl and the Tk Toolkit Libro de John Ousterhout, el inventor de Tcl.

Practical Programming in Tcl and Tk Enciclopedia de Brent Welch.

25.1.1 Módulos Tkinter

La mayoría de las veces `tkinter` es todo lo que realmente necesita, pero también hay disponible varios módulos adicionales. La interfaz Tk se encuentra en un módulo binario llamado `_tkinter`. Este módulo contiene la interfaz de bajo nivel para Tk, y nunca debe ser utilizado directamente por los desarrolladores. Por lo general, es una biblioteca compartida (o DLL), pero en algunos casos puede estar vinculada estáticamente con el intérprete de Python.

Además del módulo de interfaz Tk, `tkinter` incluye varios módulos de Python, `tkinter.constants` es uno de los más importantes. Al Importar `tkinter` importará automáticamente `tkinter.constants`, por lo tanto, para usar Tkinter todo lo que se necesita es una simple declaración import:

```
import tkinter
```

O, más a menudo:

```
from tkinter import *
```

class `tkinter.Tk` (`screenName=None`, `baseName=None`, `className='Tk'`, `useTk=1`)

La clase `Tk` se instancia sin argumentos. Esto crea un widget de nivel superior de Tk que generalmente es la ventana principal de una aplicación. Cada instancia tiene su propio intérprete Tcl asociado.

`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.

Otros módulos que proporcionan soporte Tk incluyen:

`tkinter.scrolledtext` Widget de texto con una barra de desplazamiento vertical integrada.

`tkinter.colorchooser` Cuadro de diálogo que permite al usuario elegir un color.

`tkinter.commondialog` Clase base para cuadros de diálogo definidos en los otros módulos listados aquí.

`tkinter.filedialog` Cuadros de diálogo por defecto que permiten al usuario especificar un archivo para abrir o guardar.

`tkinter.font` Utilidades para ayudar a trabajar con fuentes.

`tkinter.messagebox` Acceso a cuadros de diálogo estándar de Tk.

`tkinter.simpledialog` Cuadros de diálogo simples y funciones útiles.

tkinter.dnd Soporte de arrastrar y soltar (*drag-and-drop*) para *tkinter*. Esto es experimental y ya no se mantendrá cuando sea reemplazado por Tk DND.

turtle Gráficos de tortuga en una ventana Tk.

25.1.2 Tkinter Life Preserver

Esta sección no está diseñada para ser un tutorial exhaustivo sobre Tk o Tkinter. Más bien, está pensado como un espacio intermedio que proporciona una orientación introductoria en el sistema.

Créditos:

- Tk fue escrito por John Ousterhout mientras estaba en Berkeley.
- Tkinter fue escrito por Steen Lumholt y Guido van Rossum.
- Esta guía de supervivencia fue escrita por Matt Conway en la Universidad de Virginia.
- The HTML rendering, and some liberal editing, was produced from a FrameMaker version by Ken Manheimer.
- Fredrik Lundh elaboró y revisó las descripciones de la interfaz de clase para actualizarlas con Tk 4.2.
- Mike Clarkson convirtió la documentación a LaTeX y compiló el capítulo de Interfaz de usuario del manual de referencia.

Cómo usar esta sección

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.

Al tratar de responder preguntas sobre cómo hacer «esto o aquello», a menudo es mejor descubrir cómo hacerlo en Tk y luego convertirlo a la función correspondiente *tkinter*. Los programadores de Python a menudo pueden adivinar el comando Python correcto consultando la documentación de Tk. Esto significa que para usar Tkinter deberá conocer un poco sobre Tk. Este documento no puede cumplir esa función, por lo que lo mejor que podemos hacer es señalarle la mejor documentación que existe. Aquí hay algunos consejos:

- Los autores sugieren encarecidamente obtener una copia de las páginas del manual de Tk. Específicamente, las páginas del directorio `manN` son las más útiles. Las páginas del manual `man3` describen la interfaz C para la biblioteca Tk y, por lo tanto, no son especialmente útiles para los desarrolladores de scripts.
- Addison-Wesley publica un libro llamado «*Tcl and the Tk Toolkit*» de John Ousterhout (ISBN 0-201-63337-X) que es una buena introducción a Tcl y Tk para novatos. El libro no es exhaustivo y para muchos detalles difiere de las páginas del manual.
- `tkinter/__init__.py` es el último recurso para la mayoría, pero puede ser un buen lugar para ir cuando nada más tiene sentido.

Un simple programa Hola Mundo

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        self.hi_there = tk.Button(self)
        self.hi_there["text"] = "Hello World\n(click me)"
        self.hi_there["command"] = self.say_hi
```

(continué en la próxima página)

(proviene de la página anterior)

```

self.hi_there.pack(side="top")

self.quit = tk.Button(self, text="QUIT", fg="red",
                       command=self.master.destroy)
self.quit.pack(side="bottom")

def say_hi(self):
    print("hi there, everyone!")

root = tk.Tk()
app = Application(master=root)
app.mainloop()

```

25.1.3 Una (muy) rápida mirada a Tcl/Tk

La jerarquía de clases parece complicada, pero en la práctica, los programadores de aplicaciones casi siempre se refieren a las clases en la parte inferior de la jerarquía.

Notas:

- Estas clases se proporcionan con el propósito de organizar ciertas funciones en un solo un espacio de nombres. No están destinadas a ser instanciadas independientemente.
- La clase `Tk` está destinada a ser instanciada solo una vez en una aplicación. Los desarrolladores no necesitan crear una instancia explícitamente. El sistema crea una cada vez que se instancia cualquiera de las otras clases.
- 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”).

Para hacer uso de este material de referencia, habrá momentos en los que necesitará saber cómo leer pasajes cortos de Tk y cómo identificar las diversas partes de un comando Tk. Consulte la sección [Mapeo básico de Tk en Tkinter](#) para los equivalentes `tkinter` de lo que se muestra a continuación.

Los scripts Tk son programas Tcl. Como todos los programas Tcl, los scripts Tk son solo listas de tokens separados por espacios. Un widget Tk es solo su *clase*, las *opciones* que ayudan a configurarlo y las *acciones* que lo hacen hacer cosas útiles.

Para hacer un widget en Tk, el comando siempre tiene la siguiente forma:

```
classCommand newPathname options
```

classCommand denota qué tipo de widget hacer (un botón, una etiqueta, un menú...)

newPathname es el nuevo nombre para este widget. Todos los nombres en Tk deben ser únicos. Para ayudar a aplicar esto, los widgets en Tk se nombran con *rutras*, al igual que los archivos en un sistema de archivos. El widget de nivel superior, el *root*, se llama `.` (punto) y los elementos secundarios están delimitados por más puntos. Por ejemplo, `.myApp.controlPanel.okButton` podría ser el nombre de un widget.

options configurar la apariencia del widget y, en algunos casos, su comportamiento. Las opciones vienen en forma de una lista de parámetros y valores. Los parámetros están precedidas por un “-”, como los parámetros en una shell de Unix, y los valores se ponen entre comillas si son más de una palabra.

Por ejemplo:

```

button    .fred    -fg red -text "hi there"
  ^         ^          \_____/
  |         |            |
class    new          options
command widget (-opt val -opt val ...)

```

Una vez creado, la ruta de acceso al widget se convierte en un nuevo comando. Este nuevo *comando de widget* es el identificador para que el nuevo widget realice alguna *acción*. En C, expresarías esto como *someAction(fred, someOptions)*; en C++, expresarías esto como *fred.someAction(someOptions)*, y en Tk:

```
.fred someAction someOptions
```

Tenga en cuenta que el nombre del objeto, `.fred`, comienza con un punto.

Como era de esperar, los valores legales para *someAction* dependerán de la clase del widget: `.fred disable` funciona si *fred* es un botón (se atenúa), pero no funciona si *fred* es una etiqueta (la desactivación de etiquetas no es compatible con Tk).

Los valores legales de *someOptions* dependen de la acción. Algunas acciones, como `disable`, no requieren argumentos; otras, como el comando `delete` de un cuadro de entrada de texto, necesitarían argumentos para especificar qué rango de texto eliminar.

25.1.4 Mapeo básico de Tk en Tkinter

Los comandos de clase en Tk corresponden a constructores de clase en 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 hyphenated 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 [Configuración de opciones](#) 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 `tkinter/__init__.py`.

```
.fred invoke          =====> fred.invoke()
```

Para pasar el widget al empaquetador (que administra el diseño de la pantalla) en Tk, llame al comando *pack* con argumentos opcionales. En Tkinter, la clase *Pack* tiene todas estas funcionalidades y las diferentes formas del comando *pack* se implementan como métodos. Todos los widgets en *tkinter* son subclasses del empaquetador, por lo que heredan todos los métodos de empaquetado. Consulte la documentación del módulo *tkinter.tix* para obtener más información sobre el administrador de diseño de formularios.

```
pack .fred -side left  =====> fred.pack(side="left")
```

25.1.5 Cómo se relacionan Tk y Tkinter

De arriba para abajo:

Tu aplicación (Python) Una aplicación Python hace una llamada `tkinter`.

tkinter (paquete de Python) Esta llamada (por ejemplo, crear un widget de botón) se implementa en el paquete `tkinter`, que está escrito en Python. Esta función de Python analizará los comandos y los argumentos y los convertirá en una forma que los haga ver como si vinieran de un script Tk en lugar de un script Python.

_tkinter (C) Estos comandos y sus argumentos se pasarán a una función C en el módulo de extensión `_tkinter` -obsérvese el guión bajo-.

Tk Widgets (C y Tcl) Esta función en C puede realizar llamadas a otros módulos C, incluidas las funciones de C que forman la biblioteca Tk. Tk se implementa usando C y un poco de Tcl. La parte Tcl de los widgets Tk se usa para vincular ciertos comportamientos predeterminados de los widgets, y se ejecuta una vez cuando se importa el paquete Python `tkinter` (el usuario nunca ve esta etapa).

Tk (C) La parte Tk de los widgets Tk implementa el mapeo final a...

Xlib (C) la biblioteca Xlib para dibujar elementos gráficos en la pantalla.

25.1.6 Guía práctica

Configuración de opciones

Las opciones controlan parámetros como el color y el ancho del borde de un widget. Las opciones se pueden configurar de tres maneras:

En el momento de la creación del objeto, utilizando argumentos de palabras clave

```
fred = Button(self, fg="red", bg="blue")
```

Después de la creación del objeto, tratando el nombre de la opción como un índice de diccionario

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

Usando el método `config()` para actualizar múltiples atributos después de la creación del objeto

```
fred.config(fg="red", bg="blue")
```

Para obtener una explicación completa de las opciones y su comportamiento, consulte las páginas de manual de Tk para el widget en cuestión.

Tenga en cuenta que las páginas del manual enumeran «OPCIONES ESTÁNDAR» y «OPCIONES ESPECÍFICAS DE WIDGET» para cada widget. La primera es una lista de opciones que son comunes a muchos widgets, la segunda son las opciones que son específicas para ese widget en particular. Las opciones estándar están documentadas en la página del manual `options(3)`.

No se hace distinción entre las opciones estándar y las opciones específicas del widget en este documento. Algunas opciones no se aplican a algunos tipos de widgets. Si un determinado widget responde a una opción particular depende de la clase del widget. Los botones tienen la opción `command`, las etiquetas no.

Las opciones admitidas por un widget dado se enumeran en la página de manual de ese widget, o se pueden consultar en tiempo de ejecución llamando al método `config()` sin argumentos, o llamando al método `keys()` en ese widget. El valor retornado en esas llamadas es un diccionario cuya clave es el nombre de la opción como una cadena (por ejemplo, `'relief'`) y cuyo valor es una tupla de 5 elementos.

Algunas opciones, como `bg`, son sinónimos de opciones comunes con nombres largos (`bg` es la abreviatura de «*background*»). Al pasar el método `config()`, el nombre de una opción abreviada retornará una tupla de 2 elementos (en lugar de 5). Esta tupla contiene nombres de sinónimos y nombres de opciones «reales» (como `('bg', 'background')`).

Índice	Significado	Ejemplo
0	nombre de la opción	'relief'
1	option name for database lookup	'relief'
2	clase de la opción para la consulta de base de datos	'Relief'
3	valor por defecto	'raised'
4	valor actual	'groove'

Ejemplo:

```
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

Por supuesto, el diccionario impreso incluirá todas las opciones disponibles y sus valores. Esto es solo un ejemplo.

El empaquetador

The packer is one of Tk's geometry-management mechanisms. Geometry managers are used to specify the relative 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.

El tamaño de cualquier widget *master* está determinado por el tamaño del «widget esclavo» interno. El empaquetador se usa para controlar dónde se colocará el widget esclavo en el widget *maestro* de destino. Para lograr el diseño deseado, puede empaquetar el widget en un marco y luego empaquetar ese marco en otro. Además, una vez empaquetado, la disposición se ajusta dinámicamente de acuerdo con los cambios de configuración posteriores.

Tenga en cuenta que los widgets no aparecen hasta que su geometría no se haya especificado con un administrador de diseño de pantalla. Es un error común de principiante ignorar la especificación de la geometría, y luego sorprenderse cuando se crea el widget pero no aparece nada. Un objeto gráfico solo aparece después que, por ejemplo, se le haya aplicado el método `pack()` del empaquetador.

Se puede llamar al método `pack()` con pares palabra-clave/valor que controlan dónde debe aparecer el widget dentro de su contenedor y cómo se comportará cuando se cambie el tamaño de la ventana principal de la aplicación. Aquí hay unos ejemplos:

```
fred.pack()                                # defaults to side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```

Opciones del empaquetador

Para obtener más información sobre el empaquetador y las opciones que puede tomar, consulte el manual y la página 183 del libro de John Ousterhout.

anchor Tipo de anclaje. Indica dónde debe colocar el empaquetador a cada esclavo en su espacio.

expand Un valor booleano, 0 o 1.

fill Los valores legales son: 'x', 'y', 'both', 'none'.

ipadx y ipady Una distancia que designa el relleno interno a cada lado del widget esclavo.

padx y pady Una distancia que designa el relleno externo a cada lado del widget esclavo.

side Los valores legales son: 'left', 'right', 'top', 'bottom'.

Asociación de variables de widget

La asignación de un valor a ciertos objetos gráficos (como los widgets de entrada de texto) se puede vincular directamente a variables en su aplicación utilizando opciones especiales. Estas opciones son `variable`, `textvariable`, `onvalue`, `offvalue`, y `value`. Esta conexión funciona en ambos sentidos: si la variable cambia por algún motivo, el widget al que está conectado se actualizará para reflejar el nuevo valor.

Desafortunadamente, en la implementación actual de `tkinter` no es posible entregar una variable arbitraria de Python a un widget a través de una opción `variable` o `textvariable`. Los únicos tipos de variables para las cuales esto funciona son variables que son subclases de la clase `Variable`, definida en `tkinter`.

Hay muchas subclases útiles de `Variable` ya definidas: `StringVar`, `IntVar`, `DoubleVar`, and `BooleanVar`. Para leer el valor actual de dicha variable, es necesario llamar al método `get()`, y para cambiar su valor, al método `set()`. Si se sigue este protocolo, el widget siempre rastreará el valor de la variable, sin ser necesaria ninguna otra intervención.

Por ejemplo:

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()

        self.entrythingy = tk.Entry()
        self.entrythingy.pack()

        # Create the application variable.
        self.contents = tk.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

        # Define a callback for when the user hits return.
        # It prints the current value of the variable.
        self.entrythingy.bind('<Key-Return>',
                               self.print_contents)

    def print_contents(self, event):
        print("Hi. The current entry content is:",
              self.contents.get())

root = tk.Tk()
myapp = App(root)
myapp.mainloop()
```

El gestor de ventanas

En Tk hay un comando útil, `wm`, para interactuar con el gestor de ventanas. Las opciones del comando `wm` le permiten controlar cosas como títulos, ubicación, iconos de ventana y similares. En `tkinter`, estos comandos se han implementado como métodos de la clase `Wm`. Los widgets de *Toplevel* son subclases de `Wm`, por lo que se puede llamar directamente a los métodos de `Wm`.

Para acceder a la ventana *Toplevel* que contiene un objeto gráfico dado, a menudo puede simplemente referirse al padre de este objeto gráfico. Por supuesto, si el objeto gráfico fue empaquetado dentro de un marco, el padre no representará la ventana de nivel superior. Para acceder a la ventana de nivel superior que contiene un objeto gráfico arbitrario, puede llamar al método `_root()`. Este método comienza con un subrayado para indicar que esta función es parte de la implementación y no de una interfaz a la funcionalidad Tk.

Aquí hay algunos ejemplos típicos:

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master=None):
        super().__init__(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()
```

Tipos de datos de opciones Tk

anchor Los valores legales son los puntos de la brújula: "n", "ne", "e", "se", "s", "sw", "w", "nw", y también "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 Se puede pasar enteros 0 o 1 o las cadenas "yes" or "no".

callback Esto es cualquier función de Python que no toma argumentos. Por ejemplo:

```
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: "#RRRRGGGBBB", 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 Los nombres estándar del cursor X de `cursorfont.h` se pueden usar sin el prefijo XC_. Por ejemplo, para obtener un cursor de mano (XC_hand2), use la cadena "hand2". También se puede especificar su propio mapa de bits y archivo de máscara. Ver página 179 del libro de Ousterhout.

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 usa un formato de lista para los nombres de fuentes, como {courier 10 bold}. Los tamaños de fuente expresados en números positivos se miden en puntos, mientras que los tamaños con números negativos se miden en píxeles.

geometry Es una cadena de caracteres del estilo widthxheight, donde el ancho y la altura se miden en píxeles para la mayoría de los widgets (en caracteres para widgets que muestran texto). Por ejemplo: fred["geometry"] = "200x100".

justify Los valores legales son las cadenas de caracteres: "left", "center", "right", y "fill".

region Es una cadena de caracteres con cuatro elementos delimitados por espacios, cada uno de ellos es una distancia legal (ver arriba). Por ejemplo: "2 3 4 5", "3i 2i 4.5i 2i" y "3c 2c 4c 10.43c" son todas regiones legales.

relief Determina cuál será el estilo de borde de un widget. Los valores legales son: "raised", "sunken", "flat", "groove", y "ridge".

scrollcommand Este es casi siempre el método `set()` de algún widget de barra de desplazamiento, pero puede ser cualquier método de un widget que tome un solo argumento.

wrap: Debe ser uno de estos: "none", "char", o "word".

Enlaces y eventos

El método de enlace (*binding*) del comando del widget le permite observar ciertos eventos y hacer que la función de devolución de llamada se active cuando se produce ese tipo de evento. La forma del método de enlace es:

```
def bind(self, sequence, func, add=''):
```

donde:

sequence es una cadena que denota el tipo de evento objetivo. Para obtener más información, consulte la página del manual de Bind y la página 201 del libro de John Ousterhout.

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 es opcional, ya sea ' ' o '+'. Pasar una cadena de caracteres vacía indica que este enlace anulará cualquier otro enlace asociado con este evento. Pasar '+' agrega esta función a la lista de funciones vinculadas a este tipo de evento.

Por ejemplo:

```
def turn_red(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turn_red)
```

Notice how the widget field of the event is being accessed in the `turn_red()` 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	Campo evento de Tkinter	Tk	Campo evento de Tkinter
%f	focus	%A	char
%h	height	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

El parámetro índice

Muchos widgets requieren que se pase un parámetro *índice*. Se utiliza para señalar ubicaciones específicas en el widget de texto, caracteres específicos en el widget de entrada, o elementos particulares en el widget de menú.

Índice de widget de entrada (índice, índice de vista, etc.) El widget entrada tiene una opción para referirse a la posición de los caracteres del texto que se muestra. Puede acceder a estos puntos especiales en un widget de texto utilizando la siguiente función de *tkinter*:

Índice de widget de texto La notación de índice del widget de texto es muy rica y se detalla mejor en las páginas del manual de Tk.

Índices de menú (*menu.invoke()*, *menu.entryconfig()*, etc.) Algunas opciones y métodos para menús manipulan entradas de menú específicas. Cada vez que se necesita un índice de menú para una opción o un parámetro, se puede pasar:

- un número entero que se refiere a la posición numérica de la entrada en el widget, contada desde arriba, comenzando con 0;
- la cadena "active", que se refiere a la posición del menú que está actualmente debajo del cursor;
- la cadena de caracteres "last" que se refiere al último elemento del menú;
- Un número entero precedido por @, como en @6, donde el entero es interpretado como una coordenada y de píxeles en el sistema de coordenadas del menú;
- la cadena de caracteres "none", que indica que no hay entrada de menú, usado frecuentemente con *menu.activate()* para desactivar todas las entradas; y, finalmente,
- una cadena de texto cuyo patrón coincide con la etiqueta de la entrada del menú, tal como se explora desde la parte superior del menú hasta la parte inferior. Tenga en cuenta que este tipo de índice se considera después de todos los demás, lo que significa que las coincidencias para los elementos del menú etiquetados last, active, o none pueden interpretarse de hecho como los literales anteriores.

Imágenes

Se pueden crear imágenes de diferentes formatos a través de la correspondiente subclase de `tkinter.Image`:

- `BitmapImage` para imágenes en formato XBM.
- `PhotoImage` para imágenes en formatos PGM, PPM, GIF y PNG. Este último es compatible a partir de Tk 8.6.

Cualquier tipo de imagen se crea a través de la opción `file` o `data` (también hay otras opciones disponibles).

El objeto imagen se puede usar siempre que algún widget admita una opción de `image` (por ejemplo, etiquetas, botones, menús). En estos casos, Tk no mantendrá una referencia a la imagen. Cuando se elimina la última referencia de Python al objeto de imagen, los datos de la imagen también se eliminan, y Tk mostrará un cuadro vacío donde se utilizó la imagen.

Ver también:

El paquete [Pillow](#) añade soporte para los formatos del tipo BMP, JPEG, TIFF, y WebP, entre otros.

25.1.7 Gestor de archivos

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)
```

Esta función no está disponible en Windows.

Dado que no se sabe cuántos bytes están disponibles para su lectura, no use métodos de *BufferedIOBase* o *TextIOBase* *read()* o *readline()*, ya que estos requieren leer un número predefinido de bytes. Para *sockets*, los métodos *recv()* o *recvfrom()* trabajan bien; para otros archivos, use lectura sin formato o `os.read(file.fileno(), maxbytecount)`.

`Widget.tk.createfilehandler(file, mask, func)`

Registra la función *callback* gestor de archivos *func*. El argumento *file* puede ser un objeto con un método *fileno()* (como un archivo u objeto de socket), o un descriptor de archivo entero. El argumento *mask* es

una combinación ORed de cualquiera de las tres constantes que siguen. La retrollamada se llama de la siguiente manera:

```
callback(file, mask)
```

`Widget.tk.deletefilehandler` (*file*)

Anula el registro de un gestor de archivos.

`tkinter.READABLE`

`tkinter.WRITABLE`

`tkinter.EXCEPTION`

Constantes utilizadas en los argumentos *mask*.

25.2 `tkinter.ttk` — Tk widgets temáticos

Código fuente: [Lib/tkinter/ttk.py](#)

El módulo `tkinter.ttk` proporciona acceso al conjunto de widgets temáticos Tk, introducido en Tk 8.5. Si Python no se ha compilado con Tk 8.5, todavía se puede acceder a este módulo si se ha instalado *Tile*. El método anterior que utiliza Tk 8.5 proporciona ventajas adicionales, incluida la representación de fuentes suavizada en X11 y la transparencia de ventanas (requiere un administrador de ventanas de composición en X11).

La idea básica de `tkinter.ttk` es separar, en la medida de lo posible, el comportamiento de un widget del código que implementa su apariencia.

Ver también:

Soporte para estilos de Widgets Tk Un documento que presenta apoyo temático para Tk

25.2.1 Uso de Ttk

Para empezar a utilizar Ttk, hay que importar su módulo:

```
from tkinter import ttk
```

Para anular los widgets Tk básicos, la importación debe seguir la importación de Tk:

```
from tkinter import *
from tkinter.ttk import *
```

Ese código hace que varios widgets `tkinter.ttk` (`Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` y `Scrollbar`) reemplacen automáticamente los widgets Tk.

Esto tiene el beneficio de usar los nuevos widgets que dan una mejor apariencia en todas las plataformas; sin embargo, el reemplazo de widgets no es completamente compatible. La principal diferencia es que las opciones de widgets como «fg», «bg» y otras relacionadas con el estilo del widget ya no están presentes en los widgets de Ttk. En su lugar, utiliza la clase `ttk.Style` para mejorar los efectos de estilo.

Ver también:

Conversión de aplicaciones existentes para usar widgets Tile Una monografía (utilizando la terminología Tcl) sobre las diferencias que normalmente se encuentran al modificar aplicaciones para usar los nuevos widgets.

25.2.2 Ttk widgets

Ttk viene con 18 widgets, doce de los cuales ya existían en tkinter: `Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale`, `Scrollbar` y `Spinbox`. Los otros seis son nuevos: `Combobox`, `Notebook`, `Progressbar`, `Separator`, `Sizegrip` y `Treeview`. Y todas ellas son subclases de `Widget`.

El uso de los widgets Ttk le da a la aplicación un aspecto mejorado. Como se ha mencionado anteriormente, hay diferencias en cómo se codifica el estilo.

Código Tk:

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Código Ttk:

```
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")
```

Para obtener más información acerca de *TtkStyling*, consulta la documentación de la clase `Style`.

25.2.3 Widget

`ttk.Widget` define las opciones y métodos estándar compatibles con los widgets temáticos de Tk y no se crea una instancia directamente.

Opciones estándar

Todos los widgets `ttk` aceptan las siguientes opciones:

Opción	Descripción
<code>class</code>	Especifica la clase de ventana. La clase se usa cuando se consulta la base de datos de opciones para las otras opciones de la ventana, para determinar las etiquetas de enlace (<i>bindtags</i>) predeterminadas para la ventana y para seleccionar el diseño y estilo predeterminados del widget. Esta opción es de solo lectura y solo se puede especificar cuando se crea la ventana.
<code>cursor</code>	Especifica el cursor del mouse que se utilizará para el widget. Si se establece en la cadena vacía (el valor predeterminado), el cursor se hereda para el widget principal.
<code>takefocus</code>	Determina si la ventana acepta el foco durante el recorrido del teclado. Se retorna 0, 1 o una cadena vacía. Si se retorna 0, significa que la ventana debe omitirse por completo durante el recorrido del teclado. Si es 1, significa que la ventana debe recibir el foco de entrada siempre que sea visible. Y una cadena vacía significa que los scripts transversales toman la decisión sobre si enfocarse o no en la ventana.
<code>style</code>	Se puede usar para especificar un estilo personalizado para el widget.

Opciones de widgets desplegables

Los widgets controlados por una barra deslizante presentan las siguientes opciones.

Opción	Descripción
xscrollcommand	Se usa para interactuar con las barras deslizantes horizontales. Cuando la vista en la ventana del widget cambia, el widget generará un comando Tcl basado en el scrollcommand. Por lo general, esta opción consiste en el método <code>Scrollbar.set()</code> de barras deslizantes. Esto hará que la barra deslizante se actualice cada vez que cambie la vista de la ventana.
yscrollcommand	Se utiliza para comunicarse con las barras deslizantes verticales. Para obtener más información, consulta más arriba.

Opciones de etiqueta

Las siguientes opciones son compatibles con etiquetas, botones y otros widgets similares a botones.

Opción	Descripción
text	Especifica una cadena de texto que se mostrará dentro del widget.
textvariable	Especifica un nombre cuyo valor se utilizará en lugar del recurso de opción de texto.
underline	Si se activa, especifica el índice (empezando por 0) de un carácter que se va a subrayar en la cadena de texto. El carácter subrayado se utiliza para la activación mnemotécnica.
image	Especifica una imagen que se va a mostrar. Es una lista de 1 o más elementos. El primer elemento es el nombre de imagen predeterminado. El resto de la lista es una secuencia de pares <code>statespec/value</code> según lo definido por <code>Style.map()</code> , especificando diferentes imágenes para usar cuando el widget está en un estado determinado o una combinación de estados. Todas las imágenes de la lista deben tener el mismo tamaño.
compound	Especifica cómo mostrar la imagen en relación con el texto, en el caso de que estén presentes las opciones de texto e imágenes. Los valores válidos son: <ul style="list-style-type: none">• text: mostrar solo texto• image: mostrar solo la imagen• top, bottom, left, right: muestra la imagen por encima, por debajo, a la izquierda o a la derecha del texto, respectivamente.• none: valor predeterminado. Mostrar la imagen si está presente, de lo contrario el texto.
width	Si es mayor que cero, especifica cuánto espacio, en ancho de caracteres, se debe asignar para la etiqueta de texto; si es menor que cero, especifica un ancho mínimo. Si es cero o no se especifica, se utiliza el ancho natural de la etiqueta de texto.

Opciones de compatibilidad

Opción	Descripción
state	Se puede establecer en «normal» o «deshabilitado» para controlar el bit de estado «deshabilitado». Esta es una opción de solo escritura: establecerlo cambia el estado del widget, pero el método <code>Widget.state()</code> no afecta a esta opción.

Estados del widget

El estado del widget es un mapa de bits de indicadores de estado independientes.

Indicador de estado	Descripción
active	El puntero está sobre el widget y clickeando sobre él producirá alguna acción
disabled	El widget está desactivado bajo el control del programa
focus	El widget tiene el enfoque del teclado
pressed	El widget está siendo pulsado
selected	«On», «true» o «current» para aspectos como Checkbuttons y radiobuttons
background	Windows y Mac tienen el concepto de ventana «activa» o en primer plano. El estado <i>background</i> se establece para los widgets en una ventana de fondo y se borra para los que están en la ventana en primer plano
readonly	El widget no debe permitir la modificación del usuario
alternate	Un formato de visualización alternativo específico del widget
invalid	El valor del widget no es válido

Una especificación de estado es una secuencia de nombres de estado, opcionalmente prefijados con un signo de exclamación que indica que el bit está desactivado.

ttk.Widget

Además de los métodos descritos a continuación, el `ttk.Widget` soporta los métodos `tkinter.Widget.cget()` y `tkinter.Widget.configure()`.

class `tkinter.ttk.Widget`

identify (*x*, *y*)

Retorna el nombre del elemento en la posición *x* y *y* o la cadena vacía si el punto no se encuentra dentro de ningún elemento.

x e *y* son coordenadas en píxeles relativas al widget.

instate (*statespec*, *callback=None*, **args*, ***kw*)

Prueba el estado del widget. Si no se especifica una retrollamada, retorna `True` si el estado del widget coincide con *statespec* y `False` en caso contrario. Si se especifica la retrollamada, se llama con argumentos (*args*) si el estado del widget coincide con *statespec*.

state (*statespec=None*)

Modifica o pregunta el estado del widget. Si se especifica *statespec*, establece el estado del widget según éste y retorna un nuevo *statespec* que indica qué indicadores se han cambiado. Si no se especifica *statespec*, retorna los indicadores de estado habilitados actualmente.

statespec es generalmente una lista o una tupla.

25.2.4 Combobox

El widget `ttk.Combobox` combina un campo de texto con una lista desplegable de valores. Este widget es una subclase de `Entry`.

Además de los métodos heredados de `Widget`: `Widget.cget()`, `Widget.configure()`, `Widget.identify()`, `Widget.instate()` y `Widget.state()`, y los siguientes heredados de `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()`, tiene algunos otros métodos, descritos en `ttk.Combobox`.

Opciones

Este widget acepta las siguientes opciones específicas:

Opción	Descripción
<code>exportselection</code>	Valor booleano. Si se establece, la selección del widget está vinculada a la selección del Administrador de ventanas (que se puede retornar invocando <code>Misc.selection_get</code> , por ejemplo).
<code>justify</code>	Especifica cómo el texto se alinea en el widget. Uno de «left», «center» o «right».
<code>height</code>	Especifica el largo/altura del cuadro de la lista desplegable, en filas.
<code>postcommand</code>	Un script (posiblemente registrado con <code>Misc.register</code>) que se llama inmediatamente antes de mostrar los valores. Puede especificar qué valores mostrar.
<code>state</code>	Uno de «normal», «readonly» o «disabled». En el estado «readonly» el valor no se puede editar directamente y el usuario solo puede seleccionar los valores de la lista desplegable. En el estado «normal» el campo de texto se puede editar directamente. En el estado «deshabilitado» no es posible ninguna interacción.
<code>textvariable</code>	Especifica un nombre cuyo valor está vinculado al valor del widget. Cada vez que cambia el valor asociado a ese nombre, se actualiza el valor del widget y viceversa. Véase <code>tkinter.StringVar</code> .
<code>values</code>	Especifica la lista de valores que se mostrarán en el cuadro de lista desplegable.
<code>width</code>	Especifica un valor entero que indica el ancho deseado de la ventana de entrada, en caracteres de tamaño medio de la fuente del widget.

Eventos virtuales

Los widgets de cuadro combinado generan un evento virtual **«ComboboxSelected»** cuando el usuario selecciona un elemento de la lista de valores.

`ttk.Combobox`

```
class tkinter.ttk.Combobox
```

```
current (newindex=None)
```

Si se especifica *newindex*, establece el valor del cuadro combinado en la posición del elemento *newindex*. De lo contrario, retorna el índice del valor actual o -1 si el valor actual no está en la lista de valores.

```
get ()
```

Retorna el valor actual del cuadro combinado.

```
set (value)
```

Establece el valor del cuadro combinado a *value*.

25.2.5 Spinbox

El widget `ttk.Spinbox` es un `ttk.Entry` mejorado con flechas de incremento y decremento. Se puede utilizar para números o listas de valores de cadena. Este widget es una subclase de `Entry`.

Además de los métodos heredados de `Widget`: `Widget.cget()`, `Widget.configure()`, `Widget.identify()`, `Widget.instate()` y `Widget.state()`, y los siguientes heredados de `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.xview()`, tiene algunos otros métodos, descritos en `ttk.Spinbox`.

Opciones

Este widget acepta las siguientes opciones específicas:

Opción	Descripción
<code>from</code>	Valor flotante. Si se establece, este es el valor mínimo al que se reducirá el botón de disminución. Debe escribirse como <code>from_</code> cuando se usa como argumento, ya que <code>from</code> es una palabra clave de Python.
<code>to</code>	Valor flotante. Si se establece, este es el valor máximo al que se incrementará el botón de incremento.
<code>increment</code>	Valor flotante. Especifica la cantidad por la cual los botones de incremento/disminución cambian el valor. Por defecto la cantidad es de 1.0.
<code>values</code>	Secuencia de valores de cadena o flotantes. Si se especifica, los botones de incremento/disminución recorrerán los elementos de esta secuencia en lugar de incrementar o disminuir los números.
<code>wrap</code>	Valor booleano. Si es <code>True</code> , los botones de incremento y disminución pasarán del valor <code>to</code> al valor <code>from</code> o del valor <code>from</code> al valor <code>to</code> , respectivamente.
<code>format</code>	Valor de cadena. Especifica el formato de los números establecidos por los botones de incremento/disminución. Debe tener la forma <code>«%W.Pf»</code> , donde <code>W</code> es el ancho de relleno del valor, <code>P</code> es la precisión, y <code>“%”</code> y <code>“f”</code> son literales.
<code>command</code>	Python invocable. Se llamará sin argumentos cada vez que se presione alguno de los botones de incremento o disminución.

Eventos virtuales

El widget `spinbox` genera un evento virtual **«Increment»** cuando el usuario presiona `<Up>`, y un evento virtual **«Decrement»** cuando el usuario presiona `<Down>`.

ttk.Spinbox

```
class tkinter.ttk.Spinbox
```

```
get ()
    Retorna el valor actual del spinbox.

set (value)
    Establece el valor del spinbox a value.
```

25.2.6 Notebook

El widget Tk Notebook administra una colección de ventanas y muestra una sola a la vez. Cada ventana secundaria está asociada a una pestaña, que el usuario puede seleccionar para cambiar la ventana que se muestra actualmente.

Opciones

Este widget acepta las siguientes opciones específicas:

Opción	Descripción
height	Si está presente y es mayor que cero, especifica la altura deseada del área del panel (sin incluir el relleno interno o las pestañas). De lo contrario, se utiliza la altura máxima de todos los paneles.
padding	Especifica la cantidad de espacio adicional que se va a agregar alrededor del exterior del bloc de notas. El relleno es una lista de hasta cuatro especificaciones de longitud izquierda superior derecha inferior. Si se especifican menos de cuatro elementos, el valor predeterminado inferior es el superior, el valor predeterminado de la derecha es el de la izquierda y el valor predeterminado superior es el de la izquierda.
width	Si está presente y es mayor que cero, especifica el ancho deseado del área del panel (sin incluir el relleno interno). De lo contrario, se utiliza el ancho máximo de todos los paneles.

Opciones de pestañas

Las opciones específicas para pestañas son:

Opción	Descripción
state	O bien «normal», «disabled» o «hidden». Si es «disabled», la pestaña no se puede seleccionar. Si es «hidden», la pestaña no se muestra.
sticky	Especifica cómo se coloca la ventana secundaria dentro del área del panel. <i>Value</i> es una cadena que contiene cero o más de los caracteres «n», «s», «e» o «w». Cada letra se refiere a un lado (norte, sur, este u oeste) al que se pegará la ventana secundaria, según el administrador de geometría <code>grid()</code> .
padding	Especifica la cantidad de espacio adicional que se va a agregar entre el notebook y este panel. La sintaxis es la misma que para el relleno de opciones utilizado por Notebook.
text	Especifica un texto que se muestra en la pestaña.
image	Especifica una imagen que se muestra en la pestaña. Consulta la opción de imagen descrita en Widget .
compound	Especifica cómo mostrar la imagen en relación con el texto, en el caso de que tanto el texto como la imagen estén presentes. Consulta Label Options para obtener valores válidos.
underline	Especifica el índice (basado en 0) de un carácter que se va a subrayar en la cadena de texto. El carácter subrayado se utiliza para la activación mnemotécnica si se llama a <code>Notebook.enable_traversal()</code> .

Identificadores de pestañas

El `tab_id` presente en varios métodos de `ttk.Notebook` puede tener cualquiera de las siguientes formas:

- Un entero entre cero y el número de pestañas
- El nombre de una ventana secundaria
- Un especificación de posición de la forma «@x,y» que identifique la pestaña
- El valor «current» que identifica la pestaña seleccionada actualmente
- El valor «end» que retorna el número de pestañas (válido solo para `Notebook.index()`)

Eventos virtuales

Este widget genera un evento virtual «**NotebookTabChanged**» después de seleccionar una nueva pestaña.

ttk.Notebook

class `tkinter.ttk.Notebook`

add (*child*, ***kw*)

Añade una nueva pestaña al notebook.

Si la ventana está actualmente administrada por el notebook pero oculta, se restaura a su posición anterior.

Consulta [Tab Options](#) para la lista de opciones disponibles.

forget (*tab_id*)

Quita la pestaña especificada por *tab_id*, desasigna y quita la ventana asociada.

hide (*tab_id*)

Oculta la pestaña especificada por *tab_id*.

La pestaña no se mostrará, pero la ventana asociada permanece administrada por el notebook y se recordará su configuración. Las pestañas ocultas se pueden restaurar con el comando `add()`.

identify (*x*, *y*)

Retorna el nombre del elemento de la pestaña en la posición *x*, *y* o cadena vacía si no hay ninguno.

index (*tab_id*)

Retorna el índice numérico de la pestaña especificada por *tab_id*, o el número total de pestañas si *tab_id* es la cadena «end».

insert (*pos*, *child*, ***kw*)

Añade un panel en la posición especificada.

pos es la cadena «end», un índice entero o el nombre de un elemento secundario administrado. Si el bloc de notas ya administra *child*, lo mueve a la posición especificada.

Consulta [Tab Options](#) para la lista de opciones disponibles.

select (*tab_id=None*)

Selecciona el *tab_id* especificado.

Se mostrará la ventana secundaria asociada y la ventana previamente seleccionada (si es diferente) no se debe asignar. Si se omite *tab_id*, retorna el nombre del widget del panel seleccionado actualmente.

tab (*tab_id*, *option=None*, ***kw*)

Consultar o modificar las opciones del *tab_id* específico.

Si no se proporciona *kw*, retorna un diccionario de los valores de las opciones de pestañas. Si se especifica *option*, retorna el valor de esa *option*. De lo contrario, establece las opciones en los valores correspondientes.

tabs()

Retorna una lista de ventanas administradas por el notebook.

enable_traversal()

Habilita la tabulación para una ventana de nivel superior que contenga este notebook.

Esto extenderá los enlaces para la ventana de nivel superior que contiene el notebook del siguiente modo:

- **Control-Tab**: selecciona la pestaña siguiente a la seleccionada actualmente.
- **Shift-Control-Tab**: selecciona la pestaña precedente a la seleccionada actualmente.
- **Alt-K**: donde *K* es el carácter mnemotécnico (subrayado) de cualquier pestaña, seleccionará esa pestaña.

Se pueden habilitar varios notebooks en un único nivel superior para la tabulación, incluidos los notebooks anidados. Sin embargo, la tabulación entre notebooks solo funciona correctamente si todos los paneles tienen el notebook en el que se encuentran como maestro.

25.2.7 Progressbar

El widget `ttk.Progressbar` muestra el estado de una operación de larga duración. Puede funcionar en dos modos: 1) el modo determinado que muestra la cantidad completada en función de la cantidad total de trabajo a realizar y 2) el modo indeterminado que proporciona una pantalla animada para que el usuario sepa que la operación está en curso.

Opciones

Este widget acepta las siguientes opciones específicas:

Opción	Descripción
<code>orient</code>	Puede ser «horizontal» o «vertical». Especifica la orientación de la barra de progreso.
<code>length</code>	Especifica la longitud del eje largo de la barra de progreso (ancho si horizontal, alto si es vertical).
<code>mode</code>	Puede ser «determinate» o «indeterminate».
<code>maximum</code>	Número que especifica el valor máximo. Por defecto el valor es 100.
<code>value</code>	El valor actual de la barra de progreso. En el modo «determinado», representa la cantidad de trabajo completado. En el modo «indeterminado», se interpreta como módulo <i>maximum</i> , es decir, la barra de progreso completa un «ciclo» cuando su valor aumenta en <i>maximum</i> .
<code>variable</code>	Nombre vinculado al valor de la opción. Si se especifica, el valor de la barra de progreso se establece automáticamente en el valor de este nombre cada vez que se modifica dicho valor.
<code>phase</code>	Variable de solo lectura. El widget incrementa periódicamente el valor de esta opción siempre que su valor sea mayor que 0 y, en modo determinado, menor que el máximo. Esta opción puede ser utilizada por el tema actual para proporcionar efectos de animación adicionales.

ttk.Progressbar

class tkinter.ttk.Progressbar

start (*interval=None*)

Inicia el modo de incremento automático: programa un evento *timer* periódico que llama a `Progressbar.step()` cada *interval* milisegundos. Si se omite, *interval* tiene como valor predeterminado 50 milisegundos.

step (*amount=None*)

Incrementa el valor de la barra de progreso en *amount*.

amount vale 1.0 por defecto si se omite.

stop ()

Para el modo de incremento automático: cancela cualquier evento *timer* periódico iniciado por `Progressbar.start()` para la barra de progreso en cuestión.

25.2.8 Separator

El widget `ttk.Separator` muestra una barra de separación horizontal o vertical.

No tiene otros métodos aparte de aquellos heredados de `ttk.Widget`.

Opciones

Este widget acepta las opción específica siguiente:

Opción	Descripción
orient	Puede ser «horizontal» o «vertical». Especifica la orientación del separador.

25.2.9 Sizegrip

El widget `ttk.Sizegrip` (también conocido como *grow box*) permite al usuario cambiar el tamaño de la ventana de nivel superior que lo contiene presionando y arrastrando la esquina.

Este widget no tiene ni opciones ni métodos específicos, aparte de aquellos heredados de `ttk.Widget`.

Notas específicas por plataforma

- En MacOS X, las ventanas de nivel superior incluyen automáticamente un cambio de tamaño integrado de forma predeterminada. Agregar un `Sizegrip` es poco útil, ya que el cambio de tamaño integrado quedará encima del widget.

Errores detectados

- Si la posición del nivel superior que lo contiene se especifica en relación con la parte derecha o inferior de la pantalla (por ejemplo,...), el widget `Sizegrip` no cambiará el tamaño de la ventana.
- El widget solo soporta el cambio de tamaño desde la esquina inferior derecha.

25.2.10 Treeview

El widget `ttk.Treeview` muestra una colección en árbol de elementos. Cada elemento tiene una etiqueta textual, una imagen opcional y una lista opcional de valores de datos. Los valores de datos se muestran en columnas sucesivas después de la etiqueta de árbol.

El orden en que se muestran los valores de datos se puede controlar estableciendo la opción de widget `displaycolumns`. El Treeview también puede mostrar encabezados. Se puede acceder a las columnas por número o nombres simbólicos enumerados en las columnas de opciones del widget. Consulte [Column Identifiers](#).

Cada elemento se identifica con un nombre único. El widget genera IDs para los elementos si no se proporcionan en la declaración. Hay un elemento raíz distinguido, denominado `{ }`. El elemento raíz en sí no se muestra; sus hijos aparecen en el nivel superior de la jerarquía.

Cada elemento también tiene una lista de etiquetas que se pueden usar para asociar enlaces de eventos con elementos individuales y controlar la apariencia del elemento.

El widget Treeview admite el deslizamiento horizontal y vertical, según las opciones descritas en [Scrollable Widget Options](#) y los métodos `Treeview.xview()` y `Treeview.yview()`.

Opciones

Este widget acepta las siguientes opciones específicas:

Opción	Descripción
<code>columns</code>	Una lista de identificadores de columna, que especifican el número de columnas y sus nombres.
<code>displaycolumns</code>	Una lista de identificadores de columna (índices simbólicos o enteros) que especifican qué columnas de datos se muestran y el orden en que aparecen, o la cadena «#all».
<code>height</code>	Especifica el número de filas que deben estar visibles. Nota: el ancho solicitado se determina a partir de la suma de los anchos de columna.
<code>padding</code>	Especifica el relleno interno del widget. El relleno es una lista de hasta cuatro especificaciones de longitud.
<code>selectmode</code>	Controla cómo los enlaces de clase integrados administran la selección. Puede ser «extended», «browse» o «none». Si se establece en «extended» (valor predeterminado), se pueden seleccionar varios elementos. Si se elige «browse», se seleccionará un solo elemento a la vez. Si se elige «none», la selección no se cambiará. Tenga en cuenta que el código de la aplicación y los enlaces de etiqueta pueden establecer la selección como deseen, independientemente del valor de esta opción.
<code>show</code>	Una lista que contiene cero o más de los siguientes valores, especificando qué elementos del árbol se van a mostrar. <ul style="list-style-type: none">• <code>tree</code>: muestra las etiquetas del árbol en la columna #0.• <code>headings</code>: muestra la fila de encabezado. El valor predeterminado es «tree headings», es decir, mostrar todos los elementos. Nota: la columna #0 siempre hace referencia a la columna del árbol, incluso si no se especifica <code>show=»tree»</code> .

Opciones de elementos

Las siguientes opciones de elemento se pueden especificar para los elementos de los comandos de inserción y de elementos del widget.

Opción	Descripción
text	La etiqueta textual que se va a mostrar para el elemento.
image	Una imagen Tk, que se muestra a la izquierda de la etiqueta.
values	La lista de valores asociados al elemento. Cada elemento debe tener el mismo número de valores que las columnas de opciones del widget. Si hay menos valores que columnas, los valores restantes se asumen vacíos. Si hay más valores que columnas, se omiten los valores adicionales.
open	Valor <code>True/False</code> que indica si los elementos secundarios deben mostrarse u ocultarse.
tags	La lista de etiquetas asociadas al elemento.

Opciones de etiqueta

Se pueden especificar las opciones siguientes para etiquetas:

Opción	Descripción
foreground	Especifica el color de primer plano del texto.
background	Especifica el color de fondo de la celda o elemento.
font	Especifica la fuente que se utilizará al añadir texto.
image	Especifica la imagen del elemento, en caso de que la opción de imagen del elemento esté vacía.

Identificadores de columna

Los identificadores de columna toman cualquiera de los siguientes formas:

- Un nombre simbólico de la lista de opciones de columna.
- Un entero `n`, especificando la `n`-ésima columna de datos.
- Una cadena de la forma `#n`, donde `n` es un entero, especificando la `n`-ésima columna mostrada.

Notas:

- Los valores de opciones del elemento se pueden mostrar en un orden diferente al orden en el que se almacenan.
- La columna `#0` siempre hace referencia a la columna del árbol, incluso si no se especifica `show=»tree«`.

Un número de columna de datos es un índice en la lista de valores de opciones de un elemento; el número de columna se visualiza en el árbol donde se muestran los valores. Las etiquetas de árbol se muestran en la columna `#0`. Si no se establece la opción `displaycolumns`, la columna de datos `n` se muestra en la columna `#n+1`. Una vez más, **la columna `#0` siempre hace referencia a la columna del árbol.**

Eventos virtuales

El widget Treeview genera los siguientes eventos virtuales.

Evento	Descripción
«TreeviewSelect»	Se genera cada vez que cambia la selección.
«TreeviewOpen»	Generado justo antes configurar el elemento de resalto a <code>open=True</code> .
«TreeviewClose»	Generado justo después de establecer el elemento de resalto a <code>open=False</code> .

Los métodos `Treeview.focus()` y `Treeview.selection()` se pueden utilizar para determinar el elemento o elementos afectados.

ttk.Treeview

```
class tkinter.ttk.Treeview
```

bbox (*item*, *column=None*)

Retorna el cuadro delimitador (en relación con la ventana del widget de Treeview) del *item* especificado con el formato (x, y, ancho, alto).

Si se especifica *column*, retorna el cuadro delimitador de esa celda. Si el *item* no está visible (es decir, si es descendiente de un elemento cerrado o se desplaza fuera de la pantalla), retorna una cadena vacía.

get_children (*item=None*)

Retorna la lista de elementos secundarios que pertenecen a *item*.

Si no se especifica *item*, retorna la raíz.

set_children (*item*, **newchildren*)

Reemplaza el elemento secundario de *item* por *newchildren*.

Los elementos secundarios presentes en *item* que no están presentes en *newchildren* se separan del árbol. Ningún elemento de *newchildren* puede ser un antecesor de *item*. Tenga en cuenta que si no se especifica *newchildren* se desasocian los elementos secundarios de *item*.

column (*column*, *option=None*, ***kw*)

Consultar o modificar las opciones para la *columna* especificada.

Si no se proporciona *kw*, retorna un diccionario de los valores de opciones de columna. Si se especifica *option*, se retorna el valor de esa *option*. De lo contrario, establece las opciones en los valores correspondientes.

Las opciones/valores válidos son:

- **id** Retorna el nombre de la columna. Esta es una opción de solo lectura.
- **ancla: Uno de los valores de ancla Tk estándar.** Especifica cómo se debe alinear el texto de esta columna con respecto a la celda.
- **minwidth: width** El ancho mínimo de la columna en píxeles. El widget Treeview no hará que la columna sea más pequeña de lo especificado por esta opción cuando se cambie el tamaño del widget o el usuario arrastre una columna.
- **stretch: True/False** Especifica si el ancho de la columna debe ajustarse cuando se cambia el tamaño del widget.
- **width: ancho** El ancho de la columna en píxeles.

Se puede establecer *column = «#0»* para configurar el árbol de la columna.

delete (**items*)

Elimina todos los *items* especificados y todos sus descendientes.

El elemento raíz no se elimina.

detach (*items)

Desvincula todos los *items* especificados del árbol.

Los objetos y todos sus descendientes todavía existen, y pueden ser reinsertados en otro punto del árbol, pero no se mostrarán.

El elemento raíz no se desvincula.

exists (item)

Retorna `True` si el *item* especificado está presente en el árbol.

focus (item=None)

Si se especifica *item*, establece el elemento de foco en *item*. De lo contrario, retorna el elemento de foco actual o `""` si no hay ninguno.

heading (column, option=None, **kw)

Consulta o modifica las opciones de encabezado para la *column* especificada.

Si no se proporciona *kw*, retorna un diccionario de los valores de opciones de encabezado. Si se especifica *option*, se retorna el valor de esa *option*. De lo contrario, establece las opciones en los valores correspondientes.

Las opciones/valores válidos son:

- **text: texto** El texto se muestra en el encabezado de la columna.
- **image: imageName** Especifica una imagen para mostrar en la parte derecha del encabezado de la columna.
- **anchor: ancla** Especifica el alineamiento del texto del encabezado. Es uno de los valores estándar de Tk anchor.
- **command: callback** Una retrollamada que se ejecutará cuando se presione la etiqueta de encabezado.

Se especifica *column* =>#0» para configurar el encabezado de la columna de árbol.

identify (component, x, y)

Retorna una descripción del *component* especificado bajo el punto dado por *x* e *y*, o la cadena vacía si no existe dicho *component* en esa posición.

identify_row (y)

Retorna el identificador del elemento en la posición *y*.

identify_column (x)

Retorna el identificador de los datos de columna de la celda en la posición *x*.

La columna del árbol tiene ID #0.

identify_region (x, y)

Retorna uno de:

zona	significado
heading	Zona de encabezado del árbol.
separator	Espacio entre dos encabezados de columna.
tree	La zona del árbol.
cell	Datos de celda.

Disponibilidad: Tk 8.6.

identify_element (x, y)

Retorna el elemento en la posición *x*, *y*.

Disponibilidad: Tk 8.6.

index (item)

Retorna el índice entero de *item* dentro de la lista de elementos secundarios de su elemento primario.

insert (*parent*, *index*, *iid=None*, ***kw*)

Crea un nuevo elemento y retorna el identificador del elemento recién creado.

parent es el identificador del elemento primario o la cadena vacía para crear un nuevo elemento de nivel superior. *index* es un entero, o el valor «end», especificando dónde insertar el nuevo elemento en la lista de elementos secundarios del elemento primario. Si *index* es menor o igual que cero, el nuevo nodo se inserta al principio; si *index* es mayor o igual que el número actual de elementos secundarios, se inserta al final. Si se especifica *iid*, se utiliza como identificador de elemento; *iid* no debe existir en el árbol previamente. De lo contrario, se genera un nuevo identificador único.

Consulte [Item Options](#) para obtener la lista de puntos disponibles.

item (*item*, *option=None*, ***kw*)

Consulta o modifica las opciones para el *item* especificado.

Si no se proporciona ninguna opción, se retorna un diccionario con opciones/valores para el elemento. Si se especifica *option*, se retorna el valor de esa opción. De lo contrario, establece las opciones en los valores correspondientes según *kw*.

move (*item*, *parent*, *index*)

Mueve *item* a la posición *index* en la lista de elementos secundarios de *parent*.

No se permite mover un elemento bajo uno de sus descendientes. Si *index* es menor o igual que cero, *item* se mueve al principio; si es mayor o igual que el número de hijos, se mueve hasta el final. Si *item* se desvincula, se vuelve a conectar.

next (*item*)

Retorna el identificador del siguiente elemento de *item*, o "" si *item* es el último elemento secundario de su elemento primario.

parent (*item*)

Retorna el identificador del elemento primario de *item* o "" si *item* está en el nivel superior de la jerarquía.

prev (*item*)

Retorna el identificador del elemento anterior de *item*, o "" si *item* es el primer elemento secundario de su elemento primario.

reattach (*item*, *parent*, *index*)

Un alias para [Treeview.move\(\)](#).

see (*item*)

Garantiza que *item* está visible.

Establece todas las opciones modificables de los antecesores de *item* a True y desplaza el widget si es necesario para que *item* esté dentro de la parte visible del árbol.

selection ()

Retorna una tupla de los elementos seleccionados.

Distinto en la versión 3.8: [selection\(\)](#) ya no toma argumentos. Para cambiar el estado de selección, utiliza los siguientes métodos de selección.

selection_set (**items*)

items se convierte en la nueva selección.

Distinto en la versión 3.6: *items* se pueden pasar como argumentos separados, no solo como una sola tupla.

selection_add (**items*)

Añade *items* a la selección.

Distinto en la versión 3.6: *items* se pueden pasar como argumentos separados, no solo como una sola tupla.

selection_remove (**items*)

Elimina *elementos* de la selección.

Distinto en la versión 3.6: *items* se pueden pasar como argumentos separados, no solo como una sola tupla.

selection_toggle (**items*)

Alterna el estado de selección de cada elemento en *items*.

Distinto en la versión 3.6: *items* se pueden pasar como argumentos separados, no solo como una sola tupla.

set (*item*, *column=None*, *value=None*)

Con un argumento, retorna un diccionario de pares columna/valor para el *item* especificado. Con dos argumentos, retorna el valor actual de la *column* especificada. Con tres argumentos, establece el valor de determinado *column* en un *item* determinado en el *value* especificado.

tag_bind (*tagname*, *sequence=None*, *callback=None*)

Enlaza una retrollamada para el evento *sequence* a la etiqueta *tagname*. Cuando se entrega un evento a un elemento, se llama a las retrollamadas de cada una de las etiquetas del elemento.

tag_configure (*tagname*, *option=None*, ***kw*)

Consulta o modifica las opciones para el *tagname* especificado.

Si no se proporciona *kw*, retorna un diccionario de la configuración de opciones para *tagname*. Si se especifica *option*, retorna el valor de esa *option* para el *tagname* especificado. De lo contrario, establece las opciones en los valores correspondientes para el *tagname* dado.

tag_has (*tagname*, *item=None*)

Si se especifica *item*, retorna 1 o 0 dependiendo de si el *item* tiene el *tagname* especificado. De lo contrario, retorna una lista de todos los elementos que tienen la etiqueta especificada.

Disponibilidad: Tk 8.6

xview (**args*)

Consulta o modifica la posición horizontal de la vista de árbol.

yview (**args*)

Consulta o modifica la posición vertical de la vista de árbol.

25.2.11 Ttk Styling

A cada widget en `ttk` se le asigna un estilo, que especifica el conjunto de elementos que componen el widget y cómo se organizan, junto con la configuración dinámica y predeterminada para las opciones de elemento. De forma predeterminada, el nombre del estilo es el mismo que el nombre de clase del widget, pero puede ser reemplazado por la opción de estilo del widget. Si no conoces el nombre de clase de un widget, utiliza el método `Misc.winfo_class()` (`somewidget.winfo_class()`).

Ver también:

Presentación de la conferencia Tcl'2004 Este documento explica cómo funciona el motor de temas

class `tkinter.ttk.Style`

Esta clase se utiliza para manipular la base de datos de estilos.

configure (*style*, *query_opt=None*, ***kw*)

Consulta o establece el valor predeterminado de las opciones especificadas en *style*.

Cada clave en *kw* es una opción y cada valor es una cadena que identifica el valor de esa opción.

Por ejemplo, para cambiar cualquier botón por defecto a un botón plano con borde interno y un color de fondo distinto:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()
```

(continué en la próxima página)

(proviene de la página anterior)

```
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*)Consulta o establece valores dinámicos de opciones específicas en *style*.

Cada clave en *kw* es una opción y cada valor es una lista o tupla (generalmente) que contiene *statespecs* agrupados en tuplas, listas o alguna otra preferencia. Una *statespec* es un compuesto de uno o más estados y un valor.

Un ejemplo puede hacerlo más comprensible:

```
import tkinter
from 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()

root.mainloop()
```

Ten en cuenta que el orden de las secuencias (estado, valor) para una opción es importante, si se cambia el orden a `[('active', 'blue'), ('pressed', 'red')]` en la opción en primer plano, por ejemplo, el resultado sería un primer plano azul cuando el widget se encuentre en los estados *active* o *pressed*.

lookup (*style*, *option*, *state=None*, *default=None*)Retorna el valor especificado para *option* en *style*.

Si *state* se especifica, se espera que sea una secuencia de uno o más estados. Si se establece el argumento *default*, se usa como valor de reserva en caso de que no se especifique una opción.

Para verificar qué fuente usa un Button por defecto:

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout (*style*, *layoutspect=None*)

Define el diseño del widget para un *estilo* dado. Si se omite *layoutspect*, retorna la especificación de diseño para un estilo determinado.

layoutspect, si se especifica, se espera que sea una lista o algún otro tipo de secuencia (excluyendo cadenas), donde cada elemento debe ser una tupla y el primer elemento es el nombre de diseño y el segundo elemento debe tener el formato descrito en [Layouts](#).

Para entender el formato, consulta el siguiente ejemplo (no está pensado para hacer nada útil):

```
from tkinter import ttk
import tkinter
```

(continué en la próxima página)

(proviene de la página anterior)

```

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*)

Crea un nuevo elemento en el tema actual, del *etype* dado, que se espera que sea «image», «from» o «vsapi». Este último solo está disponible en Tk 8.6a para Windows XP y Vista y no se describe aquí.

Si se utiliza «image», *args* debe contener el nombre de imagen predeterminado seguido de pares *statespec*/value (esta es *imagespec*), y *kw* puede tener las siguientes opciones:

- **border=padding** el relleno interno es una lista de hasta cuatro enteros, especificando los bordes izquierdo, superior, derecho e inferior, respectivamente.
- **height=height** Especifica una altura mínima para el elemento. Si es menor que cero, la altura de la imagen base se utiliza como valor predeterminado.
- **padding=padding** Especifica el relleno interior del elemento. El valor predeterminado es el valor del borde si no se especifica.
- **sticky=spec** Especifica cómo se coloca la imagen dentro de la sección. *spec* contiene cero o más caracteres «n», «s», «w» o «e».
- **width=width** Especifica un ancho mínimo para el elemento. Si es menor que cero, el ancho de la imagen base se utiliza como valor predeterminado.

Si se utiliza «from» como valor de *etype*, *element_create()* clonará un elemento existente. *args* contiene un *themenam*e, desde el que se clonará el elemento y, opcionalmente, un elemento desde el que clonar. Si no se especifica este elemento para clonar, se usará un elemento vacío. *kw* se descarta.

element_names ()

Retorna la lista de elementos definidos en el tema actual.

element_options (*elementname*)

Retorna la lista de opciones de *elementname*.

theme_create (*themenam*e, *parent*=None, *settings*=None)

Crea un tema nuevo.

Es un error si *themenam*e ya existe. Si se especifica *parent*, el nuevo tema heredará estilos, elementos y diseños del tema primario. Si se especifica *settings*, se espera que tengan la misma sintaxis utilizada para *theme_settings()*.

theme_settings (*themenam*e, *settings*)

Establece temporalmente el tema actual en *themenam*e, aplica los *settings* especificados y, a continuación, restaura el tema anterior.

Cada clave en *settings* es un estilo y cada valor puede contener las teclas “configure”, “map”, “layout” y “element create” y se espera que tengan el mismo formato especificado por los métodos *Style.configure()*, *Style.map()*, *Style.layout()* y *Style.element_create()* respectivamente.

Como ejemplo, vamos a cambiar un poco el Combobox por el tema predeterminado:

```
from tkinter 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")]
        }
    }
})

combo = ttk.Combobox().pack()

root.mainloop()
```

theme_names()

Retorna una lista de temas conocidos.

theme_use (*themename=None*)

Si no se proporciona *themename*, retorna el tema en uso. De lo contrario, establece el tema actual en *themename*, actualiza todos los widgets y emite un evento «ThemeChanged».

Diseños

Un diseño puede ser simplemente `None`, si no tiene opciones, o un diccionario de opciones que especifica cómo organizar el elemento. El mecanismo de diseño utiliza una versión simplificada del paquete de gestión de geometría: dada una cavidad inicial, a cada elemento se le asigna una sección. Las opciones/valores válidos son:

- **side: whichside** Especifica en qué lado de la cavidad colocar el elemento; sea arriba, derecha, abajo o izquierda. Si se omite, el elemento ocupa toda la cavidad.
- **sticky: nswe** Especifica dónde se coloca el elemento dentro de su sección asignada.
- **unit: 0 or 1** Si se establece en 1, hace que el elemento y todos sus descendientes sean tratados como un único elemento en métodos como `Widget.identify()` y otros. Se utiliza para cosas como los *thumbs* de la barra de desplazamiento.
- **children: [sublayout...]** Especifica una lista de elementos para colocar dentro del elemento. Cada elemento es una tupla (u otro tipo de secuencia) donde el primer elemento es el nombre de diseño y el otro es un *Layout*.

25.3 `tkinter.tix` — Ampliación de widgets para Tk

Código fuente: `Lib/tkinter/tix.py`

Obsoleto desde la versión 3.6: Esta ampliación de Tk no está mantenida y no debe ser usada en nuevo código. Use `tkinter.ttk` en su lugar.

El módulo `tkinter.tix` (*Tk Interface Extension* en inglés) proporciona un conjunto abundante de widgets adicionales. Aunque la biblioteca estándar Tk tiene muchos widgets útiles, están lejos de ser completos. La biblioteca `tkinter.tix` proporciona la mayoría de los widgets comúnmente necesarios que no están en el estándar Tk: `HList`, `ComboBox`, `Control` (alias `SpinBox`) y una selección de widgets desplazables. `tkinter.tix` también incluye muchos más widgets que son generalmente útiles en un rango amplio de aplicaciones: `NoteBook`, `FileEntry`, `PanedWindow`, etc; hay más de 40 de ellos.

Con todos estos nuevos widgets, puedes introducir nuevas técnicas de interacción en aplicaciones, creando interfaces de usuario más útiles y más intuitivas. Puedes diseñar tu aplicación al escoger los widgets más apropiados que combinen con las necesidades especiales de tu aplicación y usuarios.

Ver también:

Tix Homepage La página de inicio de Tix. Incluye enlaces a documentación adicional y descargas.

Tix Man Pages Versión en línea de las páginas del manual y material de referencia.

Tix Programming Guide Versión en línea del material de referencia del programador.

Tix Development Applications Aplicaciones de Tix para el desarrollo de programas de Tix y Tkinter. Las aplicaciones de *Tide* (por *Tix Integrated Development Environment* en sus siglas de inglés) trabajan bajo Tk o Tkinter, e incluyen **TixInspect**, un inspector para modificar y depurar aplicaciones Tix/Tk/Tkinter remotamente.

25.3.1 Usando Tix

class `tkinter.tix.Tk` (`screenName=None`, `baseName=None`, `className='Tix'`)

Un widget de alto nivel de Tix que representa generalmente la ventana principal de una aplicación. Tiene un intérprete `Tcl` asociado.

Las clases en el módulo `tkinter.tix` heredan de la clases en `tkinter`. El primero importa el segundo, por lo que para usar `tkinter.tix` con Tkinter, todo lo que necesitas hacer es importar un módulo. En general, puedes importar `tkinter.tix`, y reemplazar las invocaciones de alto nivel a `tkinter.Tk` con `tix.Tk`:

```
from tkinter import tix
from tkinter.constants import *
root = tix.Tk()
```

Para usar `tkinter.tix`, debes tener los widgets Tix instalados, usualmente junto a tu instalación de los widgets Tk. Para probar tu instalación, intenta lo siguiente:

```
from tkinter import tix
root = tix.Tk()
root.tk.eval('package require Tix')
```

25.3.2 Widgets de Tix

Tix introduce más de 40 clases de widget al repertorio de *tkinter*.

Widgets Básicos

class `tkinter.tix.Balloon`

Un **Globo** que aparece sobre un widget para proporcionar ayuda. Cuando el usuario mueve el cursor dentro de un widget al cual el widget Globo está ligado, una pequeña ventana emergente con un mensaje descriptivo se mostrará en la pantalla.

class `tkinter.tix.ButtonBox`

El widget **ButtonBox** crea una caja de botones, tal como es comúnmente usado para Ok Cancelar.

class `tkinter.tix.ComboBox`

El widget **ComboBox** es similar al control de caja de selección combinada en MS Windows. El usuario puede seleccionar una opción escribiendo en el *subwidget* de entrada o seleccionando de la caja de lista del subwidget.

class `tkinter.tix.Control`

El widget **Control** es también conocido como el widget **SpinBox**. El usuario puede ajustar el valor al presionar los dos botones de flecha o al ingresar el valor directamente en la entrada. El nuevo valor será comparado con los límites superiores e inferiores definidos por el usuario.

class `tkinter.tix.LabelEntry`

El widget **LabelEntry** empaqueta un widget de entrada y una etiqueta en un mega widget. Puede ser usado para simplificar la creación de los tipos de interfaz de «formularios de entrada».

class `tkinter.tix.LabelFrame`

El widget **LabelFrame** empaqueta un widget *frame* y una etiqueta en un mega widget. Para crear widgets dentro de un widget **LabelFrame**, uno crea los nuevos widgets relativos al *subwidget frame* y los gestiona dentro del subwidget *frame*.

class `tkinter.tix.Meter`

El widget **Meter** puede ser usado para mostrar el progreso de un trabajo en segundo plano que puede tomar un largo tiempo de ejecución.

class `tkinter.tix.OptionMenu`

El widget **OptionMenu** crea un botón de menú de opciones.

class `tkinter.tix.PopupMenu`

El widget **PopupMenu** puede ser usado como un reemplazo del comando `tk_popup`. La ventaja del widget **PopupMenu** de Tix es que requiere menos código de aplicación para manipular.

class `tkinter.tix.Select`

El widget **Select** es un contenedor de *subwidgets* botón. Puede ser usado para proporcionar opciones de selección de estilos radio-box o check-box para el usuario.

class `tkinter.tix.StdButtonBox`

El widget **StdButtonBox** es un grupo de botones estándares para cajas de diálogo similares a *Motif*.

Selectores de Archivos

class `tkinter.tix.DirList`

El widget **DirList** muestra una vista de lista de un directorio, sus directorios previos, y sus sub-directorios. El usuario puede mostrar uno de los directorios mostrados en la lista o cambiar a otro directorio.

class `tkinter.tix.DirTree`

El widget **DirTree** muestra una vista de árbol de un directorio, sus directorios previos y sus sub-directorios. El usuario puede escoger uno de los directorios mostrados en la lista o cambiar a otro directorio.

class `tkinter.tix.DirSelectDialog`

El widget **DirSelectDialog** presenta los directorios en el sistema de archivos en una ventana de diálogo. El

usuario puede usar esta ventana de diálogo para navegar a través del sistema de archivos para seleccionar el directorio deseado.

class `tkinter.tix.DirSelectBox`

El widget `DirSelectBox` es similar al cuadro de selección de directorio estándar de Motif(TM). Es generalmente usado para que el usuario escoja un directorio. `DirSelectBox` guarda los directorios seleccionados recientemente en un widget de cuadro combinado para que puedan ser seleccionados rápidamente de nuevo.

class `tkinter.tix.ExFileSelectBox`

El widget `ExFileSelectBox` es usualmente embebido en un widget `tixExFileSelectDialog`. Proporciona un método conveniente para que el usuario seleccione archivos. El estilo del widget `ExFileSelectBox` es muy similar al diálogo de archivo estándar en MS Windows 3.1.

class `tkinter.tix.FileSelectBox`

El widget `FileSelectBox` es similar al cuadro de selección de archivo estándar de Motif(TM). Es generalmente usado para que el usuario escoja un archivo. `FileSelectBox` guarda los archivos recientemente seleccionados en un widget `ComboBox` para que puedan ser rápidamente seleccionados de nuevo.

class `tkinter.tix.FileEntry`

El widget `FileEntry` puede ser usado para ingresar un nombre de archivo. El usuario puede tipear el nombre de archivo manualmente. Alternativamente, el usuario puede presionar el widget de botón que está al lado de la entrada, que mostrará un diálogo de selección de archivo.

ListBox jerárquico

class `tkinter.tix.HList`

El widget `HList` puede ser usado para mostrar cualquier dato que tenga una estructura jerárquica, por ejemplo, árboles del directorio del sistema de archivos. Las entradas de las líneas son sangradas y conectadas por líneas de ramas de acuerdo a sus lugares en la jerarquía.

class `tkinter.tix.CheckList`

El widget `CheckList` muestra una lista de objetos a ser seleccionados por el usuario. `CheckList` actúa de forma similar a los widget Tk `checkboxbutton` o `radiobutton`, excepto que es capaz de manejar muchos más objetos que los widgets `checkboxbutton` o `radiobutton`.

class `tkinter.tix.Tree`

Se puede usar el widget `Tree` para mostrar datos jerárquicos en una forma de árbol. El usuario puede ajustar la vista del árbol al abrir o cerrar partes del árbol.

ListBox Tabular

class `tkinter.tix.TList`

Se puede usar el widget `TList` para mostrar datos en un formato tabular. Las entradas de lista de un widget `TList` son similares a las entradas del widget `listbox`. Las principales diferencias son (1) el widget `TList` puede mostrar las entradas de la lista en un formato de dos dimensiones y (2) puedes usar imágenes gráficas así como también como múltiples colores y fuentes para las entradas de lista.

Gestores de Widgets

class `tkinter.tix.PanedWindow`

El widget `PanedWindow` permite que el usuario manipule interactivamente los tamaños de varios paneles. Los paneles pueden ser ordenados o verticalmente u horizontalmente. El usuario cambia los tamaños de los paneles al arrastrar el asa de redimensionamiento entre dos paneles.

class `tkinter.tix.ListNoteBook`

El widget `ListNoteBook` es muy similar al widget Tk `NoteBook`: puede ser usado para mostrar muchas ventanas en un espacio limitado usando la metáfora del cuaderno (*notebook*). El cuaderno es dividido en una pila de páginas (ventanas). Sólo una de estas páginas se muestra a la vez. El usuario puede navegar a través de estas páginas al escoger el nombre de la página deseada en el *subwidget* `hlist`.

class `tkinter.tix.NoteBook`

Se puede usar el widget `NoteBook` para mostrar muchas ventanas en un espacio limitado usando la metáfora del cuaderno (*notebook*). El *notebook* es dividido en una pila de páginas. Sólo una de estas páginas se puede mostrar a la vez. El usuario puede navegar a través de estas páginas al escoger las «pestañas» visuales arriba del widget *NoteBook*.

Tipos de Imágenes

El módulo `tkinter.tix` añade:

- capacidades de `bitmap` a todos los widgets de `tkinter.tix` y `tkinter` para crear imágenes de color a partir de archivos XPM.
- se pueden usar los tipos de imágenes `Compound` para crear imágenes que consisten en múltiples líneas horizontales; cada línea es compuesta de una serie de objetos (texto, bitmaps, imágenes, o espacios) ordenados de izquierda a derecha. Por ejemplo, una imagen compuesta puede ser usada para mostrar un bitmap y una cadena de texto simultáneamente en un widget `Tk.Button`.

Widgets Varios

class `tkinter.tix.InputOnly`

Los widgets `InputOnly` van a aceptar entradas del usuario, que pueden ser realizadas con el comando `bind` (sólo para Unix).

Gestor de Geometría de Formulario

Además, `tkinter.tix` mejora a `tkinter` al proporcionar:

class `tkinter.tix.Form`

El gestor de geometría de `Formulario` basado en reglas de anexo para todos los widgets `Tk`.

25.3.3 Comandos Tix

class `tkinter.tix.tixCommand`

Los comandos `tix` proporcionan acceso a elementos misceláneos del estado interno de `Tix` y del contexto de la aplicación de `Tix`. La mayoría de la información manipulada por estos métodos le pertenece a la aplicación en conjunto, o a una pantalla o monitor, en vez de una ventana en particular.

Para ver las configuraciones actuales, el uso común es:

```
from tkinter import tix
root = tix.Tk()
print(root.tix_configure())
```

tixCommand.tix_configure (*cnf=None, **kw*)

Consulta o modifica las opciones de configuración del contexto de la aplicación `Tix`. Si no se especifica ninguna opción, retorna un diccionario con todas las opciones disponibles. Si la opción es especificada sin ningún valor, entonces el método retorna una lista describiendo la opción nombrada (esta lista será idéntica a la sublista correspondiente de los valores retornados si no se especifica ninguna opción). Si uno o más pares opción-valor son especificados, entonces el método modifica las opciones dadas para tener los valores dados; en este caso el método retorna una cadena de caracteres vacía. La opción puede ser cualquiera de las opciones de configuración.

tixCommand.tix_cget (*option*)

Retorna el valor actual de la opción de configuración dada por *option*. La opción puede ser cualquiera de las opciones configurables.

`tixCommand.tix_getbitmap(name)`

Localiza un archivo bitmap con el nombre `name.xpm` o `name` en uno de los directorios bitmap (véase el método `tix_addbitmapdir()`). Al usar `tix_getbitmap()`, puedes evitar codificar directamente los nombres de ruta de los archivos bitmap en tu aplicación. Cuando tiene éxito, retorna el nombre de ruta completo del archivo bitmap, prefijado con el carácter @. El valor retornado puede ser usado para configurar la opción `bitmap` de los widgets de Tk y Tix.

`tixCommand.tix_addbitmapdir(directory)`

Tix mantiene una lista de directorios bajo los cuales los métodos `tix_getimage()` y `tix_getbitmap()` buscarán archivos de imágenes. El directorio de bitmap estándar es `$TIX_LIBRARY/bitmaps`. El método `tix_addbitmapdir()` añade `directory` a la lista. Al usar este método, los archivos de imagen de una aplicación pueden ser localizados usando el método `tix_getimage()` o `tix_getbitmap()`.

`tixCommand.tix_filedialog([dlgclass])`

Retorna el diálogo de selección de archivo que puede ser compartido entre diferentes invocaciones de esta aplicación. Este método creará una widget de diálogo de selección de archivos cuando es invocado la primera vez. Este diálogo será retornado por las subsiguientes invocaciones de `tix_filedialog()`. Un parámetro `dlgclass` opcional puede ser pasado como cadena para especificar qué tipo de widget de selección de diálogo es deseado. Las opciones posibles son `tix`, `FileSelectDialog`, o `tixExFileSelectDialog`.

`tixCommand.tix_getimage(self, name)`

Localiza un archivo de imagen con el nombre `name.xpm`, `name.xbm` o `name.ppm` en uno de los directorios de bitmap (véase el método `tix_addbitmapdir()` arriba). Si existe más de un archivo con el mismo nombre (pero con diferentes extensiones), entonces el tipo de imagen es escogido de acuerdo a la profundidad del monitor X: las imágenes xbm son escogidas en monitores monocromos e imágenes de color son escogidas en monitores de color. Al usar `tix_getimage()`, puedes evitar codificar de forma directa los nombres de ruta de los archivos de imagen en tu aplicación. Cuando tiene éxito, este método retorna el nombre de la imagen recién creada, que puede ser usada para configurar la opción `image` de los widgets Tk y Tix.

`tixCommand.tix_option_get(name)`

Obtiene las opciones mantenidas por el mecanismo de esquema de Tix.

`tixCommand.tix_resetoptions(newScheme, newFontSet[, newScmPrio])`

Reinicia el esquema y conjunto de fuentes de la aplicación Tix a `newScheme` y `newFontSet`, respectivamente. Afecta sólo a los widgets creados después de esta invocación. Por lo tanto, es mejor invocar al método `resetoptions` antes de la creación de cualquier widget en una aplicación Tix.

Se le puede dar el parámetro opcional `newScmPrio` para reiniciar el nivel de prioridad de las opciones de Tk definidas por los esquemas de Tix.

Debido a la manera en que Tk gestiona la opción de la base de datos X, después de que Tix se haya importado e inicializado, no es posible reiniciar el esquema de colores y conjunto de fuentes usando el método `tix_config()` método. En vez de eso, se debe usar el método `tix_resetoptions()`.

25.4 tkinter.scrolledtext — Widget de texto desplazado

Código fuente: [Lib/tkinter/scrolledtext.py](#)

El módulo `tkinter.scrolledtext` proporciona una clase del mismo nombre la cual implementa un widget de texto básico que tiene una barra de desplazamiento vertical para hacer lo «correcto.» Usar la clase `ScrolledText` es mucho más fácil que configurar un widget de texto y una barra de desplazamiento directamente. El constructor es el mismo que el de la clase `tkinter.Text`.

El widget de texto y la barra de desplazamiento se empaquetan juntos en un `Frame`, y los métodos de los administradores de geometría `Grid` y `Pack` se adquieren del objeto `Frame`. Esto permite que el widget `ScrolledText` sea usado directamente para lograr el funcionamiento más normal del administrador de geometría.

Si fuera necesario un control más específico, los siguientes atributos están disponibles:

`ScrolledText.frame`

El marco que rodea el texto y los widgets de la barra de desplazamiento.

`ScrolledText.vbar`

El widget de la barra de desplazamiento.

25.5 IDLE

Código fuente: [Lib/idlelib/](#)

IDLE es el entorno de desarrollo integrado de Python.

IDLE tiene las siguientes características:

- escrito 100% en Python puro, usando el kit de herramientas GUI *tkinter*
- multiplataforma: funciona en su mayoría igual en Windows, Unix y macOS
- La ventana del shell de Python (interprete interactivo) con coloreado de código de entrada, salida y mensajes de error
- editor de texto multiventana con deshacer múltiple, coloreación Python, indentado inteligente, sugerencias de llamadas a funciones, autocompletado y otras características
- búsqueda dentro de cualquier ventana, reemplazo dentro de las ventanas del editor, y búsqueda a través de múltiples archivos (grep)
- depurador con breakpoints persistentes, por pasos y visualización de espacios de nombres globales y locales
- configuración, navegadores y otros cuadros de diálogo

25.5.1 Menús

IDLE tiene dos tipos de ventana principales, la ventana del shell y la ventana del editor. Es posible tener múltiples ventanas de edición simultáneamente. En Windows y Linux, cada una tiene su propio menú principal. Cada menú documentado abajo indica con cuál tipo de ventana está asociada esta.

Las ventanas de salida, como las que se usan para Editar => Encontrar en archivos, son un subtipo de la ventana de edición. Actualmente tienen el mismo menú principal pero un título predeterminado y un menú contextual diferente.

En macOS, hay un menú de aplicación. Este cambia dinámicamente de acuerdo la ventana actualmente seleccionada. Tiene un menú IDLE y algunas de las entradas descritas a continuación se mueven de acuerdo con las pautas de Apple.

Menú de archivo (Shell y Editor)

Nuevo archivo Crea una nueva ventana de edición de archivos.

Abrir... Abre un archivo existente con un cuadro de dialogo para abrir.

Archivos recientes Abre una lista de archivos recientes. Haga click en alguno para abrirlo.

Abrir módulo... Abre un módulo existente (buscar en sys.path).

Navegador de clases Muestra las funciones, clases y métodos en estructura de árbol en el archivo actual del Editor.
En el shell, primero abra un módulo.

Navegador de ruta Muestra directorios, módulos, funciones, clases y métodos de sys.path en una estructura de árbol.

Guardar Guarda la ventana actual en el archivo asociado, si existe alguno. Las ventanas que han sido modificadas desde que se abrieron o se guardaron por última vez tienen un * antes y después del título de la ventana. Si no hay un archivo asociado, ejecute Guardar como en su lugar.

Guardar Como... Guarda la ventana actual con un cuadro de diálogo Guardar como. El archivo guardado se convierte en el nuevo archivo asociado para esta ventana.

Guardar copia como... Guarda la ventana actual en un archivo diferente sin cambiar el archivo asociado.

Imprimir ventana Imprime la ventana actual en la impresora predeterminada.

Cerrar Cierra la ventana actual (solicita guardar si no está guardado).

Salir Cierra todas las ventanas y sale de IDLE (solicita guardar ventanas no guardadas).

Menú editar (Shell y Editor)

Deshacer Deshace el último cambio a la ventana actual. Se puede deshacer un máximo de 1000 cambios.

Rehacer Vuelve a aplicar el último cambio deshecho a la ventana actual.

Cortar Copia la selección en el portapapeles global; después elimina la selección.

Copiar Copia la selección en el portapapeles global.

Pegar Inserta el contenido del portapapeles global en la ventana actual.

Las funciones del portapapeles también están disponibles en los menús contextuales.

Seleccionar todo Selecciona el contenido completo de la ventana actual.

Encontrar... Abre un cuadro de diálogo de búsqueda con muchas opciones

Buscar de nuevo Repite la última búsqueda, si hay alguna.

Encontrar selección Busca la cadena de caracteres seleccionada actualmente, si hay alguna.

Encontrar en Archivos... Abre un cuadro de diálogo de búsqueda de archivos. Presenta los resultados en una nueva ventana de salida.

Reemplazar... Abre un cuadro de diálogo de búsqueda y reemplazo.

Ir a la línea Mueve el cursor al inicio de la línea solicitada y hace que esa línea sea visible. Una solicitud a partir del final del archivo lo lleva al final. Borra cualquier selección y actualiza el estado de la línea y la columna.

Mostrar complementos Open a scrollable list allowing selection of existing names. See [Completions](#) in the Editing and navigation section below.

Expandir palabra Completa un prefijo que se ha escrito para que coincida con una palabra completa en la misma ventana; intente nuevamente para obtener un complemento diferente.

Mostrar sugerencias de llamada Después de un paréntesis abierto para una función, abre una pequeña ventana con sugerencias sobre parámetros de función. Consultar [Sugerencias de llamada](#) en la sección Edición y navegación a continuación.

Mostrar los paréntesis alrededor Resalta los paréntesis.

Menú de formato (solo ventana del Editor)

Zona de indentación Desplaza las líneas seleccionadas a la derecha en un nivel de indentación (por defecto 4 espacios).

Zona de deindentación Desplaza las líneas seleccionadas hacia la izquierda en un nivel de indentación (por defecto 4 espacios).

Zona comentada Inserta `##` delante de las líneas seleccionadas.

Zona descomentada Elimina los `#` o `##` al inicio de las líneas seleccionadas.

Zona tabulada Transforma los tramos de espacios *iniciales* en tabs. (Nota: Recomendamos usar 4 bloques de espacio para indentar el código de Python).

Zona destabulada Transforma *todos* los tabs en el número correcto de espacios.

Alternar tabs Abre un cuadro de diálogo para cambiar entre indentado con espacios y tabs.

Nuevo tamaño de indentación Abre un cuadro de diálogo para cambiar el tamaño de indentación. El valor predefinido aceptado por la comunidad de Python es de 4 espacios.

Formatear párrafo Reformatea el párrafo actual delimitado por líneas en blanco en un bloque de comentarios o una cadena de caracteres multilínea o una línea seleccionada en una cadena de caracteres. Todas las líneas en el párrafo estarán formateadas con menos de N columnas, donde N por defecto es 72.

Remover espacios en blanco al final Elimina el espacio final en la línea y otros caracteres de espacio en blanco después del último carácter que no sea un espacio en blanco aplicando `str.rstrip` a cada línea, incluyendo las líneas dentro de cadenas de caracteres multilíneas. Excepto para las ventanas de consola, elimina nuevas líneas adicionales al final del archivo.

Menú ejecutar (solo ventana Editor)

Módulo ejecutar Hace lo que dice en *Verificar módulo*. Si no hay errores, reinicia el shell para limpiar el entorno, luego ejecuta el módulo. La salida es mostrada en la ventana de shell. Tener en cuenta que la visualización requiere el uso de `print` o `write`. Cuando finaliza la ejecución, el shell permanece activo y muestra un mensaje. En este punto, se puede explorar interactivamente el resultado de la ejecución. Esto es similar a ejecutar un archivo con `python -i file` en una línea de comando.

Ejecutar... Personalizado Igual que *Módulo ejecutar*, pero ejecuta el módulo con parámetros personalizados. *Los argumentos de la línea de comandos* extienden `sys.argv` como si se pasaran por una línea de comando. El módulo se puede ejecutar en el shell sin reiniciar.

Verificar módulo Comprueba la sintaxis del módulo actualmente abierto en la ventana de edición. Si el módulo no ha sido guardado IDLE solicitará al usuario que guarde o guarde automáticamente, como se seleccionó en la pestaña General del cuadro de diálogo Configuración de inactividad. Si hay algún error de sintaxis, la ubicación aproximada será indicada en la ventana del Editor.

Shell de Python Abrir o activar la ventana del shell de Python.

Menú de shell (solo ventana de shell)

Ver el último reinicio Navega la ventana del shell hasta el último reinicio del mismo.

Reiniciar shell Restart the shell to clean the environment and reset display and exception handling.

Historial anterior Recorre los comandos anteriores en el historial que coinciden con la entrada actual.

Historial siguiente Recorre los comandos posteriores en el historial que coinciden con la entrada actual.

Ejecución de interrupción Detiene un programa en ejecución.

Menú de depuración (solo ventana de shell)

Ir al Archivo/Línea Busca en la línea actual, con el cursor y la línea de arriba para un nombre de archivo y número de línea. Si lo encuentra, abre el archivo si aún no está abierto y muestra la línea. Usa esto para ver las líneas de origen referenciadas en un rastreo de excepción y las líneas encontradas por Buscar en archivos. También disponible en el menú contextual de la ventana del shell y las ventanas de salida.

Depurador (alternar) Cuando esta función está habilitada, el código ingresado en el shell o ejecutado desde el editor se ejecutará con el depurador. En el editor, los breakpoints se pueden establecer con el menú contextual. Esta funcionalidad aún está incompleta y es en cierto modo experimental.

Visualizador de pila Muestra el seguimiento de la pila de la última excepción en un complemento de árbol, con acceso a locales y globales.

Auto-abrir visualizador de pila Activa/desactiva automáticamente el visualizador de pila en una excepción no controlada.

Menú de opciones (Shell y editor)

Configurar IDLE Abre un cuadro de diálogo de configuración y cambia las preferencias por lo siguiente: fuentes, indentación, combinaciones de teclas, temas de color de texto, ventanas y tamaño de inicio, fuentes de ayuda adicionales y extensiones. En macOS, abre el cuadro de diálogo de configuración seleccionando Preferencias en el menú de la aplicación. Para obtener más detalles, consultar [Configurar preferencias](#) en Ayuda y preferencias.

La mayoría de los ajustes de configuración se aplican a todas las ventanas, abiertas o no. Los siguientes elementos de opción se aplican solo a la ventana activa.

Mostrar/Ocultar el contexto del código (solo ventana del Editor) Abre un panel en la parte superior de la ventana de edición el cual muestra el contexto de bloque de código que se ha desplazado sobre la parte superior de la ventana. Consultar [Contexto de código](#) en la sección Edición y Navegación a continuación.

Mostrar/Ocultar números de línea (solo ventana del Editor) Abre una columna a la izquierda de la ventana de edición que muestra el número de cada línea de texto. El valor por defecto de esta característica es desactivado, este puede modificarse en las preferencias (consultar [Configuración de preferencias](#)).

Ampliar/Restaurar altura Alterna la ventana entre el tamaño normal y la altura máxima. El tamaño inicial predeterminado es 40 líneas por 80 caracteres a menos que se cambie en la pestaña General del cuadro de diálogo Configurar IDLE. La altura máxima para una pantalla es determinada maximizando momentáneamente una ventana la primera vez que se acerca la pantalla. Cambiar la configuración de la pantalla puede invalidar la altura guardada. Este alternado no tiene efecto cuando se maximiza una ventana.

Menú de ventana (shell y editor)

Enumera los nombres de todas las ventanas abiertas; seleccione uno para ponerlo en primer plano (deiconificándolo si es necesario).

Menú de ayuda (shell y editor)

Acerca de IDLE Muestra la versión, derechos de autor, licencia, créditos y más.

Ayuda de IDLE Muestra este documento IDLE, que detalla las opciones del menú, edición y navegación básica y otros consejos.

Documentación de Python Accede a la documentación local de Python, si está instalada, o inicia un navegador web y abre docs.python.org mostrando la última documentación de Python.

Demostración Turtle Ejecuta el módulo `turtledemo` con ejemplos de código Python y dibujos de tortugas.

Se pueden agregar fuentes de ayuda adicionales aquí con el cuadro de diálogo Configurar IDLE en la pestaña General. Consultar la subsección [Fuentes de ayuda](#) a continuación para obtener más información sobre las opciones del menú Ayuda.

Menús contextuales

Se abre un menú contextual haciendo click derecho en una ventana (Control-click en macOS). Los menús contextuales tienen las funciones estándar del portapapeles también en el menú Editar.

Cortar Copia la selección en el portapapeles global; después elimina la selección.

Copiar Copia la selección en el portapapeles global.

Pegar Inserta el contenido del portapapeles global en la ventana actual.

Las ventanas del editor también tienen funciones de breakpoint. Las líneas con un conjunto de breakpoints están especialmente marcadas. Los breakpoints solo tienen efecto cuando se ejecutan bajo el depurador. Los breakpoints para un archivo se guardan en el directorio `.idlerc` del usuario.

Establecer breakpoint Establecer un breakpoint en la línea actual.

Eliminar breakpoint Eliminar el breakpoint en esa línea.

Las ventanas de shell y de salida también tienen los siguientes elementos.

Ir a archivo/línea Hace lo mismo que el menú depurar.

La ventana de shell también tiene una función de *squeezing* de salida explicada en la subsección *ventana de shell de Python* a continuación.

Exprimir Si el cursor está sobre una línea de salida, comprime toda la salida entre el código de arriba y el mensaje de abajo hasta la etiqueta “Texto *squeezed*”.

25.5.2 Edición y navegación

Ventana del editor

IDLE puede abrir ventanas del editor cuando se inicia, dependiendo de la configuración y de cómo inicies el IDLE. A partir de ahí, usar el menú Archivo. Solo puede haber una ventana de editor abierta para un archivo determinado.

La barra de título contiene el nombre del archivo, la ruta completa y la versión de Python e IDLE que ejecuta la ventana. La barra de estado contiene el número de línea (“Ln”) y el número de columna (“Col”). Los números de línea comienzan con 1; los números de columna con 0.

El IDLE supone que los archivos con una extensión .py* conocida contienen código Python y que los otros archivos no. Ejecuta el código Python con el menú Ejecutar.

Atajos de teclado

En esta sección, “C” hace referencia a la tecla `Control` en Windows y Unix y la tecla `Command` en macOS.

- `Backspace` borra hacia la izquierda; `Del` borra hacia la derecha
- `C-Backspace` borra la palabra a la izquierda; `C-Del` borra la palabra a la derecha
- Las teclas con flechas y `Page Up/Page Down` para moverse alrededor
- `C-LeftArrow` y `C-RightArrow` para moverse por palabras
- `Home/End` para ir al inicio/fin de la línea
- `C-Home/C-End` para ir al inicio/fin del archivo
- Algunos atajos útiles de Emacs son heredados de Tcl / Tk:
 - `C-a` al inicio de la línea
 - `C-e` al final de la línea
 - `C-k` corta la línea (pero no la pone en el portapapeles)
 - `C-l` centra la ventana alrededor del punto de inserción
 - `C-b` retrocede un carácter sin eliminarlo (generalmente también se puede usar la tecla de cursor para esto)
 - `C-f` avanza un carácter sin eliminarlo (generalmente también se puede usar la tecla de cursor para esto)
 - `C-p` sube una línea (generalmente también se puede usar la tecla del cursor para esto)
 - `C-d` borra el siguiente carácter

Las combinaciones de teclas estándar (como `C-c` para copiar y `C-v` para pegar) pueden funcionar. Las combinaciones de teclas se seleccionan en el cuadro de diálogo Configurar IDLE.

Indentación automática

Después de una declaración de apertura de bloque, la siguiente línea está indentada por 4 espacios (en la ventana del shell de Python por un tab). Después de ciertas palabras clave (saltar, retornar, etc.), la siguiente línea se deindenta. En la indentación principal, `Backspace` elimina hasta 4 espacios si están allí. `Tab` inserta espacios (en la ventana del shell de Python un tab), el número depende del tamaño de la indentación. Actualmente, los tabs están restringidos a cuatro espacios debido a las limitaciones de Tcl/Tk.

Consulte también los comandos de zona de indexación/deindentación en [Menú de formato](#).

Terminaciones

Completions are supplied, when requested and available, for module names, attributes of classes or functions, or filenames. Each request method displays a completion box with existing names. (See tab completions below for an exception.) For any box, change the name being completed and the item highlighted in the box by typing and deleting characters; by hitting Up, Down, PageUp, PageDown, Home, and End keys; and by a single click within the box. Close the box with Escape, Enter, and double Tab keys or clicks outside the box. A double click within the box selects and closes.

One way to open a box is to type a key character and wait for a predefined interval. This defaults to 2 seconds; customize it in the settings dialog. (To prevent auto popups, set the delay to a large number of milliseconds, such as 100000000.) For imported module names or class or function attributes, type `“.”`. For filenames in the root directory, type `os.sep` or `os.altsep` immediately after an opening quote. (On Windows, one can specify a drive first.) Move into subdirectories by typing a directory name and a separator.

Instead of waiting, or after a box is closed, open a completion box immediately with Show Completions on the Edit menu. The default hot key is `C-space`. If one types a prefix for the desired name before opening the box, the first match or near miss is made visible. The result is the same as if one enters a prefix after the box is displayed. Show Completions after a quote completes filenames in the current directory instead of a root directory.

Hitting `Tab` after a prefix usually has the same effect as Show Completions. (With no prefix, it indents.) However, if there is only one match to the prefix, that match is immediately added to the editor text without opening a box.

Invoking “Show Completions”, or hitting `Tab` after a prefix, outside of a string and without a preceding `“.”` opens a box with keywords, builtin names, and available module-level names.

When editing code in an editor (as oppose to Shell), increase the available module-level names by running your code and not restarting the Shell thereafter. This is especially useful after adding imports at the top of a file. This also increases possible attribute completions.

Completion boxes initially exclude names beginning with `“_”` or, for modules, not included in `“__all__”`. The hidden names can be accessed by typing `“_”` after `“.”`, either before or after the box is opened.

Sugerencias de llamada

A calltip is shown automatically when one types (after the name of an *accessible* function. A function 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. Whenever the cursor is in the argument part of a definition, select Edit and «Show Call Tip» on the menu or enter its shortcut to display a calltip.

The calltip consists of the function’s signature and docstring up to the latter’s first blank line or the fifth non-blank line. (Some builtin functions lack an accessible signature.) A `/` or `*` in the signature indicates that the preceding or following arguments are passed by position or name (keyword) only. Details are subject to change.

In Shell, the accessible functions depends on what modules have been imported into the user process, including those imported by Idle itself, and which 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 itself import `turtle`. The menu entry and shortcut also do nothing. Enter `import turtle`. Thereafter, `turtle.write (` will display a calltip.

In an editor, import statements have no effect until one runs the file. One might want to run a file after writing import statements, after adding function definitions, or after opening an existing file.

Contexto del código

Dentro de una ventana del editor que contiene código Python, el contexto del código se puede alternar para mostrar u ocultar un panel en la parte superior de la ventana. Cuando se muestra, este panel congela las líneas de apertura por bloques de código, como aquellos que comienzan con las palabras clave `class`, `def`, o `if`, que de otro modo se habrían desplazado fuera de la vista. El tamaño del panel se expandirá y contraerá según sea necesario para mostrar todos los niveles actuales de contexto, hasta el número máximo de líneas definidas en el cuadro de diálogo Configurar IDLE (que por defecto es 15). Si no hay líneas de contexto actuales y la función está activada, se visualizará una sola línea en blanco. Al hacer click en una línea en el panel de contexto, esa línea se moverá a la parte superior del editor.

El texto y los colores de fondo para el panel de contexto se pueden configurar en la pestaña destacados en el cuadro de diálogo Configurar IDLE.

Shell de Python

Con el shell del IDLE, se ingresa, edita y hace recae declaraciones completas. La mayoría de consolas y terminales solo funcionan con una sola línea física a la vez.

Cuando se pega código en el shell, este no es compilado y posiblemente ejecutado hasta que se teclea `Return`. Se puede editar el código pegado primero. Si se pega más de una declaración en el shell, el resultado será un `SyntaxError` cuando se compilan varias declaraciones como si fueran una sola.

Las funciones de edición descritas en subsecciones anteriores funcionan cuando se ingresa código de forma interactiva. La ventana de shell del IDLE también responde a las siguientes combinaciones de teclas.

- `C-c` interrumpe la ejecución del comando
- `C-d` lo envía el final del archivo; la ventana se cierra si se escribe en un mensaje `>>>`
- `Alt-/` (Expandir palabra) también es útil para reducir la escritura

Historial de comandos

- `Alt-p` recupera el comando anterior que coincide con lo que ha escrito. En macOS use `C-p`.
- `Alt-n` recupera el siguiente. En macOS use `C-n`.
- `Return` ingresando en un comando anterior, recupera ese comando

Colores del texto

El idle por defecto es negro sobre texto blanco, pero colorea el texto con significados especiales. Para el shell, estos son: la salida del shell, error del shell, salida del usuario y error del usuario. Para el código Python, en el indicador de comandos de shell o en un editor, estos son: palabras clave, nombres de funciones y clases incorporadas, los siguientes nombres `class` and `def`, cadenas de caracteres y comentarios. Para cualquier ventana de texto, estos son: el cursor (cuando está presente), el texto encontrado (cuando sea posible) y el texto seleccionado.

La coloración del texto se realiza en segundo plano, por lo que ocasionalmente se puede ver el texto sin colorear. Para cambiar la combinación de colores, use la pestaña Resaltar en el cuadro de diálogo Configurar IDLE. El marcado de líneas de breakpoint del depurador en el editor y el texto en ventanas emergentes y cuadros de diálogo no es configurable por el usuario.

25.5.3 Inicio y ejecución de código

Al iniciar con la opción `-s`, el IDLE ejecutará el archivo al que hacen referencia las variables de entorno `IDLESTARTUP` o `PYTHONSTARTUP`. El IDLE primero verifica `IDLESTARTUP`; si `IDLESTARTUP` está presente, se ejecuta el archivo al que se hace referencia. Si `IDLESTARTUP` no está presente, IDLE verifica `PYTHONSTARTUP`. Los archivos referenciados por estas variables de entorno son lugares convenientes para almacenar funciones que son usadas con frecuencia desde el shell del IDLE o para ejecutar declaraciones importadas para importar módulos comunes.

Además, Tk también carga un archivo de inicio si este está presente. Tenga en cuenta que el archivo Tk se carga incondicionalmente. Este archivo adicional es `.Idle.py` y es buscado en el directorio de inicio del usuario. Las declaraciones en este archivo se ejecutarán en el espacio de nombres Tk, por lo que este archivo no es útil para importar funciones que se utilizarán desde el shell de Python del IDLE.

Uso de línea de comando

```
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)
```

Si están los argumentos:

- Si se usa `-`, `-c` o `r`, todos los argumentos se colocan en `sys.argv [1:...] `` y ``sys.argv[0]` se establece en `' '`, `'-c'` o `'-r'`. No se abre ninguna ventana del editor, incluso si ese es el conjunto predeterminado en el cuadro de diálogo opciones.
- De lo contrario, los argumentos son archivos abiertos para edición y `sys.argv` refleja los argumentos pasados al IDLE.

Error de inicio

IDLE uses a socket to communicate between the IDLE GUI process and the user code execution process. A connection must be established whenever the Shell starts or restarts. (The latter is indicated by a divider line that says “RESTART”). If the user process fails to connect to the GUI process, it usually displays a Tk error box with a “cannot connect” message that directs the user here. It then exits.

One specific connection failure on Unix systems results from misconfigured masquerading rules somewhere in a system’s network setup. When IDLE is started from a terminal, one will see a message starting with `** Invalid host:.` The valid value is `127.0.0.1 (idlelib.rpc.LOCALHOST)`. One can diagnose with `tcpconnect -irv 127.0.0.1 6543` in one terminal window and `tcplisten <same args>` in another.

Una causa común de falla se da cuando un archivo escrito por el usuario tiene el mismo nombre de un módulo de biblioteca estándar, como `random.py` y `tkinter.py`. Cuando dicho archivo se encuentra en el mismo directorio que un archivo que está por ejecutarse, el IDLE no puede importar el archivo `stdlib`. La solución actual es cambiar el nombre del archivo del usuario.

Aunque es menos común que en el pasado, un programa de antivirus o firewall puede detener la conexión. Si el programa no se puede configurar para permitir la conexión, debe estar apagado para que funcione el IDLE. Es seguro permitir esta conexión interna porque no hay datos visibles en puertos externos. Un problema similar es una configuración incorrecta de la red que bloquea las conexiones.

Los problemas de instalación de Python ocasionalmente detienen el IDLE: puede darse un conflicto entre versiones o una sola instalación puede requerir privilegios de administrador. Si se soluciona el conflicto o no se puede o quiere ejecutar como administrador, puede ser más fácil desinstalar completamente Python y comenzar de nuevo.

Un proceso zombie pythonw.exe podría ser un problema. En Windows, use el Administrador de tareas para buscar uno y detenerlo si lo hay. A veces, un reinicio iniciado por un bloqueo del programa o una interrupción del teclado (control-C) puede fallar al conectarse. Descartar el cuadro de error o usar Reiniciar Shell en el menú del Shell puede solucionar un problema temporal.

Cuando el IDLE se inicia por primera vez, intenta leer los archivos de configuración de usuario en `~/.idlerc/` (~ este es el directorio principal). Si hay un problema, se mostrará un mensaje de error. Dejando de lado los fallos aleatorios del disco, esto se puede evitar si nunca se editan los archivos a mano. En su lugar, utilice el cuadro de diálogo de configuración, en Opciones. Una vez que hay un error en un archivo de configuración de usuario, la mejor solución puede ser eliminarlo y empezar de nuevo con el cuadro de diálogo de configuración.

Si el IDLE se cierra sin mensaje y este no fue iniciado desde una consola, intente iniciarlo desde una consola o terminal (`python -m idlelib`) y observe si esto genera un mensaje de error.

On Unix-based systems with tcl/tk older than 8.6.11 (see About IDLE) certain characters of certain fonts can cause a tk failure with a message to the terminal. This can happen either if one starts IDLE to edit a file with such a character or later when entering such a character. If one cannot upgrade tcl/tk, then re-configure IDLE to use a font that works better.

Ejecutando código del usuario

With rare exceptions, the result of executing Python code with IDLE is intended to be the same as executing the same code by the default method, directly with Python in a text-mode system console or terminal window. However, the different interface and operation occasionally affect visible results. For instance, `sys.modules` starts with more entries, and `threading.active_count()` returns 2 instead of 1.

De forma predeterminada, el IDLE ejecuta el código de usuario en un proceso separado del sistema operativo en lugar de hacerlo en el proceso de la interfaz de usuario que ejecuta el shell y el editor. En el proceso de ejecución, este reemplaza `sys.stdin`, `sys.stdout`, y `sys.stderr` con objetos que recuperan las entradas desde y envían las salidas hacia la ventana de la consola. Los valores originales almacenados en `sys.__stdin__`, `sys.__stdout__`, y `sys.__stderr__` no se ven afectados, pero pueden ser `None`.

Sending print output from one process to a text widget in another is slower than printing to a system terminal in the same process. This has the most effect when printing multiple arguments, as the string for each argument, each separator, the newline are sent separately. For development, this is usually not a problem, but if one wants to print faster in IDLE, `format` and `join` together everything one wants displayed together and then print a single string. Both `format` strings and `str.join()` can help combine fields and lines.

Las sustituciones de flujo estándar del IDLE no son heredadas por subprocessos creados en el proceso de ejecución, siendo directamente por código de usuario o por módulos como el multiprocessing. Si tal subprocesso usa `input` desde `sys.stdin` o `print` o `write` hacia `sys.stdout` o `sys.stderr`, el IDLE debe ser iniciado en una ventana de línea de comando. El subprocesso secundario será adjuntado a esa ventana para entrada y salida.

Si `sys` es restablecido mediante un código de usuario, tal como `importlib.reload(sys)`, los cambios del IDLE se perderán y la entrada del teclado y la salida de la pantalla no funcionarán correctamente.

Cuando el shell está en primer plano, controla el teclado y la pantalla. Este es transparente normalmente, pero las funciones que acceden directamente al teclado y la pantalla no funcionarán. Estas incluyen funciones específicas del sistema que determinan si se ha presionado una tecla y de ser así, cuál.

La ejecución de código del IDLE en el proceso de ejecución agrega marcos a la pila de llamadas que de otro modo no estarían allí. el IDLE encapsula `sys.getrecursionlimit` y `sys.setrecursionlimit` para reducir el efecto de los marcos de pila adicionales.

Cuando el código de usuario genera `SystemExit` directamente o llamando a `sys.exit`, el IDLE regresa al visualizador del Shell en lugar de salir.

Salida del usuario en consola

Cuando un programa muestra texto, el resultado está determinado por el dispositivo de salida correspondiente. Cuando el IDLE ejecuta el código de usuario `sys.stdout` y `sys.stderr` se conectan al área de visualización del Shell del IDLE. Algunas de estas características son heredadas de los widgets de texto Tk subyacentes. Otras son adiciones programadas. Donde es importante, el shell está diseñado para el desarrollo en lugar de la ejecución de producción.

Por ejemplo, el Shell nunca elimina la salida. Un programa que envía salidas ilimitadas al Shell eventualmente llenará la memoria, lo que resultará en un error de memoria. Por el contrario, algunas ventanas de texto del sistema solo conservan las últimas *n* líneas de salida. Una consola de Windows, por ejemplo, mantiene una línea configurable por el usuario de 1 a 9999, con 300 por defecto.

Un widget de texto de Tk, y por lo tanto el Shell de IDLE, muestra caracteres (puntos de código) en el subconjunto BMP (plano multilingual básico) de Unicode. Aquellos caracteres que se muestran con un glifo adecuado y los cuales con una caja de reemplazo depende del sistema operativo y las fuentes instaladas. Los caracteres de tabulación hacen que el texto siguiente comience después de la siguiente tabulación. (Ocurre cada 8 “caracteres”). Los caracteres de nueva línea hacen que el texto siguiente aparezca en una nueva línea. Otros caracteres de control se omiten o se muestran como un espacio, cuadro u otra cosa, dependiendo del sistema operativo y la fuente. (Mover el cursor del texto a través de esa salida con las teclas de flecha puede mostrar algún comportamiento de espaciado sorpresivo.)

```
>>> s = 'a\tb\ac<\x02><\r>\bc\nd' # Enter 22 chars.
>>> len(s)
14
>>> s # Display repr(s)
'a\tb\x07<\x02><\r>\x08c\nd'
>>> print(s, end='') # Display s as is.
# Result varies by OS and font. Try it.
```

La función `repr` es usada para la visualización interactiva del valor de las expresiones. Esta retorna una versión modificada de la cadena de caracteres de entrada en la que los códigos de control, algunos puntos de código BMP y todos los puntos de código que no son BMP son reemplazados con caracteres de escape. Como se demostró anteriormente, esto permite identificar los caracteres en una cadena de caracteres, independientemente de cómo sean mostrados.

La salida normal y de error generalmente se mantienen separadas (en líneas separadas) de la entrada de código y entre sí. Cada una obtiene diferentes colores de resaltado.

Para el *traceback* de `SyntaxError`, el marcado normal “^” dónde se detectó el error se reemplaza coloreando el texto con un resaltado de error. Cuando el código ejecutado desde un archivo causa otras excepciones, se puede hacer click derecho en un *traceback* para saltar a la línea correspondiente en un editor del IDLE. El archivo se abrirá si es necesario.

El Shell tiene una funcionalidad especial para exprimir las líneas de salida hasta una etiqueta de “Texto *Squeezed*”. Esto se hace automáticamente para una salida sobre *N* líneas (*N* = 50 por defecto). *N* se puede cambiar en la sección PyShell de la página General del cuadro de diálogo Configuración. La salida con menos líneas puede ser *squeezed* haciendo click derecho en la salida. Esto puede ser útil en líneas suficientemente largas para bajar el tiempo de desplazamiento.

La salida *squeezed* se expande en su lugar haciendo doble click en la etiqueta. También se puede enviar al portapapeles o a una ventana de vista separada haciendo click derecho en la etiqueta.

Desarrollando aplicaciones tkinter

El IDLE es intencionalmente diferente de Python estándar para facilitar el desarrollo de programas tkinter. Ingrese `import tkinter as tk; root = tk.Tk()` en Python estándar y no aparecerá nada. Ingrese lo mismo en el IDLE y aparecerá una ventana tk. En Python estándar, también se debe ingresar `root.update()` para ver la ventana. El IDLE hace el equivalente en segundo plano, aproximadamente 20 veces por segundo, lo que es aproximadamente cada 50 milisegundos. Luego ingrese `b = tk.Button(root, text='button');` `b.pack()`. Del mismo modo, no hay cambios visibles en Python estándar hasta que ingrese `root.update()`.

La mayoría de los programas tkinter ejecutan `root.mainloop()`, que generalmente no retorna hasta que se destruye la aplicación tk. Si el programa se ejecuta con `python -i` o desde un editor del IDLE, no aparecerá un mensaje `>>>` del shell hasta que retorne `mainloop()`, momento en el cual no queda nada con lo que interactuar.

Al ejecutar un programa tkinter desde un editor IDLE, se puede comentar la llamada de bucle principal. Luego se recibe un aviso del shell inmediatamente y se puede interactuar con la aplicación en vivo. Solo se debe recordar reactivar la llamada al bucle principal cuando se ejecuta en Python estándar.

Ejecutando sin un subprocesso

Por defecto, el IDLE ejecuta el código de usuario en un subprocesso separado a través de un socket, el cual utiliza la interfaz de bucle interno. Esta conexión no es visible externamente y no son enviados ni recibidos datos de Internet. Si el software de cortafuego sigue presentando problemas, puede ignorarlo.

Si el intento de conexión del socket falla, IDLE lo notificará. Este tipo de falla a veces es temporal, pero si persiste, el problema puede provenir de un cortafuego que bloquea la conexión o de una mala configuración en un sistema en particular. Hasta que se solucione el problema, se puede ejecutar el Idle con el modificador de línea de comandos `-n`.

Si el IDLE se inicia con el modificador de línea de comandos `-n`, se ejecutará en un único proceso y no creará el subprocesso que ejecuta el servidor de ejecución de Python RPC. Esto puede ser útil si Python no puede crear el subprocesso o la interfaz de socket RPC en su plataforma. Sin embargo, en este modo el código de usuario no está aislado en sí del IDLE. Además, el entorno no se reinicia cuando se selecciona Ejecutar/Ejecutar módulo (F5). Si el código se ha modificado, se debe volver a cargar() los módulos afectados y volver a importar cualquier elemento específico (por ejemplo, desde `foo` importar `baz`) para que los cambios surtan efecto. Por estas razones, es preferible ejecutar el IDLE con el subprocesso predeterminado si es posible.

Obsoleto desde la versión 3.4.

25.5.4 Ayuda y preferencias

Recursos de ayuda

La entrada del menú Ayuda «Ayuda IDLE» muestra una versión HTML formateada del capítulo IDLE de la Referencia de la biblioteca. El resultado, en una ventana de texto tkinter de solo lectura, está cerca de lo que se ve en un navegador web. Navegue por el texto con la rueda del ratón, la barra de desplazamiento o presionando las teclas de flecha arriba y abajo. O haga click en el botón TDC (Tabla de contenido) y seleccione un encabezado de sección en el cuadro abierto.

La entrada del menú de ayuda «Documentos de Python» abre amplias fuentes de ayuda, incluyendo tutoriales, disponibles en `docs.python.org/x.y`, donde “x.y” es la versión de Python actualmente en ejecución. Si su sistema tiene una copia fuera de línea de los documentos (esta puede ser una opción de instalación), esta se abrirá en su lugar.

Las URL seleccionadas se pueden agregar o eliminar del menú de ayuda en cualquier momento utilizando la pestaña General del cuadro de diálogo Configurar IDLE.

Preferencias de configuración

Las preferencias de fuente, resaltado, teclas y preferencias generales se pueden cambiar en Configurar IDLE en el menú opción. Las configuraciones de usuario no predeterminadas se guardan en un directorio `.idlerc` en el directorio de inicio del usuario. Los problemas causados por archivos de configuración de usuario incorrectos se resuelven editando o eliminando uno o más de los archivos en `.idlerc`.

En la pestaña Fuentes (*Font*), consulte la muestra de texto para ver el efecto de la fuente y el tamaño de la fuente en varios caracteres en varios idiomas. Edite la muestra para agregar otros caracteres de interés personal. Use la muestra para seleccionar fuentes monoespaciadas. Si determinados caracteres tienen problemas en el shell o en el editor, agréguelos a la parte inicial de la muestra e intente cambiar primero el tamaño y luego la fuente.

En la pestaña Resaltado y Teclas (*Highlights and Keys*), seleccione un tema de color integrado o personalizado y un conjunto de teclas. Para utilizar un nuevo tema de color integrado o un conjunto de teclas con IDLE más antiguos, guárdelo como un nuevo tema personalizado o conjunto de teclas y será accesible para los IDLE más antiguos.

IDLE en macOS

En Preferencias del sistema: Dock, puede establecer «Preferencias de pestañas al abrir documentos» en el valor «Siempre». Este parámetro no es compatible con la interfaz gráfica de usuario del framework tk/tkinter utilizado por IDLE y rompe algunas funcionalidades del IDLE.

Extensiones

IDLE incluye una herramienta de extensiones. Las preferencias para las extensiones se pueden cambiar con la pestaña Extensiones de la ventana de preferencias. Lea el inicio de `config-extensions.def` en la carpeta `idlelib` para obtener más información. La única extensión utilizada actualmente por defecto es `zdummy`, un ejemplo que también se utiliza para realizar pruebas.

25.6 Otros paquetes de interfaz gráfica de usuario

Los principales kits de herramientas GUI multiplataforma (Windows, Mac OS X, similar a Unix) están disponibles para Python:

Ver también:

“PyGObject <<https://wiki.gnome.org/Projects/PyGObject>>”_ PyGObject proporciona enlaces de introspección para bibliotecas de C mediante GObject. Una de estas bibliotecas es el conjunto de widgets GTK+ 3. GTK+ viene con muchos más widgets de los que proporciona Tkinter. Un [Python GTK+ 3 Tutorial](#) está disponible.

PyGTK PyGTK proporciona enlaces para una versión anterior de la biblioteca, GTK+ 2. Proporciona una interfaz orientada a objetos que es ligeramente más alto que el nivel C. También hay enlaces a [GNOME](#). Un [tutorial](#) en línea está disponible.

PyQt PyQt es un enlace **sip**-wrapped al kit de herramientas Qt. Qt es un extenso marco de desarrollo de aplicaciones de C++ GUI que está disponible para Unix, Windows y Mac OS X. **sip** es una herramienta para generar enlaces para bibliotecas de C++ como clases de Python, y está diseñado específicamente para Python.

PySide2 Also known as the Qt for Python project, PySide2 is a newer binding to the Qt toolkit. It is provided by The Qt Company and aims to provide a complete port of PySide to Qt 5. Compared to PyQt, its licensing scheme is friendlier to non-open source applications.

wxPython wxPython es un kit de herramientas GUI multiplataforma para Python que se basa en el popular kit de herramientas de C++ **wxWidgets** (anteriormente **wxWindows**). Proporciona un aspecto nativo para aplicaciones en sistemas Windows, Mac OS X y Unix mediante el uso de widgets nativos de cada plataforma siempre que sea posible, (GTK + en sistemas similares a Unix). Además de un amplio conjunto de widgets, wxPython proporciona clases de documentación en línea y ayuda contextual, impresión, visualización HTML, dibujo

de contexto de dispositivo de bajo nivel, arrastrar y soltar, acceso al portapapeles del sistema, un formato de recursos basado en XML y más, incluida una biblioteca cada vez mayor de módulos aportados por el usuario.

PyGTK, PyQt, PySide2, 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.

Herramientas de desarrollo

Los módulos descritos en este capítulo le ayudan a escribir software. Por ejemplo, el módulo *pydoc* toma un módulo y genera documentación basada en el contenido del módulo. Los módulos *doctest* y *unittest* contienen frameworks para escribir pruebas unitarias que ejecutan y validan automáticamente el código, verificando que se produce la salida esperada. **2to3** puede traducir el código fuente de Python 2.x en código válido de Python 3.x.

La lista de módulos descritos en este capítulo es:

26.1 typing — Soporte para *type hints*

Nuevo en la versión 3.5.

Source code: [Lib/typing.py](#)

Nota: En tiempo de ejecución, Python no impone las anotaciones de tipado en funciones y variables. Pueden ser utilizadas por herramientas de terceros como validadores de tipado, IDEs, linters, etc.

Este módulo proporciona soporte en tiempo de ejecución (*runtime*) para anotaciones de tipado, tal y como se especifica en **PEP 484**, **PEP 526**, **PEP 544**, **PEP 586**, **PEP 589**, y **PEP 591**. Las características fundamentales consisten en los tipos *Any*, *Union*, *Tuple*, *Callable*, *TypeVar*, and *Generic*. Para la especificación completa véase **PEP 484**. Para una introducción simplificada a *type hints* véase **PEP 483**.

La siguiente función toma y retorna una cadena de texto, que se anota de la siguiente manera:

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

En la función `greeting`, se espera que el argumento `name` sea de tipo `str` y que el tipo retornado sea `str`. Los subtipos también son aceptados como argumento válido.

26.1.1 Alias de tipo

Un alias de tipo se define asignando el tipo al alias. En este ejemplo, `Vector` y `List[float]` serán tratados como sinónimos intercambiables:

```
from typing import List
Vector = List[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# typechecks; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

Los alias de tipo son útiles para simplificar indicadores de tipo complejos. Por ejemplo:

```
from typing import Dict, Tuple, Sequence

ConnectionOptions = Dict[str, str]
Address = Tuple[str, int]
Server = Tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
    message: str,
    servers: Sequence[Tuple[Tuple[str, int], Dict[str, str]]] -> None:
    ...
```

Nótese que `None` como indicador de tipo es un caso especial y es substituido por `type(None)`.

26.1.2 NewType

Úsease la función auxiliar `NewType()` para crear tipos distinguibles entre sí:

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

El validador estático de tipos tratará el nuevo tipo como si fuera una subclase del tipo original. Esto es útil para capturar errores lógicos:

```
def get_user_name(user_id: UserId) -> str:
    ...

# typechecks
user_a = get_user_name(UserId(42351))

# does not typecheck; an int is not a UserId
user_b = get_user_name(-1)
```

Se pueden realizar todas las operaciones de `int` en una variable de tipo `UserId`, pero el resultado siempre será de tipo `int`. Esto permite pasar un `UserId` allí donde se espere un `int`, pero evitará la creación accidental de un `UserId` de manera incorrecta:

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

Nótese que estas comprobaciones son impuestas solo en la validación de tipado estática. En tiempo de ejecución, la sentencia `Derived = NewType('Derived', Base)` hará de `Derived` una función que retornará inmediatamente el parámetro que se le pase. Esto implica que la expresión `Derived(some_value)` no crea una nueva clase o genera más coste que la llamada a una función normal.

Más concretamente, la expresión `some_value is Derived(some_value)` será siempre verdadera en tiempo de ejecución.

Esto también implica que no es posible crear un subtipo de `Derived` ya que, en tiempo de ejecución, es una función de identidad, no un tipo propiamente dicho:

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not typecheck
class AdminUserId(UserId): pass
```

Sin embargo, es posible crear un `NewType()` basado en un `NewType` “derivado”:

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

y la comprobación de tipo para `ProUserId` funcionará como se espera.

Véase [PEP 484](#) para más detalle.

Nota: Recuérdese que el uso de alias de tipo implica que los dos tipos son *equivalentes* entre sí. Haciendo `Alias = Original` provocará que el Validador estático de tipos trate `Alias` como algo *exactamente equivalente* a `Original` en todos los casos. Esto es útil para cuando se quiera simplificar indicadores de tipo complejos.

En cambio, `NewType` declara un tipo que es *subtipo* de otro. Haciendo `Derived = NewType('Derived', Original)` hará que el Validador estático de tipos trate `Derived` como una *subclase* de `Original`, lo que implica que un valor de tipo `Original` no puede ser usado allí donde se espere un valor de tipo `Derived`. Esto es útil para prevenir errores lógicos con un coste de ejecución mínimo.

Nuevo en la versión 3.5.2.

26.1.3 Callable

Entidades que esperen llamadas a funciones con interfaces específicas puede ser anotadas usando `Callable[[Arg1Type, Arg2Type], ReturnType]`.

Por ejemplo:

```
from typing import Callable

def feeder(get_next_item: Callable[[], str]) -> None:
    # Body

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    # Body
```

Es posible declarar el tipo de retorno de un *callable* (invocable) sin especificar tipos en los parámetros substituyendo la lista de argumentos por unos puntos suspensivos (...) en el indicador de tipo: `Callable[..., ReturnType]`.

26.1.4 Genéricos

Ya que no es posible inferir estáticamente y de una manera genérica la información de tipo de objetos dentro de contenedores, las clases base abstractas han sido mejoradas para permitir sintaxis de subíndice para denotar los tipos esperados en elementos contenedores.

```
from typing import Mapping, Sequence

def notify_by_email(employees: Sequence[Employee],
                   overrides: Mapping[str, str]) -> None: ...
```

Los genéricos se pueden parametrizar usando una nueva factoría disponible en *typing* llamada *TypeVar*.

```
from typing import Sequence, TypeVar

T = TypeVar('T')           # Declare type variable

def first(l: Sequence[T]) -> T:    # Generic function
    return l[0]
```

26.1.5 Tipos genéricos definidos por el usuario

Una clase definida por el usuario puede ser definida como una clase genérica.

```
from typing import TypeVar, Generic
from logging import Logger

T = TypeVar('T')

class LoggedVar(Generic[T]):
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

`Generic[T]` como clase base define que la clase `LoggedVar` toma un solo parámetro `T`. Esto también implica que `T` es un tipo válido dentro del cuerpo de la clase.

La clase base *Generic* define `__class_getitem__()` para que `LoggedVar[t]` sea válido como tipo:

```
from typing import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

Un tipo genérico puede tener un número indefinido de variables de tipo, y pueden limitarse a tipos concretos:

```

from typing import TypeVar, Generic
...

T = TypeVar('T')
S = TypeVar('S', int, str)

class StrangePair(Generic[T, S]):
    ...

```

Cada argumento de variable de tipo en una clase *Generic* debe ser distinto. Así, no será válido:

```

from typing import TypeVar, Generic
...

T = TypeVar('T')

class Pair(Generic[T, T]):    # INVALID
    ...

```

Se puede utilizar herencia múltiple con *Generic*:

```

from typing import TypeVar, Generic, Sized

T = TypeVar('T')

class LinkedList(Sized, Generic[T]):
    ...

```

Cuando se hereda de clases genéricas, se pueden fijar algunas variables de tipo:

```

from typing import TypeVar, Mapping

T = TypeVar('T')

class MyDict(Mapping[str, T]):
    ...

```

En este caso `MyDict` tiene un solo parámetro, `T`.

Al usar una clase genérica sin especificar parámetros de tipo se asume *Any* para todas las posiciones. En el siguiente ejemplo, `MyIterable` no es genérico pero hereda implícitamente de `Iterable[Any]`:

```

from typing import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]

```

Son posibles los alias de tipos genéricos definidos por el usuario. Ejemplos:

```

from typing import TypeVar, Iterable, Tuple, Union
S = TypeVar('S')
Response = Union[Iterable[S], int]

# Return type here is same as Union[Iterable[str], int]
def response(query: str) -> Response[str]:
    ...

T = TypeVar('T', int, float, complex)
Vec = Iterable[Tuple[T, T]]

def inproduct(v: Vec[T]) -> T: # Same as Iterable[Tuple[T, T]]
    return sum(x*y for x, y in v)

```

Distinto en la versión 3.7: *Generic* ya no posee una metaclasses personalizable.

Un clase genérica definida por el usuario puede tener clases ABC como clase base sin conflicto de metaclasses. Las metaclasses genéricas no están permitidas. El resultado de parametrizar clases genéricas se cachea, y la mayoría de los tipos en el módulo *typing* pueden tener un hash y ser comparables por igualdad (*equality*).

26.1.6 El tipo Any

Un caso especial de tipo es *Any*. Un Validador estático de tipos tratará cualquier tipo como compatible con *Any*, y *Any* como compatible con todos los tipos.

This means that it is possible to perform any operation or method call on a value of type *Any* and assign it to any variable:

```
from typing import Any

a = None      # type: Any
a = []        # OK
a = 2         # OK

s = ''        # type: str
s = a         # OK

def foo(item: Any) -> int:
    # Typechecks; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

Nótese que no se realiza comprobación de tipo cuando se asigna un valor de tipo *Any* a un tipo más preciso. Por ejemplo, el Validador estático de tipos no reportó ningún error cuando se asignó *a* a *s*, aún cuando se declaró *s* como de tipo *str* y recibió un valor *int* en tiempo de ejecución.

Además, todas las funciones sin un tipo de retorno o tipos en los parámetros serán asignadas implícitamente a *Any* por defecto:

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

Este comportamiento permite que *Any* sea usado como una *vía de escape* cuando es necesario mezclar código tipado estática y dinámicamente.

Compárese el comportamiento de *Any* con el de *object*. De manera similar a *Any*, todo tipo es un subtipo de *object*. Sin embargo, en oposición a *Any*, lo contrario no es cierto: *object* no es un subtipo de ningún otro tipo.

Esto implica que cuando el tipo de un valor es *object*, un validador de tipos rechazará prácticamente todas las operaciones con él, y al asignarlo a una variable (o usarlo como valor de retorno) de un tipo más preciso será un error de tipo. Por ejemplo:

```
def hash_a(item: object) -> int:
    # Fails; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Typechecks
    item.magic()
```

(continué en la próxima página)

(proviene de la página anterior)

```
...

# Typechecks, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")

# Typechecks, since Any is compatible with all types
hash_b(42)
hash_b("foo")
```

Úsese *object* para indicar que un valor puede ser de cualquier tipo de manera segura. Úsese *Any* para indicar que un valor es de tipado dinámico.

26.1.7 Subtipado nominal vs estructural

Inicialmente, el [PEP 484](#) definió el sistema de tipado estático de Python como *nominal*. Esto implica que una clase A será permitida allí donde se espere una clase B si y solo si A es una subclase de B.

This requirement previously also applied to abstract base classes, such as *Iterable*. The problem with this approach is that a class had to be explicitly marked to support them, which is unpythonic and unlike what one would normally do in idiomatic dynamically typed Python code. For example, this conforms to [PEP 484](#):

```
from typing import Sized, Iterable, Iterator

class Bucket(Sized, Iterable[int]):
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...
```

El [PEP 544](#) permite resolver este problema al permitir escribir el código anterior sin una clase base explícita en la definición de la clase, permitiendo que el Validador estático de tipo considere implícitamente que *Bucket* es un subtipo tanto de *Sized* como de *Iterable[int]*. Esto se conoce como tipado *estructural* (o *duck-typing* estático):

```
from typing import Iterator, Iterable

class Bucket: # Note: no base classes
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...

def collect(items: Iterable[int]) -> int: ...
result = collect(Bucket()) # Passes type check
```

Asimismo, creando subclases de la clase especial *Protocol*, el usuario puede definir nuevos protocolos personalizados y beneficiarse del tipado estructural (véanse los ejemplos de abajo).

26.1.8 Clases, funciones y decoradores

El módulo define las siguientes clases, funciones y decoradores:

class `typing.TypeVar`
Variable de tipo.

Uso:

```
T = TypeVar('T') # Can be anything
A = TypeVar('A', str, bytes) # Must be str or bytes
```

Las variables de tipo son principalmente para ayudar a los validadores estáticos de tipos. Sirven tanto como de parámetros para tipos genéricos como para definición de funciones genéricas. Véase la clase *Generic* para más información sobre tipos genéricos. Las funciones genéricas funcionan como sigue:

```
def repeat(x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n

def longest(x: A, y: A) -> A:
    """Return the longest of two strings."""
    return x if len(x) >= len(y) else y
```

La signatura de los ejemplos anteriores es esencialmente la superposición de `(str, str) -> str` y `(bytes, bytes) -> bytes`. Nótese también que aunque los argumentos sean instancias de alguna subclase de *str*, el tipo retornado aún será una simple *str*.

En tiempo de ejecución, `isinstance(x, T)` lanzará una excepción *TypeError*. En general, `isinstance()` y `issubclass()` no se deben usar con variables de tipo.

Las variables de tipo pueden ser marcadas como covariantes o contravariantes pasando `covariant=True` o `contravariant=True`, respectivamente. Véase **PEP 484** para más detalles. Por defecto, las variables de tipo son invariantes. Opcionalmente, una variable de tipo puede especificar un límite superior usando `bound=<type>`. Esto significa que el tipo (explícitamente o implícitamente) tiene que ser una subclase del tipo límite, véase **PEP 484**.

class `typing.Generic`

Clase base abstracta para tipos genéricos.

Un tipo genérico se declara habitualmente heredando de una instancia de esta clase con una o más variables de tipo. Por ejemplo, un tipo de mapeo genérico se podría definir como:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

Entonces, esta clase se puede usar como sigue:

```
X = TypeVar('X')
Y = TypeVar('Y')

def lookup_name(mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

class `typing.Protocol` (*Generic*)

Clase base para clases protocolo. Las clases protocolo se definen así:

```
class Proto(Protocol):
    def meth(self) -> int:
        ...
```

Tales clases son usadas principalmente con validadores estáticos de tipos que detectan subtipado estructural (*duck-typing* estático), por ejemplo:

```
class C:
    def meth(self) -> int:
        return 0

def func(x: Proto) -> int:
    return x.meth()
```

(continué en la próxima página)

(proviene de la página anterior)

```
func(C()) # Passes static type check
```

Véase [PEP 544](#) para más detalles. Las clases protocolo decoradas con `runtime_checkable()` (que se explica más adelante) se comportan como protocolos simplistas en tiempo de ejecución que solo comprueban la presencia de atributos dados, ignorando su firma de tipo.

Las clases protocolo pueden ser genéricas, por ejemplo:

```
class GenProto(Protocol[T]):
    def meth(self) -> T:
        ...
```

Nuevo en la versión 3.8.

class `typing.Type` (`Generic[CT_co]`)

Una variable indicada como `C` puede aceptar valores de tipo `C`. Sin embargo, un variable indicada como `Type[C]` puede aceptar valores que son clases en sí mismas – específicamente, aceptará el *objecto clase* de `C`. Por ejemplo.:

```
a = 3          # Has type 'int'
b = int        # Has type 'Type[int]'
c = type(a)    # Also has type 'Type[int]'
```

Nótese que `Type[C]` es covariante:

```
class User: ...
class BasicUser(User): ...
class ProUser(User): ...
class TeamUser(User): ...

# Accepts User, BasicUser, ProUser, TeamUser, ...
def make_new_user(user_class: Type[User]) -> User:
    # ...
    return user_class()
```

El hecho de que `Type[C]` sea covariante implica que todas las subclases de `C` deben implementar la misma interfaz del constructor y las mismas interfaces de los métodos de clase que `C`. El validador de tipos marcará cualquier incumplimiento de esto, pero permitirá llamadas al constructor que coincida con la llamada al constructor de la clase base indicada. El modo en que el validador de tipos debe gestionar este caso particular podría cambiar en futuras revisiones de [PEP 484](#).

Lo únicos parámetros válidos de `Type` son clases, *Any*, *type variables*, y uniones de cualquiera de los tipos anteriores. Por ejemplo:

```
def new_non_team_user(user_class: Type[Union[BasicUser, ProUser]]): ...
```

`Type[Any]` es equivalente a `Type`, que a su vez es equivalente a `type`, que es la raíz de la jerarquía de metaclasses de Python.

Nuevo en la versión 3.5.2.

class `typing.Iterable` (`Generic[T_co]`)

Una versión genérica de `collections.abc.Iterable`.

class `typing.Iterator` (`Iterable[T_co]`)

Una versión genérica de `collections.abc.Iterator`.

class `typing.Reversible` (`Iterable[T_co]`)

Una versión genérica de `collections.abc.Reversible`.

class `typing.SupportsInt`

Una ABC con un método abstracto `__int__`.

class `typing.SupportsFloat`

Una ABC con un método abstracto `__float__`.

class `typing.SupportsComplex`

Una ABC con un método abstracto `__complex__`.

class `typing.SupportsBytes`

Una ABC con un método abstracto `__bytes__`.

class `typing.SupportsIndex`

Una ABC con un método abstracto `__index__`.

Nuevo en la versión 3.8.

class `typing.SupportsAbs`

Una ABC con un método abstracto `__abs__` que es covariante en su tipo retornado.

class `typing.SupportsRound`

Una ABC con un método abstracto `__round__` que es covariantes en su tipo retornado.

class `typing.Container` (*Generic*[*T_co*])

Una versión genérica de `collections.abc.Container`.

class `typing.Hashable`

Un alias de `collections.abc.Hashable`

class `typing.Sized`

Un alias de `collections.abc.Sized`

class `typing.Collection` (*Sized*, *Iterable*[*T_co*], *Container*[*T_co*])

Una versión genérica de `collections.abc.Collection`

Nuevo en la versión 3.6.0.

class `typing.AbstractSet` (*Sized*, *Collection*[*T_co*])

Una versión genérica de `collections.abc.Set`.

class `typing.MutableSet` (*AbstractSet*[*T*])

Una versión genérica de `collections.abc.MutableSet`.

class `typing.Mapping` (*Sized*, *Collection*[*KT*], *Generic*[*VT_co*])

Una versión genérica de `collections.abc.Mapping`. Este tipo se puede usar de la siguiente manera:

```
def get_position_in_index(word_list: Mapping[str, int], word: str) -> int:
    return word_list[word]
```

class `typing.MutableMapping` (*Mapping*[*KT*, *VT*])

Una versión genérica de `collections.abc.MutableMapping`.

class `typing.Sequence` (*Reversible*[*T_co*], *Collection*[*T_co*])

Una versión genérica de `collections.abc.Sequence`.

class `typing.MutableSequence` (*Sequence*[*T*])

Una versión genérica de `collections.abc.MutableSequence`.

class `typing.ByteString` (*Sequence*[*int*])

Una versión genérica de `collections.abc.ByteString`.

This type represents the types `bytes`, `bytearray`, and `memoryview` of byte sequences.

Como abreviación para este tipo, `bytes` se puede usar para anotar argumentos de cualquiera de los tipos mencionados arriba.

class `typing.Deque` (*deque*, *MutableSequence*[*T*])

Una versión genérica de `collections.deque`.

Nuevo en la versión 3.5.4.

Nuevo en la versión 3.6.1.

class `typing.List` (*list*, *MutableSequence*[*T*])

Versión genérica de `list`. Útil para anotar tipos de retorno. Para anotar argumentos es preferible usar un tipo abstracto de colección como `Sequence` o `Iterable`.

Este tipo se puede usar del siguiente modo:

```
T = TypeVar('T', int, float)

def vec2(x: T, y: T) -> List[T]:
    return [x, y]

def keep_positives(vector: Sequence[T]) -> List[T]:
    return [item for item in vector if item > 0]
```

class `typing.Set` (*set*, *MutableSet*[*T*])

Una versión genérica de `builtins.set`. Útil para anotar tipos de retornos. Para anotar argumentos es preferible usar un tipo abstracto de colección como `AbstractSet`.

class `typing.FrozenSet` (*frozenset*, *AbstractSet*[*T_co*])

Una versión genérica de `builtins.frozenset`.

class `typing.MappingView` (*Sized*, *Iterable*[*T_co*])

Una versión genérica de `collections.abc.MappingView`.

class `typing.KeysView` (*MappingView*[*KT_co*], *AbstractSet*[*KT_co*])

Una versión genérica de `collections.abc.KeysView`.

class `typing.ItemsView` (*MappingView*, *Generic*[*KT_co*, *VT_co*])

Una versión genérica de `collections.abc.ItemsView`.

class `typing.ValuesView` (*MappingView*[*VT_co*])

Una versión genérica de `collections.abc.ValuesView`.

class `typing.Awaitable` (*Generic*[*T_co*])

Una versión genérica de `collections.abc.Awaitable`.

Nuevo en la versión 3.5.2.

class `typing.Coroutine` (*Awaitable*[*V_co*], *Generic*[*T_co*, *T_contra*, *V_co*])

Una versión genérica de `collections.abc.Coroutine`, y orden de las variables de tipo se corresponde con aquellas de `Generator`, por ejemplo:

```
from typing import List, Coroutine
c = None # type: Coroutine[List[str], str, int]
...
x = c.send('hi') # type: List[str]
async def bar() -> None:
    x = await c # type: int
```

Nuevo en la versión 3.5.3.

class `typing.AsyncIterable` (*Generic*[*T_co*])

Una versión genérica de `collections.abc.AsyncIterable`.

Nuevo en la versión 3.5.2.

class `typing.AsyncIterator` (*AsyncIterable*[*T_co*])

Una versión genérica de `collections.abc.AsyncIterator`.

Nuevo en la versión 3.5.2.

class `typing.ContextManager` (*Generic*[*T_co*])

Una versión genérica de `contextlib.AbstractContextManager`.

Nuevo en la versión 3.5.4.

Nuevo en la versión 3.6.0.

class `typing.AsyncContextManager` (*Generic*[*T_co*])

Una versión genérica de `contextlib.AbstractAsyncContextManager`.

Nuevo en la versión 3.5.4.

Nuevo en la versión 3.6.2.

class `typing.Dict` (*dict*, *MutableMapping*[*KT*, *VT*])

Una versión genérica de `dict`. Útil para anotar tipos de retorno. Para anotar argumentos es preferible usar un tipo abstracto de colección como `Mapping`.

Este tipo se puede usar de la siguiente manera:

```
def count_words(text: str) -> Dict[str, int]:
    ...
```

class `typing.DefaultDict` (*collections.defaultdict*, *MutableMapping*[*KT*, *VT*])

Una versión genérica de `collections.defaultdict`.

Nuevo en la versión 3.5.2.

class `typing.OrderedDict` (*collections.OrderedDict*, *MutableMapping*[*KT*, *VT*])

Una versión genérica de `collections.OrderedDict`.

Nuevo en la versión 3.7.2.

class `typing.Counter` (*collections.Counter*, *Dict*[*T*, *int*])

Una versión genérica de `collections.Counter`.

Nuevo en la versión 3.5.4.

Nuevo en la versión 3.6.1.

class `typing.ChainMap` (*collections.ChainMap*, *MutableMapping*[*KT*, *VT*])

Una versión genérica de `collections.ChainMap`.

Nuevo en la versión 3.5.4.

Nuevo en la versión 3.6.1.

class `typing.Generator` (*Iterator*[*T_co*], *Generic*[*T_co*, *T_contra*, *V_co*])

Un generador puede ser anotado con el tipo genérico `Generator`[*YieldType*, *SendType*, *ReturnType*]. Por ejemplo:

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

Nótese que en contraste con muchos otros genéricos en el módulo `typing`, el `SendType` de `Generator` se comporta como contravariante, no covariante ni invariante.

Si tu generador solo retornará valores con `yield`, establece `SendType` y `ReturnType` como `None`:

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

Opcionalmente, anota tu generador con un tipo de retorno de `Iterable`[*YieldType*] o `Iterator`[*YieldType*]:

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

class `typing.AsyncGenerator` (`AsyncIterator`[`T_co`], `Generic`[`T_co`, `T_contra`])

Un generador asíncrono se puede anotar con el tipo genérico `AsyncGenerator`[`YieldType`, `SendType`]. Por ejemplo:

```
async def echo_round() -> AsyncGenerator[int, float]:
    sent = yield 0
    while sent >= 0.0:
        rounded = await round(sent)
        sent = yield rounded
```

A diferencia de los generadores normales, los generadores asíncronos no pueden retornar un valor, por lo que no hay un parámetro de tipo “Return Type”. Igual que `Generator`, `SendType` se comporta como contravariante.

Si tu generador solo retornará valores con `yield`, establece el `SendType` como `None`:

```
async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

Opcionalmente, anota el generador con un tipo de retorno `AsyncIterable`[`YieldType`] o `AsyncIterator`[`YieldType`]:

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

Nuevo en la versión 3.6.1.

class `typing.Text`

`Text` es un alias para `str`. Ésta disponible para proporcionar un mecanismo compatible hacia delante para código en Python 2: en Python 2, `Text` es un alias de `unicode`.

Úsese `Text` para indicar que un valor debe contener una cadena de texto Unicode de manera que sea compatible con Python 2 y Python 3:

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

Nuevo en la versión 3.5.2.

class `typing.IO`

class `typing.TextIO`

class `typing.BinaryIO`

El tipo genérico `IO`[`AnyStr`] y sus subclases `TextIO`(`IO`[`str`]) y `BinaryIO`(`IO`[`bytes`]) representan tipos de flujo de entrada/salida (*I/O streams*) como los usados por la función `open()`.

class `typing.Pattern`

class `typing.Match`

Estos alias de tipo se corresponden con los tipos de retorno de `re.compile()` y `re.match()`. Estos tipos (y sus correspondientes funciones) son genéricos en `AnyStr` y se puede especificar escribiendo `Pattern`[`str`], `Pattern`[`bytes`], `Match`[`str`], o `Match`[`bytes`].

class `typing.NamedTuple`

Versión para anotación de tipos de `collections.namedtuple()`.

Uso:

```
class Employee(NamedTuple):
    name: str
    id: int
```

Esto es equivalente a:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

Para proporcionar a un campo un valor por defecto se puede asignar en el cuerpo de la clase:

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

Los campos con un valor por defecto deben ir después de los campos sin valor por defecto.

La clase resultante tiene un atributo extra `__annotations__` que proporciona un diccionario que mapea el nombre de los campos con su tipo. (Los nombres de los campos están en el atributo `_fields` y sus valores por defecto en el atributo `_field_defaults`, ambos parte de la API de `namedtuple`.)

Las subclases de `NamedTuple` también pueden tener *docstrings* y métodos:

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

Uso retrocompatible:

```
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

Distinto en la versión 3.6: Soporte añadido para la sintaxis de anotación de variables propuesto en [PEP 526](#).

Distinto en la versión 3.6.1: Soporte añadido para valores por defecto, métodos y *docstrings*.

Deprecated since version 3.8, will be removed in version 3.9: Convierte en obsoleto el atributo `_field_types` en favor del atributo más estándar `__annotations__`, que tiene la misma información.

Distinto en la versión 3.8: Los atributos `_field_types` y `__annotations__` son simples diccionarios en vez de instancias de `OrderedDict`.

class `typing.TypedDict` (*dict*)

Un espacio de nombres con un tipado simple. En tiempo de ejecución es equivalente a un simple *dict*.

`TypedDict` crea un tipo de diccionario que espera que todas sus instancias tenga un cierto conjunto de claves, donde cada clave está asociada con un valor de un tipo determinado. Esta exigencia no se comprueba en tiempo de ejecución y solo es impuesta por validadores de tipo. Uso:

```
class Point2D(TypedDict):
    x: int
    y: int
    label: str

a: Point2D = {'x': 1, 'y': 2, 'label': 'good'} # OK
b: Point2D = {'z': 3, 'label': 'bad'}          # Fails type check

assert Point2D(x=1, y=2, label='first') == dict(x=1, y=2, label='first')
```

Se puede acceder a la información de tipo para introspección vía `Point2D.__annotations__` y `Point2D.__total__`. Para poder utilizar estas características con versiones antiguas de Python que no soporten [PEP 526](#), `TypedDict` proporciona adicionalmente dos formatos sintácticos equivalentes:


```
Point2D = TypedDict('Point2D', x=int, y=int, label=str)
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': str})
```

Por defecto, todas las claves deben estar presentes en un `TypedDict`. Es posible anularlo especificando el parámetro *total*. De esta manera:

```
class point2D(TypedDict, total=False):
    x: int
    y: int
```

Esto significa que un `TypedDict` `point2D` puede omitir cualquiera de sus claves. Un validador de tipos solo se esperará un valor literal *False* o *True* como valor del argumento *total*. *True* es el valor por defecto, y hace que todos los elementos definidos en el cuerpo de la clases sean obligatorios.

Véase [PEP 589](#) para más ejemplos y reglas detalladas del uso de `TypedDict`.

Nuevo en la versión 3.8.

`class typing.ForwardRef`

Una clase usada para la representación de tipado interno de referencias directas en forma de cadenas de texto. Por ejemplo, `List["SomeClass"]` es transformado implícitamente en `List[ForwardRef("SomeClass")]`. Esta clase no debe ser instanciada por el usuario, pero puede ser usada por herramientas de introspección.

Nuevo en la versión 3.7.4.

`typing.NewType(name, tp)`

A helper function to indicate a distinct type to a typechecker, see [NewType](#). At runtime it returns a function that returns its argument. Usage:

```
UserId = NewType('UserId', int)
first_user = UserId(1)
```

Nuevo en la versión 3.5.2.

`typing.cast(typ, val)`

Convertir un valor a su tipo.

Esto retorna el valor sin modificar. Para el validador de tipos esto indica que el valor de retorno tiene el tipo señalado pero, de manera intencionada, no se comprobará en tiempo de ejecución (para maximizar la velocidad).

`typing.get_type_hints(obj[, globals[, locals]])`

Retorna un diccionario que contiene indicaciones de tipo para una función, método, módulo o objeto clase.

Habitualmente, esto es lo mismo que `obj.__annotations__`. Además, las referencias indicadas como cadenas de texto se gestionan evaluándolas en los espacios de nombres “globals” y *locals*. Si es necesario, se añade “Optional[t]” para anotar una función o método, si se establece *None* como valor por defecto. Para una clase *C*, se retorna un diccionario construido por la combinación de `__annotations__` y `C.__mro` en orden inverso.

`typing.get_origin(tp)`

`typing.get_args(tp)`

Provee introspección básica para tipos genéricos y construcciones especiales de tipado.

Para un objeto de tipado de la forma `X[Y, Z, ...]`, estas funciones retornan `X` y `(Y, Z, ...)`. Si `X` es un alias genérico para una clase *built-in* o una clase de [collections](#), se normaliza a su clase original. Los objetos no soportados retornan *None* y `()`. Ejemplos:

```
assert get_origin(Dict[str, int]) is dict
assert get_args(Dict[int, str]) == (int, str)

assert get_origin(Union[int, str]) is Union
assert get_args(Union[int, str]) == (int, str)
```

Nuevo en la versión 3.8.

`@typing.overload`

El decorador `@overload` permite describir funciones y métodos que soportan diferentes combinaciones de tipos de argumento. A una serie de definiciones decoradas con `@overload` debe seguir exactamente una definición no decorada con `@overload` (para la misma función o método). Las definiciones decoradas con `@overload` son solo para beneficio del validador de tipos, ya que serán sobrescritas por la definición no decorada con `@overload`. Esta última se usa en tiempo de ejecución y debería ser ignorada por el validador de tipos. En tiempo de ejecución, llamar a una función decorada con `@overload` lanzará directamente `NotImplementedError`. Un ejemplo de sobrecarga que proporciona un tipo más preciso se puede expresar con una unión o una variable de tipo:

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> Tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    <actual implementation>
```

Véase [PEP 484](#) para más detalle, y compárese con otras semánticas de tipado.

`@typing.final`

Un decorador que indica a los validadores de tipos que el método decorado no se puede sobrescribir, o que la clase decorada no se puede derivar (*subclass*). Por ejemplo:

```
class Base:
    @final
    def done(self) -> None:
        ...
class Sub(Base):
    def done(self) -> None: # Error reported by type checker
        ...

@final
class Leaf:
    ...
class Other(Leaf): # Error reported by type checker
    ...
```

No hay comprobación en tiempo de ejecución para estas propiedades. Véase [PEP 591](#) para más detalles.

Nuevo en la versión 3.8.

`@typing.no_type_check`

Un decorador para indicar que la anotaciones no deben ser comprobadas como indicadores de tipo.

Esto funciona como un *decorator* (decorador) de clase o función. Con una clase, se aplica recursivamente a todos los métodos definidos en dichas clase (pero no a los métodos definidos en sus superclases y subclases).

Esto modifica la función o funciones *in situ*.

`@typing.no_type_check_decorator`

Un decorador que asigna a otro decorador el efecto de `no_type_check()` (no comprobar tipo).

Esto hace que el decorador decorado añada el efecto de `no_type_check()` a la función decorada.

`@typing.type_check_only`

Un decorador que marca una clase o función como no disponible en tiempo de ejecución.

Este decorador no está disponible en tiempo de ejecución. Existe principalmente para marcar clases que se definen en archivos *stub* para cuando una implementación retorna una instancia de una clase privada:

```
@type_check_only
class Response: # private or not available at runtime
    code: int
    def get_header(self, name: str) -> str: ...

def fetch_response() -> Response: ...
```

Nótese que no se recomienda retornar instancias de clases privadas. Normalmente es preferible convertirlas en clases públicas.

@typing.runtime_checkable

Marca una clase protocolo como aplicable en tiempo de ejecución (lo convierte en un *runtime protocol*).

Such a protocol can be used with `isinstance()` and `issubclass()`. This raises `TypeError` when applied to a non-protocol class. This allows a simple-minded structural check, very similar to «one trick ponies» in `collections.abc` such as `Iterable`. For example:

```
@runtime_checkable
class Closable(Protocol):
    def close(self): ...

assert isinstance(open('/some/file'), Closable)
```

Atención: esto solo comprobará la presencia de los métodos requeridos, no su interfaz.

Nuevo en la versión 3.8.

typing.Any

Tipo especial que indica un tipo sin restricciones.

- Todos los tipos son compatibles con `Any`.
- `Any` es compatible con todos los tipos.

typing.NoReturn

Tipo especial que indica que una función nunca retorna un valor. Por ejemplo:

```
from typing import NoReturn

def stop() -> NoReturn:
    raise RuntimeError('no way')
```

Nuevo en la versión 3.5.4.

Nuevo en la versión 3.6.2.

typing.Union

Tipo unión; `Union[X, Y]` significa que o bien X o bien Y.

Para definir una unión, úsese p. ej. `Union[int, str]`. Más detalles:

- Los argumentos deben ser tipos y haber al menos uno.
- Las uniones de uniones se simplifican (se aplanan), p. ej.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Las uniones con un solo argumento se eliminan, p. ej.:

```
Union[int] == int # The constructor actually returns int
```

- Argumentos repetidos se omiten, p. ej.:

```
Union[int, str, int] == Union[int, str]
```

- Cuando se comparan uniones, el orden de los argumentos se ignoran, p. ej.:

```
Union[int, str] == Union[str, int]
```

- No se puede derivar (*subclass*) o instanciar una unión.
- No es posible escribir `Union[X][Y]`.
- Se puede usar `Optional[X]` como una versión corta de `Union[X, None]`.

Distinto en la versión 3.7: No elimina subclases explícitas de una unión en tiempo de ejecución.

`typing.Optional`

Tipo `Optional`.

`Optional[X]` es equivalente a `Union[X, None]`.

Nótese que no es lo mismo que un argumento opcional, que es aquel que tiene un valor por defecto. Un argumento opcional con un valor por defecto no necesita el indicador `Optional` en su anotación de tipo simplemente por que sea opcional. Por ejemplo:

```
def foo(arg: int = 0) -> None:
    ...
```

Por otro lado, si se permite un valor `None`, es apropiado el uso de `Optional`, independientemente de que sea opcional o no. Por ejemplo:

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

`typing.Tuple`

El tipo `Tuple`, `Tuple[X, Y]` es el tipo de una tupla de dos ítems con el primer ítem de tipo `X` y el segundo de tipo `Y`. El tipo de una tupla vacía se puede escribir así: `Tuple[()]`.

Ejemplo: `Tuple[T1, T2]` es una tupla de dos elementos con sus correspondientes variables de tipo `T1` y `T2`. `Tuple[int, float, str]` es una tupla con un número entero, un número de punto flotante y una cadena de texto.

Para especificar una tupla de longitud variable y tipo homogéneo, se usan puntos suspensivos, p. ej. `Tuple[int, ...]`. Un simple *`Tuple`* es equivalente a `Tuple[Any, ...]` y, a su vez, a *`tuple`*.

`typing.Callable`

Tipo `Callable` (invocable); `Callable[[int], str]` es una función de `(int) -> str`.

La sintaxis de subscripción (con corchetes `[]`) debe usarse siempre con dos valores: la lista de argumentos y el tipo de retorno. La lista de argumentos debe ser una lista de tipos o unos puntos suspensivos; el tipo de retorno debe ser un único tipo.

No existe una sintaxis para indicar argumentos opcionales o con clave (*keyword*); tales funciones rara vez se utilizan como tipos para llamadas. `Callable[..., ReturnType]` (puntos suspensivos) se puede usar para indicar que un *callable* admite un número indeterminado de argumentos y retorna `ReturnType`. Un simple *`Callable`* es equivalente a `Callable[..., Any]` y, a su vez, a *`collections.abc.Callable`*.

`typing.Literal`

Un tipo que puede ser utilizado para indicar a los validadores de tipos que una variable o un parámetro de una función tiene un valor equivalente al valor literal proveído (o uno de los proveídos). Por ejemplo:

```
def validate_simple(data: Any) -> Literal[True]: # always returns True
    ...

MODE = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: MODE) -> str:
    ...

open_helper('/some/path', 'r') # Passes type check
open_helper('/other/path', 'typo') # Error in type checker
```

`Literal[...]` no puede ser derivado. En tiempo de ejecución, se permite un valor arbitrario como argumento de tipo de `Literal[...]`, pero los validadores de tipos pueden imponer sus restricciones. Véase [PEP 585](#) para más detalles sobre tipos literales.

Nuevo en la versión 3.8.

`typing.ClassVar`

Construcción especial para tipado para marcar variables de clase.

Tal y como introduce [PEP 526](#), una anotación de variable rodeada por `ClassVar` indica que la intención de un atributo dado es ser usado como variable de clase y que no debería ser modificado en las instancias de esa misma clase. Uso:

```
class Starship:
    stats: ClassVar[Dict[str, int]] = {} # class variable
    damage: int = 10                    # instance variable
```

`ClassVar` solo acepta tipos y no admite más niveles de subíndices.

`ClassVar` no es un clase en sí misma, y no debe ser usado con `isinstance()` o `issubclass()`. `ClassVar` no modifica el comportamiento de Python en tiempo de ejecución pero puede ser utilizado por validadores de terceros. Por ejemplo, un validador de tipos puede marcar el siguiente código como erróneo:

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
Starship.stats = {}    # This is OK
```

Nuevo en la versión 3.5.3.

`typing.Final`

Un construcción especial para tipado que indica a los validadores de tipo que un nombre no puede ser reasignado o sobrescrito en una subclase. Por ejemplo:

```
MAX_SIZE: Final = 9000
MAX_SIZE += 1 # Error reported by type checker

class Connection:
    TIMEOUT: Final[int] = 10

class FastConnector(Connection):
    TIMEOUT = 1 # Error reported by type checker
```

No hay comprobación en tiempo de ejecución para estas propiedades. Véase [PEP 591](#) para más detalles.

Nuevo en la versión 3.8.

`typing.AnyStr`

`AnyStr` es una variable de tipo definida como `AnyStr = TypeVar('AnyStr', str, bytes)`.

Su objetivo es ser usada por funciones que pueden aceptar cualquier tipo de cadena de texto sin permitir mezclar diferentes tipos al mismo tiempo. Por ejemplo:

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat(u"foo", u"bar") # Ok, output has type 'unicode'
concat(b"foo", b"bar") # Ok, output has type 'bytes'
concat(u"foo", b"bar") # Error, cannot mix unicode and bytes
```

`typing.TYPE_CHECKING`

Una constante especial que se asume como cierta (`True`) por validadores estáticos de tipos de terceros. Es falsa (`False`) en tiempo de ejecución. Uso:

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

Nótese que la primera anotación de tipo debe estar rodeada por comillas, convirtiéndola en una «referencia directa», para ocultar al intérprete la referencia `expensive_mod` en tiempo de ejecución. Las anotaciones de tipo para variables locales no se evalúan, así que la segunda anotación no necesita comillas.

Nuevo en la versión 3.5.2.

26.2 pydoc — Generador de documentación y Sistema de ayuda en línea

Código fuente: [Lib/pydoc.py](#)

El módulo `pydoc` genera automáticamente documentación de módulos de Python. La documentación se puede presentar como páginas de texto en la consola, servidos en un buscador web, o guardados en archivos HTML.

Para módulos, clases, funciones y métodos, la documentación mostrada es derivada del *docstring* (i.e. el atributo `__doc__`) del objeto, y recursivamente de sus miembros que se puedan documentar. Si no existe el *docstring*, `pydoc` trata de obtener una descripción del bloque de comentarios arriba de la definición de la clase, función o método en el archivo fuente, o encima del módulo (véase `inspect.getcomments()`).

La función incorporada `help()` invoca el sistema de ayuda en línea en el interpretador interactivo, que usa `pydoc` para generar su documentación como texto en la consola. La misma documentación del texto se puede ver desde afuera del interpretador de python al ejecutar **pydoc** como un script en la consola del sistema operativo. Por ejemplo, ejecutando

```
pydoc sys
```

en la entrada de la consola mostrará la documentación sobre el módulo `sys`, en un estilo similar a las páginas del manual que se muestran por el comando **man** de Unix. El argumento de **pydoc** puede ser el nombre de una función, módulo, o paquete, o una referencia con puntos (*dotted reference*) de una clase, método, o función dentro de un módulo o módulo en un paquete. Si el argumento de **pydoc** se parece a una ruta (*path*) (es decir, que contiene el separador de rutas para tu sistema operativo como una barra en Unix), y hace referencia a un archivo fuente de Python existente, entonces se produce la documentación para ese archivo.

Nota: Para encontrar los objetos y su documentación, `pydoc` importa el módulo o los módulos que serán documentados. Por lo tanto, cualquier código a nivel de módulo va a ser ejecutado en esa ocasión. Use `if __name__ == '__main__':` como protección para sólo ejecutar código cuando un archivo es invocado como un script y no sólo importado.

Cuando se imprime la salida a la consola, **pydoc** intenta paginar la salida para una lectura más fácil. Si la variable de entorno `PAGER` está puesta, **pydoc** usará su valor como el programa de paginación.

Especificar un bandera (*flag*) `-w` antes del argumento hará que la documentación HTML sea escrita en un archivo de la carpeta actual, en vez de mostrar el texto en la consola.

Especificar una bandera (*flag*) `-k` antes del argumento buscará las líneas de la sinopsis de todos los módulos disponibles por la palabra clave (*keyword*) dada como argumento, de nuevo en una manera similar al comando **man** de Unix. La sinopsis de un módulo es la primera línea de su cadena de documentación.

Puedes usar **pydoc** para empezar un servidor HTTP en la máquina local que va a servir la documentación para buscadores Web invitados. **pydoc -p 1234** empezará un servidor HTTP en el puerto 1234, permitiéndote buscar

la documentación en `http://localhost:1234/` en tu buscador de preferencia. Especificar 0 como el número de puerto seleccionará un puerto arbitrario no usado.

pydoc -n <hostname> empezará el servidor escuchando al *hostname* (nombre del servidor) dado. Por defecto, el nombre del servidor es “localhost” pero si quieres que se llegue al servidor desde otras máquinas, quizás quieras cambiar el nombre por el cual el servidor responde. Durante el desarrollo esto es especialmente útil si quieres correr *pydoc* desde dentro de un contenedor.

pydoc -b empezará un servidor y adicionalmente abrirá un buscador web con una página del índice del módulo. Cada página servida tiene una barra de navegación arriba donde puedes obtener (*Get*) ayuda sobre una sección individual, buscar (*Search*) todos los módulos con una palabra clave y sus sinopsis, e ir a las páginas del índice del módulo (*Module index*), temas (*Topics*), y palabras clave (*Keywords*).

Cuando **pydoc** genera la documentación, se usa el entorno y ruta actual para localizar los módulos. Así, invocar **pydoc spam** precisamente documenta la versión del módulo que tu obtendrías si empezaras el interpretador de Python y escribieras `import spam`.

Se asume que la documentación de los módulos principales residen en `https://docs.python.org/X.Y/library/` donde X y Y son la versión mayor y menor de tu interpretador de Python. Esto puede ser sobre-escrito al poner a la variable de entorno `PYTHONDOS` una URL diferente o un directorio local conteniendo la páginas de referencia.

Distinto en la versión 3.2: Se añadió la opción `-b`.

Distinto en la versión 3.3: La opción de la línea de comandos `-g` se eliminó.

Distinto en la versión 3.4: *pydoc* ahora usa `inspect.signature()` en vez de `inspect.getfullargspec()` para extraer la información distintiva de los invocables (*callables*).

Distinto en la versión 3.7: Se añadió la opción `-n`.

26.3 doctest — Test interactive Python examples

Source code: `Lib/doctest.py`

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.
```

(continué en la próxima página)

(proviene de la página anterior)

```

>>> [factorial(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
>>> factorial(30)
265252859812191058636308480000000
>>> 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)
265252859812191058636308480000000

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
Trying:
    [factorial(n) for n in range(6)]
Expecting:

```

(continué en la próxima página)

(proviene de la página anterior)

```
[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`.

26.3.1 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).

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*.

26.3.2 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`.

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](#).

26.3.3 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.

CPython implementation detail: Prior to version 3.4, extension modules written in C were not fully searched by doctest.

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!!!")
...
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.
- 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.

- 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
```

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 `globals=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:

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.

```
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.

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
Traceback (most recent call last):
  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*, and may be passed to the doctest command line interface via the `-o` option.

Nuevo en la versión 3.4: The `-o` command line option.

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 doc-test 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».

Distinto en la versión 3.2: `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.

`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_CDIF`

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.FAIL_FAST`

When specified, exit after the first failing example and don't attempt to run the remaining examples. Thus, the number of failures reported will be at most 1. This flag may be useful during debugging, since examples after the first failure won't even produce debugging output.

The doctest command line accepts the option `-f` as a shorthand for `-o FAIL_FAST`.

Nuevo en la versión 3.4.

`doctest.REPORTING_FLAGS`

A bitmask or'ing together all the reporting flags above.

There is also a way to register new option flag names, though 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')
```

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(list(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(list(range(20)))
[0, 1, ..., 18, 19]
```

Multiple directives can be used on a single physical line, separated by commas:

```
>>> print(list(range(20)))
[0, 1, ..., 18, 19]
```

If multiple directive comments are used for a single example, then they are combined:

```
>>> print(list(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(list(range(5)) + list(range(10, 20)) + list(range(30, 40)))
...
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

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.

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 set, Python doesn't guarantee that the element is printed in any particular order, so a test like

```
>>> foo()
{"Hermione", "Harry"}
```

is vulnerable! One workaround is to do

```
>>> foo() == {"Hermione", "Harry"}
True
```

instead. Another is to do

```
>>> d = sorted(foo())
>>> d
['Harry', 'Hermione']
```

Nota: Before Python 3.6, when printing a dict, Python did not guarantee that the key-value pairs was printed in any particular order.

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.

26.3.4 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=True, name=None, package=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, parser=DocTestParser(), encoding=None)`

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` (default value 0) takes the bitwise OR of 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.

`doctest.testmod(m=None, name=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, exclude_empty=False)`

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__`.

```
doctest.run_docstring_examples(f, globs, verbose=False, name="NoName", compileflags=None,
                               optionflags=0)
```

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.

26.3.5 unittest API

As your collection of doctest'ed modules grows, you'll want a way to run all their doctests systematically. `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=True, package=None, setUp=None, tearDown=None,
                     globs=None, optionflags=0, parser=DocTestParser(), encoding=None)
```

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.

The global `__file__` is added to the globals provided to doctests loaded from a text file using *DocFileSuite()*.

`doctest.DocTestSuite` (*module=None, globs=None, extraglobs=None, test_finder=None, setUp=None, tearDown=None, checker=None*)

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.

This function uses the same search technique as *testmod()*.

Distinto en la versión 3.5: *DocTestSuite()* returns an empty *unittest.TestSuite* if *module* contains no docstrings instead of raising *ValueError*.

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` takes the bitwise OR of 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.

26.3.6 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.

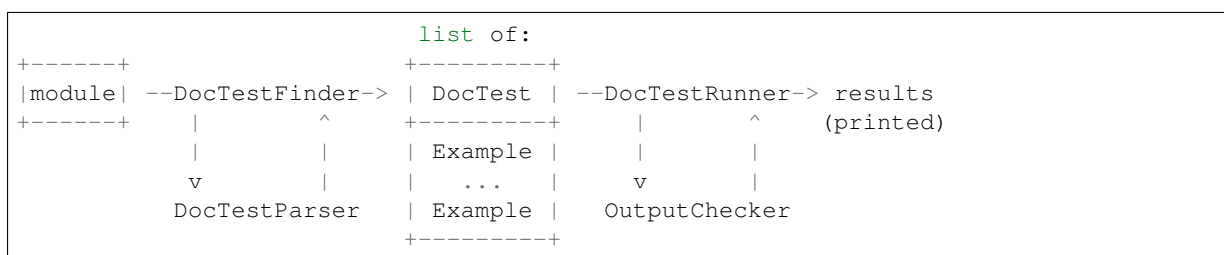
The advanced API revolves around two container classes, which are used to store the interactive examples extracted from `doctest` cases:

- *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.

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=None, lineno=0, indent=0, options=None*)

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.

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=False*, *parser=DocTestParser()*, *recurse=True*, *exclude_empty=True*)

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 be extracted from modules, classes, functions, 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.

`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.

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*='<string>')

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*='<string>')

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*=None, *verbose*=None, *optionflags*=0)

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*.

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=None, out=None, clear_globs=True*)

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=None*)

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.

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.

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*.

26.3.7 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) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
1      def f(x):
2  ->      g(x*2)
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

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.

`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.

`doctest.debug(module, name, pm=False)`

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 `exec()` call to `pdb.run()`.

`doctest.debug_src(src, pm=False, globs=None)`

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.

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=None, verbose=None, optionflags=0*)

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()`.

26.3.8 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.FAIL_FAST
    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))
```

26.4 unittest — Unit testing framework

Source code: `Lib/unittest/__init__.py`

(If you are already familiar with the basic concepts of testing, you might want to skip to [the list of assert methods](#).)

The `unittest` unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It 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.

To achieve this, `unittest` supports some important concepts in an object-oriented way:

test fixture A *test fixture* represents the preparation needed to perform one or more tests, and any associated 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 individual 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.

Ver también:

Module *doctest* Another test-support module with a very different flavor.

Simple Smalltalk Testing: With Patterns Kent Beck's original paper on testing frameworks using the pattern shared by *unittest*.

pytest Third-party *unittest* framework 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.

The script `Tools/unittestgui/unittestgui.py` in the Python source distribution is a GUI tool for test discovery and execution. This is intended largely for ease of use for those new to unit testing. For production environments it is recommended that tests be driven by a continuous integration system such as [Buildbot](#), [Jenkins](#) or [Travis-CI](#), or [AppVeyor](#).

26.4.1 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'])
        # 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
```

Passing the `-v` option to your test script will instruct `unittest.main()` to enable a higher level of verbosity, and produce 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.

26.4.2 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.

Test modules can be specified by file path as well:

```
python -m unittest tests/test_something.py
```

This allows you to use the shell filename completion to specify the test module. The file specified must still be importable as a module. The path is converted to a module name by removing the “.py” and converting path separators into “.”. If you want to execute a test file that isn’t importable as a module you should execute the file directly instead.

You can run tests with more detail (higher verbosity) by passing in the `-v` flag:

```
python -m unittest -v test_module
```

When executed without arguments *Test Discovery* is started:

```
python -m unittest
```

For a list of all the command-line options:

```
python -m unittest -h
```

Distinto en la versión 3.2: 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.

-k

Only run test methods and classes that match the pattern or substring. This option may be used multiple times, in which case all test cases that match of the given patterns are included.

Patterns that contain a wildcard character (*) are matched against the test name using *fnmatch.fnmatchcase()*; otherwise simple case-sensitive substring matching is used.

Patterns are matched against the fully qualified test method name as imported by the test loader.

For example, `-k foo` matches `foo_tests.SomeTest.test_something`, `bar_tests.SomeTest.test_foo`, but not `bar_tests.FooTest.test_something`.

--locals

Show local variables in tracebacks.

Nuevo en la versión 3.2: The command-line options `-b`, `-c` and `-f` were added.

Nuevo en la versión 3.5: The command-line option `--locals`.

Nuevo en la versión 3.7: The command-line option `-k`.

The command line can also be used for test discovery, for running all of the tests in a project or just a subset.

26.4.3 Test Discovery

Nuevo en la versión 3.2.

Unittest supports simple test discovery. In order to be compatible with test discovery, all of the test files must be modules or packages (including *namespace 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
```

Nota: As a shortcut, `python -m unittest` is the equivalent of `python -m unittest discover`. If you want to pass arguments to test discovery the `discover` sub-command must be used explicitly.

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.

Prudencia: 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*.

Distinto en la versión 3.4: Test discovery supports *namespace packages* for start directory. Note that you need to the top level directory too. (e.g. `python -m unittest discover -s root/namespace -t root`).

26.4.4 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 `unittest.TestCase` instances. To make your own test cases you must write subclasses of `TestCase` or use `FunctionTestCase`.

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 implement a test method (i.e. a method whose name starts with `test`) in order to perform specific testing code:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

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 with an explanatory message, and `unittest` will identify the test case as a *failure*. Any other exceptions will be treated as *errors*.

Tests can be numerous, and their set-up can be repetitive. Luckily, we can factor out set-up code by implementing a method called `setUp()`, which the testing framework will automatically call for every single test we run:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
```

(continué en la próxima página)

(proviene de la página anterior)

```
self.widget = Widget('The widget')

def test_default_widget_size(self):
    self.assertEqual(self.widget.size(), (50,50),
                      'incorrect default size')

def test_widget_resize(self):
    self.widget.resize(100,150)
    self.assertEqual(self.widget.size(), (100,150),
                      'wrong size after resize')
```

Nota: The order in which the various tests will be run is determined by sorting the test method names with respect to the built-in ordering for strings.

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 test method will not be executed.

Similarly, we can provide a `tearDown()` method that tidies up after the test method has been run:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

If `setUp()` succeeded, `tearDown()` will be run whether the test method succeeded or not.

Such a working environment for the testing code is called a *test fixture*. A new `TestCase` instance is created as a unique test fixture used to execute each individual test method. Thus `setUp()`, `tearDown()`, and `__init__()` will be called once per test.

It is recommended that you use `TestCase` implementations to group tests together according to the features they test. `unittest` provides a mechanism for this: the *test suite*, represented by `unittest`'s `TestSuite` class. In most cases, calling `unittest.main()` will do the right thing and collect all the module's test cases for you and execute them.

However, should you want to customize the building of your test suite, you can do it yourself:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

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.

26.4.5 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, with optional set-up and tear-down methods:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

Nota: 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.

26.4.6 Skipping tests and expected failures

Nuevo en la versión 3.1.

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, calling `TestCase.skipTest()` within a `setUp()` or test method, or raising `SkipTest` directly.

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
```

(continué en la próxima página)

(proviene de la página anterior)

```
@unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
def test_windows_support(self):
    # windows specific testing code
    pass

def test_maybe_skipped(self):
    if not external_resource_available():
        self.skipTest("external resource not available")
    # test code that depends on the external resource
    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_maybe_skipped (__main__.MyTestCase) ... skipped 'external resource not_
↪available'
test_windows_support (__main__.MyTestCase) ... skipped 'requires Windows'

-----
Ran 4 tests in 0.005s

OK (skipped=4)
```

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 and exception 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 or error. If the test fails or errors it will be considered a success. If the test passes, it will be considered 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. Skipped modules will not have `setUpModule()` or `tearDownModule()` run.

26.4.7 Distinguishing test iterations using subtests

Nuevo en la versión 3.4.

When there are very small differences among your tests, for instance some parameters, unittest allows you to distinguish them inside the body of a test method using the `subTest()` context manager.

For example, the following test:

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

will produce the following output:

```
=====
FAIL: test_even (__main__.NumbersTest) (i=1)
-----
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

=====
FAIL: test_even (__main__.NumbersTest) (i=3)
-----
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

=====
FAIL: test_even (__main__.NumbersTest) (i=5)
-----
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

Without using a subtest, execution would stop after the first failure, and the error would be less easy to diagnose because the value of `i` wouldn't be displayed:

```
=====
FAIL: test_even (__main__.NumbersTest)
-----
Traceback (most recent call last):
```

(continué en la próxima página)

(proviene de la página anterior)

```
File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

26.4.8 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 logical test 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 tests, 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 base method: the method named *methodName*. In most uses of `TestCase`, you will neither change the *methodName* nor reimplement the default `runTest()` method.

Distinto en la versión 3.2: `TestCase` can be instantiated successfully without providing a *methodName*. This makes it easier to experiment with `TestCase` from the interactive interpreter.

`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.

Nuevo en la versión 3.2.

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.

Nuevo en la versión 3.2.

run (*result=None*)

Run the test, collecting the result into the *TestResult* 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 returned to *run()*'s caller.

The same effect may be had by simply calling the *TestCase* instance.

Distinto en la versión 3.3: Previous versions of *run* did not return the result. Neither did calling an instance.

skipTest (*reason*)

Calling this during a test method or *setUp()* skips the current test. See *Skipping tests and expected failures* for more information.

Nuevo en la versión 3.1.

subTest (*msg=None, **params*)

Return a context manager which executes the enclosed code block as a subtest. *msg* and *params* are optional, arbitrary values which are displayed whenever a subtest fails, allowing you to identify them clearly.

A test case can contain any number of subtest declarations, and they can be arbitrarily nested.

See *Distinguishing test iterations using subtests* for more information.

Nuevo en la versión 3.4.

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
<i>assertEqual(a, b)</i>	<code>a == b</code>	
<i>assertNotEqual(a, b)</i>	<code>a != b</code>	
<i>assertTrue(x)</i>	<code>bool(x) is True</code>	
<i>assertFalse(x)</i>	<code>bool(x) is False</code>	
<i>assertIs(a, b)</i>	<code>a is b</code>	3.1
<i>assertIsNot(a, b)</i>	<code>a is not b</code>	3.1
<i>assertIsNone(x)</i>	<code>x is None</code>	3.1
<i>assertIsNotNone(x)</i>	<code>x is not None</code>	3.1
<i>assertIn(a, b)</i>	<code>a in b</code>	3.1
<i>assertNotIn(a, b)</i>	<code>a not in b</code>	3.1
<i>assertIsInstance(a, b)</i>	<code>isinstance(a, b)</code>	3.2
<i>assertNotIsInstance(a, b)</i>	<code>not isinstance(a, b)</code>	3.2

All the assert methods accept a *msg* argument that, if specified, is used as the error message on failure (see also *longMessage*). Note that the *msg* keyword argument can be passed to *assertRaises()*, *assertRaisesRegex()*, *assertWarns()*, *assertWarnsRegex()* only when they are used as a context manager.

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 str 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*).

Distinto en la versión 3.1: Added the automatic calling of type-specific equality function.

Distinto en la versión 3.2: `assertMultiLineEqual()` added as the default type equality function for comparing strings.

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* are (or are not) the same object.

Nuevo en la versión 3.1.

assertIsNone (*expr, msg=None*)

assertIsNotNone (*expr, msg=None*)

Test that *expr* is (or is not) `None`.

Nuevo en la versión 3.1.

assertIn (*member, container, msg=None*)

assertNotIn (*member, container, msg=None*)

Test that *member* is (or is not) in *container*.

Nuevo en la versión 3.1.

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)`.

Nuevo en la versión 3.2.

It is also possible to check the production of exceptions, warnings, and log messages using the following methods:

Method	Checks that	New in
<code>assertRaises(exc, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> raises <i>exc</i>	
<code>assertRaisesRegex(exc, r, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> raises <i>exc</i> and the message matches regex <i>r</i>	3.1
<code>assertWarns(warn, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> raises <i>warn</i>	3.2
<code>assertWarnsRegex(warn, r, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> raises <i>warn</i> and the message matches regex <i>r</i>	3.2
<code>assertLogs(logger, level)</code>	The <code>with</code> block logs on <i>logger</i> with minimum <i>level</i>	3.4

assertRaises (*exception, callable, *args, **kwargs*)

assertRaises (*exception, *, msg=None*)

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* and possibly the *msg* arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertRaises(SomeException):
    do_something()
```

When used as a context manager, `assertRaises()` accepts the additional keyword argument *msg*.

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)
```

Distinto en la versión 3.1: Added the ability to use `assertRaises()` as a context manager.

Distinto en la versión 3.2: Added the `exception` attribute.

Distinto en la versión 3.3: Added the *msg* keyword argument when used as a context manager.

assertRaisesRegex (*exception, regex, callable, *args, **kwargs*)

assertRaisesRegex (*exception, regex, *, msg=None*)

Like `assertRaises()` but also tests that *regex* matches on the string representation of the raised exception. *regex* may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`. Examples:

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                        int, 'XYZ')
```

or:

```
with self.assertRaisesRegex(ValueError, 'literal'):
    int('XYZ')
```

Nuevo en la versión 3.1: Added under the name `assertRaisesRegexp`.

Distinto en la versión 3.2: Renamed to `assertRaisesRegex()`.

Distinto en la versión 3.3: Added the *msg* keyword argument when used as a context manager.

assertWarns (*warning, callable, *args, **kwargs*)

assertWarns (*warning, *, msg=None*)

Test that a warning is triggered when *callable* is called with any positional or keyword arguments that are also passed to `assertWarns()`. The test passes if *warning* is triggered and fails if it isn't. Any exception is an error. To catch any of a group of warnings, a tuple containing the warning classes may be passed as *warnings*.

If only the *warning* and possibly the *msg* arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertWarns(SomeWarning):
    do_something()
```

When used as a context manager, `assertWarns()` accepts the additional keyword argument *msg*.

The context manager will store the caught warning object in its `warning` attribute, and the source line which triggered the warnings in the `filename` and `lineno` attributes. This can be useful if the intention is to perform additional checks on the warning caught:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()
```

(continué en la próxima página)

(proviene de la página anterior)

```
self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

This method works regardless of the warning filters in place when it is called.

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Added the *msg* keyword argument when used as a context manager.

assertWarnsRegex (*warning, regex, callable, *args, **kwargs*)

assertWarnsRegex (*warning, regex, *, msg=None*)

Like [*assertWarns\(\)*](#) but also tests that *regex* matches on the message of the triggered warning. *regex* may be a regular expression object or a string containing a regular expression suitable for use by [*re.search\(\)*](#). Example:

```
self.assertWarnsRegex(DeprecationWarning,
                       r'legacy_function\(\) is deprecated',
                       legacy_function, 'XYZ')
```

or:

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Added the *msg* keyword argument when used as a context manager.

assertLogs (*logger=None, level=None*)

A context manager to test that at least one message is logged on the *logger* or one of its children, with at least the given *level*.

If given, *logger* should be a [*logging.Logger*](#) object or a *str* giving the name of a logger. The default is the root logger, which will catch all messages that were not blocked by a non-propagating descendent logger.

If given, *level* should be either a numeric logging level or its string equivalent (for example either "ERROR" or [*logging.ERROR*](#)). The default is [*logging.INFO*](#).

The test passes if at least one message emitted inside the *with* block matches the *logger* and *level* conditions, otherwise it fails.

The object returned by the context manager is a recording helper which keeps tracks of the matching log messages. It has two attributes:

records

A list of [*logging.LogRecord*](#) objects of the matching log messages.

output

A list of *str* objects with the formatted output of matching messages.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

Nuevo en la versión 3.4.

There are also other methods used to perform more specific checks, such as:

Method	Checks that	New in
<code>assertAlmostEqual(a, b)</code>	<code>round(a-b, 7) == 0</code>	
<code>assertNotAlmostEqual(a, b)</code>	<code>round(a-b, 7) != 0</code>	
<code>assertGreater(a, b)</code>	<code>a > b</code>	3.1
<code>assertGreaterEqual(a, b)</code>	<code>a >= b</code>	3.1
<code>assertLess(a, b)</code>	<code>a < b</code>	3.1
<code>assertLessEqual(a, b)</code>	<code>a <= b</code>	3.1
<code>assertRegex(s, r)</code>	<code>r.search(s)</code>	3.1
<code>assertNotRegex(s, r)</code>	<code>not r.search(s)</code>	3.2
<code>assertCountEqual(a, b)</code>	<i>a</i> and <i>b</i> have the same elements in the same number, regardless of their order.	3.2

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`.

Distinto en la versión 3.2: `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"
```

Nuevo en la versión 3.1.

assertRegex (*text*, *regex*, *msg*=None)

assertNotRegex (*text*, *regex*, *msg*=None)

Test that a *regex* search matches (or does not match) *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). *regex* may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`.

Nuevo en la versión 3.1: Added under the name `assertRegexpMatches`.

Distinto en la versión 3.2: The method `assertRegexpMatches()` has been renamed to `assertRegex()`.

Nuevo en la versión 3.2: `assertNotRegex()`.

Nuevo en la versión 3.5: The name `assertNotRegexpMatches` is a deprecated alias for `assertNotRegex()`.

assertCountEqual (*first, second, msg=None*)

Test that sequence *first* contains the same elements as *second*, 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 *first* and *second*. It verifies whether each element has the same count in both sequences. Equivalent to: `assertEqual(Counter(list(first)), Counter(list(second)))` but works with sequences of unhashable objects as well.

Nuevo en la versión 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.

Nuevo en la versión 3.1.

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
<code>assertMultiLineEqual(a, b)</code>	strings	3.1
<code>assertSequenceEqual(a, b)</code>	sequences	3.1
<code>assertListEqual(a, b)</code>	lists	3.1
<code>assertTupleEqual(a, b)</code>	tuples	3.1
<code>assertSetEqual(a, b)</code>	sets or frozensets	3.1
<code>assertDictEqual(a, b)</code>	dicts	3.1

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 strings with `assertEqual()`.

Nuevo en la versión 3.1.

assertSequenceEqual (*first, second, msg=None, seq_type=None*)

Tests that two sequences are equal. If a *seq_type* is supplied, both *first* and *second* 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()`.

Nuevo en la versión 3.1.

assertListEqual (*first, second, msg=None*)**assertTupleEqual** (*first, second, 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()`.

Nuevo en la versión 3.1.

assertSetEqual (*first, second, 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 *first* or *second* does not have a `set.difference()` method.

Nuevo en la versión 3.1.

assertDictEqual (*first, second, 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()`.

Nuevo en la versión 3.1.

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

This class attribute determines what happens when a custom failure message is passed as the *msg* argument to an `assertXXX` call that fails. `True` is the default value. In this case, the custom message is appended to the end of the standard failure message. When set to `False`, the custom message replaces the standard message.

The class setting can be overridden in individual test methods by assigning an instance attribute, `self.longMessage`, to `True` or `False` before calling the assert methods.

The class setting gets reset before each test call.

Nuevo en la versión 3.1.

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.

Nuevo en la versión 3.2.

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`.

Distinto en la versión 3.1: In 3.1 this was changed to add the test name to the short description even in the presence of a docstring. This caused compatibility issues with unittest extensions and adding the test name was moved to the `TextTestResult` in Python 3.2.

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.

Nuevo en la versión 3.1.

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.

Nuevo en la versión 3.1.

classmethod addClassCleanup(function, /, *args, **kwargs)

Add a function to be called after `tearDownClass()` to cleanup resources used during the test class. 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 `addClassCleanup()` when they are added.

If `setUpClass()` fails, meaning that `tearDownClass()` is not called, then any cleanup functions added will still be called.

Nuevo en la versión 3.8.

classmethod doClassCleanups()

This method is called unconditionally after `tearDownClass()`, or after `setUpClass()` if `setUpClass()` raises an exception.

It is responsible for calling all the cleanup functions added by `addClassCleanup()`. If you need cleanup functions to be called *prior* to `tearDownClass()` then you can call `doClassCleanups()` yourself.

`doClassCleanups()` pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

Nuevo en la versión 3.8.

class unittest.IsolatedAsyncioTestCase(methodName='runTest')

This class provides an API similar to `TestCase` and also accepts coroutines as test functions.

Nuevo en la versión 3.8.

coroutine asyncSetUp()

Method called to prepare the test fixture. This is called after `setUp()`. 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.

coroutine asyncTearDown()

Method called immediately after the test method has been called and the result recorded. This is called before `tearDown()`. 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 `asyncSetUp()` succeeds, regardless of the outcome of the test method. The default implementation does nothing.

addAsyncCleanup(function, /, *args, **kwargs)

This method accepts a coroutine that can be used as a cleanup function.

run (*result=None*)

Sets up a new event loop to run the test, collecting the result into the `TestResult` 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 returned to `run()`'s caller. At the end of the test all the tasks in the event loop are cancelled.

An example illustrating the order:

```
from unittest import IsolatedAsyncioTestCase

events = []

class Test(IsolatedAsyncioTestCase):

    def setUp(self):
        events.append("setUp")

    async def asyncSetUp(self):
        self._async_connection = await AsyncConnection()
        events.append("asyncSetUp")

    async def test_response(self):
        events.append("test_response")
        response = await self._async_connection.get("https://example.com")
        self.assertEqual(response.status_code, 200)
        self.addAsyncCleanup(self.on_cleanup)

    def tearDown(self):
        events.append("tearDown")

    async def asyncTearDown(self):
        await self._async_connection.close()
        events.append("asyncTearDown")

    async def on_cleanup(self):
        events.append("cleanup")

if __name__ == "__main__":
    unittest.main()
```

After running the test, `events` would contain `["setUp", "asyncSetUp", "test_response", "asyncTearDown", "tearDown", "cleanup"]`.

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	Deprecated alias
<i>assertEqual()</i>	failUnlessEqual	assertEquals
<i>assertNotEqual()</i>	failIfEqual	assertNotEquals
<i>assertTrue()</i>	failUnless	assert_
<i>assertFalse()</i>	failIf	
<i>assertRaises()</i>	failUnlessRaises	
<i>assertAlmostEqual()</i>	failUnlessAlmostEqual	assertAlmostEquals
<i>assertNotAlmostEqual()</i>	failIfAlmostEqual	assertNotAlmostEquals
<i>assertRegex()</i>		assertRegexMatches
<i>assertNotRegex()</i>		assertNotRegexMatches
<i>assertRaisesRegex()</i>		assertRaisesRegex

Obsoleto desde la versión 3.1: The fail* aliases listed in the second column have been deprecated.

Obsoleto desde la versión 3.2: The assert* aliases listed in the third column have been deprecated.

Obsoleto desde la versión 3.2: *assertRegexMatches* and *assertRaisesRegex* have been renamed to *assertRegex()* and *assertRaisesRegex()*.

Obsoleto desde la versión 3.5: The *assertNotRegexMatches* name is deprecated in favor of *assertNotRegex()*.

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 may be called several times on a single suite (for example when counting tests or comparing for equality) so the tests returned by repeated iterations before *TestSuite.run()* must be the same for each call iteration. After *TestSuite.run()*, callers should not rely on the tests returned by this method unless the caller uses a subclass that overrides *TestSuite._removeTestAtIndex()* to preserve test references.

Distinto en la versión 3.2: In earlier versions the *TestSuite* accessed tests directly rather than through iteration, so overriding *__iter__()* wasn't sufficient for providing tests.

Distinto en la versión 3.4: In earlier versions the *TestSuite* held references to each *TestCase* after *TestSuite.run()*. Subclasses can restore that behavior by overriding *TestSuite._removeTestAtIndex()*.

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 attributes:

errors

A list of the non-fatal errors encountered while loading tests. Not reset by the loader at any point. Fatal errors are signalled by the relevant a method raising an exception to the caller. Non-fatal errors are also indicated by a synthetic test that will raise the original error when run.

Nuevo en la versión 3.5.

TestLoader objects have the following methods:

loadTestsFromTestCase (testCaseClass)

Return a suite of all test cases contained in the *TestCase*-derived *testCaseClass*.

A test case instance is created for each method named by *getTestCaseNames()*. By default these are the method names beginning with *test*. If *getTestCaseNames()* returns no methods, but the *runTest()* method is implemented, a single test case is created for that method instead.

loadTestsFromModule (module, pattern=None)

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.

Nota: 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*. The *pattern* argument is passed as the third argument to *load_tests*.

Distinto en la versión 3.2: Support for *load_tests* added.

Distinto en la versión 3.5: The undocumented and unofficial *use_load_tests* default argument is deprecated and ignored, although it is still accepted for backward compatibility. The method also now accepts a keyword-only argument *pattern* which is passed to *load_tests* as the third argument.

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*.

Distinto en la versión 3.5: If an *ImportError* or *AttributeError* occurs while traversing *name* then a synthetic test that raises that error when run will be returned. These errors are included in the errors accumulated by `self.errors`.

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 the import failure is due to *SkipTest* being raised, it will be recorded as a skip instead of an error.

If a package (a directory containing a file named `__init__.py`) is found, the package will be checked for a `load_tests` function. If this exists then it will be called `package.load_tests(loader, tests, pattern)`. Test discovery takes care to ensure that a package is only checked for tests once during an invocation, even if the `load_tests` function itself calls `loader.discover`.

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.

Nuevo en la versión 3.2.

Distinto en la versión 3.4: Modules that raise *SkipTest* on import are recorded as skips, not errors.

Distinto en la versión 3.4: *start_dir* can be a *namespace packages*.

Distinto en la versión 3.4: Paths are sorted before being imported so that execution order is the same even if the underlying file system's ordering is not dependent on file name.

Distinto en la versión 3.5: Found packages are now checked for `load_tests` regardless of whether their path matches *pattern*, because it is impossible for a package name to match the default pattern.

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.

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.

testNamePatterns

List of Unix shell-style wildcard test name patterns that test methods have to match to be included in test suites (see `-v` option).

If this attribute is not `None` (the default), all test methods to be included in test suites must match one of the patterns in this list. Note that matches are always performed using *fnmatch.fnmatchcase()*, so unlike patterns passed to the `-v` option, simple substring patterns will have to be converted using `*` wildcards.

This affects all the *loadTestsFrom**() methods.

Nuevo en la versión 3.7.

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.

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.

skipped

A list containing 2-tuples of *TestCase* instances and strings holding the reason for skipping the test.

Nuevo en la versión 3.1.

expectedFailures

A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents an expected failure or error 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.

Nuevo en la versión 3.2.

failfast

If set to `true` `stop()` will be called on the first failure or error, halting the test run.

Nuevo en la versión 3.2.

tb_locals

If set to `true` then local variables will be shown in tracebacks.

Nuevo en la versión 3.5.

wasSuccessful()

Return `True` if all tests run so far have passed, otherwise returns `False`.

Distinto en la versión 3.4: Returns `False` if there were any `unexpectedSuccesses` from tests marked with the `expectedFailure()` decorator.

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.

Nuevo en la versión 3.1.

stopTestRun()

Called once after all tests are executed.

Nuevo en la versión 3.1.

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 or errors, 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.

addSubTest (*test*, *subtest*, *outcome*)

Called when a subtest finishes. *test* is the test case corresponding to the test method. *subtest* is a custom *TestCase* instance describing the subtest.

If *outcome* is *None*, the subtest succeeded. Otherwise, it failed with an exception where *outcome* is a tuple of the form returned by *sys.exc_info()*: (*type*, *value*, *traceback*).

The default implementation does nothing when the outcome is a success, and records subtest failures as normal failures.

Nuevo en la versión 3.4.

class `unittest.TextTestResult` (*stream*, *descriptions*, *verbosity*)

A concrete implementation of *TestResult* used by the *TextTestRunner*.

Nuevo en la versión 3.2: 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=None*, *descriptions=True*, *verbosity=1*, *failfast=False*,
buffer=False, *resultclass=None*, *warnings=None*, *,
tb_locals=False)

A basic test runner implementation that outputs results to a stream. If *stream* is *None*, the default, *sys.stderr* is used as the output stream. This class has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations. Such implementations should accept ***kwargs* as the interface to construct runners changes when features are added to *unittest*.

By default this runner shows *DeprecationWarning*, *PendingDeprecationWarning*, *ResourceWarning* and *ImportWarning* even if they are *ignored by default*. Deprecation warnings caused by *deprecated unittest methods* are also special-cased and, when the warning filters are 'default' or 'always', they will appear only once per-module, in order to avoid too many warning messages. This behavior can be overridden using Python's *-Wd* or *-Wa* options (see Warning control) and leaving *warnings* to *None*.

Distinto en la versión 3.2: Added the *warnings* argument.

Distinto en la versión 3.2: The default stream is set to *sys.stderr* at instantiation time rather than import time.

Distinto en la versión 3.5: Added the *tb_locals* parameter.

`_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
```

`run(test)`

This method is the main public interface to the `TextTestRunner`. This method takes a `TestSuite` or `TestCase` instance. A `TestResult` is created by calling `_makeResult()` and the test(s) are run and the results printed to stdout.

```
unittest.main(module='__main__', defaultTest=None, argv=None, testRunner=None, testLoader=unittest.defaultTestLoader, exit=True, verbosity=1, failfast=None, catchbreak=None, buffer=None, warnings=None)
```

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 either the name of a single test or an iterable of test names 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*.

The `warnings` argument specifies the *warning filter* that should be used while running the tests. If it's not specified, it will remain `None` if a `-W` option is passed to **python** (see Warning control), otherwise it will be set to `'default'`.

Calling `main` actually returns an instance of the `TestProgram` class. This stores the result of the tests run as the `result` attribute.

Distinto en la versión 3.1: The `exit` parameter was added.

Distinto en la versión 3.2: The `verbosity`, `failfast`, `catchbreak`, `buffer` and `warnings` parameters were added.

Distinto en la versión 3.4: The `defaultTest` parameter was changed to also accept an iterable of test names.

load_tests Protocol

Nuevo en la versión 3.2.

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, pattern)
```

where *pattern* is passed straight through from `loadTestsFromModule`. It defaults to `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 in a directory containing a package, either from the command line or by calling `TestLoader.discover()`, then the package `__init__.py` will be checked for `load_tests`. If that function does not exist, discovery will recurse into the package as though it were just another directory. Otherwise, discovery of the package's tests will be left up to `load_tests` which 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
```

Distinto en la versión 3.5: Discovery no longer checks package names for matching *pattern* due to the impossibility of package names matching the default pattern.

26.4.9 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 `_ErrorHandler` 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.

To add cleanup code that must be run even in the case of an exception, use `addModuleCleanup`:

`unittest.addModuleCleanup` (*function*, /, **args*, ***kwargs*)

Add a function to be called after `tearDownModule()` to cleanup resources used during the test class. 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 `addModuleCleanup()` when they are added.

If `setUpModule()` fails, meaning that `tearDownModule()` is not called, then any cleanup functions added will still be called.

Nuevo en la versión 3.8.

`unittest.doModuleCleanups` ()

This function is called unconditionally after `tearDownModule()`, or after `setUpModule()` if `setUpModule()` raises an exception.

It is responsible for calling all the cleanup functions added by `addCleanupModule()`. If you need cleanup functions to be called *prior* to `tearDownModule()` then you can call `doModuleCleanups()` yourself.

`doModuleCleanups()` pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

Nuevo en la versión 3.8.

26.4.10 Signal Handling

Nuevo en la versión 3.2.

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.

`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.

`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.

`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):
    ...
```

26.5 unittest.mock — Biblioteca de objetos simulados

Nuevo en la versión 3.3.

Source code: [Lib/unittest/mock.py](#)

`unittest.mock` es una biblioteca para pruebas de software en Python. Te permite reemplazar partes del sistema bajo prueba con objetos simulados y hacer aserciones sobre cómo se han utilizado.

El módulo `unittest.mock` proporciona una clase principal `Mock` eliminando la necesidad de crear una gran cantidad de stubs en todo el conjunto de pruebas. Después de realizar una determinada acción, puedes hacer aserciones sobre qué métodos/atributos se usaron y los argumentos con los que se llamaron. También puedes especificar valores de retorno y establecer los atributos necesarios de la forma habitual.

Además, `mock` proporciona un decorador `patch()` que puede manejar el parcheo de atributos a nivel de clase y de módulo dentro del ámbito de una prueba, junto con `sentinel` para crear objetos únicos. Consulta [quick guide](#) para ver algunos ejemplos de cómo utilizar `Mock`, `MagicMock` y `patch()`.

`Mock` es muy fácil de usar y está diseñado para ser utilizado junto a `unittest`. `Mock` se basa en el patrón “acción -> aserción” en lugar de usar el patrón “grabación -> reproducción” utilizado por muchos frameworks de simulación.

Hay un backport del módulo `unittest.mock` para versiones anteriores de Python [disponible en PyPI](#).

26.5.1 Guía rápida

Los objetos de las clases `Mock` y `MagicMock` van creando todos los atributos y métodos a medida que se accede a ellos y almacenan detalles de cómo se han utilizado. Puedes configurarlos para especificar valores de retorno o limitar qué atributos están disponibles y posteriormente hacer aserciones sobre cómo han sido utilizados:

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

`side_effect` te permite implementar efectos colaterales, lo que incluye lanzar una excepción cuando se llama a un objeto simulado:

```
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

Existen muchas otras formas de configurar y controlar el comportamiento de `Mock`. Por ejemplo, el argumento `spec` configura el objeto simulado para que tome su especificación de otro objeto. Cualquier intento de acceder a atributos o métodos en el objeto simulado que no existan en la especificación fallará lanzando una excepción `AttributeError`.

El decorador / gestor de contexto `patch()` facilita la simulación de clases u objetos en un módulo bajo prueba. El objeto que especifiques será reemplazado por un objeto simulado (u otro objeto) durante la prueba y será restaurado cuando esta finalice:

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

Nota: Cuando anidas decoradores `patch`, los objetos simulados se pasan a la función decorada en el mismo orden en el que fueron aplicados (el orden normal en el que se aplican los decoradores en *Python*). Esto significa de abajo hacia arriba, por lo que en el ejemplo anterior se pasa primero el objeto simulado para `module.ClassName1`.

Al usar `patch()` es importante que parchees los objetos en el espacio de nombres donde son buscados. Esto normalmente es sencillo, pero para una guía rápida, lee *dónde parchear*.

Además de decorador, la función `patch()` se puede usar como gestor de contexto en una declaración `with`:

```
>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_with(1, 2, 3)
```

También existe la función `patch.dict()` que permite establecer valores en un diccionario dentro de un ámbito y restaurar el diccionario a su estado original cuando finaliza la prueba:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

Mock admite la simulación de los *métodos mágicos* de Python. La forma más sencilla de utilizar métodos mágicos es mediante la clase `MagicMock`. Te permite hacer cosas como:

```
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

Mock también permite asignar funciones (u otras instancias de Mock) a métodos mágicos y se asegura de que serán llamadas de forma apropiada. La clase `MagicMock` es solo una variante de Mock con la diferencia de que tiene todos los métodos mágicos previamente creados para ti (bueno, todos los que son útiles).

El siguiente es un ejemplo de uso de métodos mágicos utilizando la clase Mock ordinaria:

```
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='whewheeee')
>>> str(mock)
'whewheeee'
```

Para asegurarte de que los objetos simulados en tus pruebas tienen exactamente la misma API que los objetos que están reemplazando, puedes usar *autoespecificación*. La autoespecificación se puede realizar a través del argumento *autospec* de *patch*, o mediante la función *create_autospec()*. La autoespecificación crea objetos simulados que tienen los mismos atributos y métodos que los objetos que están reemplazando, y todas las función y métodos (incluidos los constructores) tienen la misma firma de llamada que los objetos reales.

Esto asegura que tus simulaciones fallarán, si se utilizan incorrectamente, de la misma manera que lo haría tu código en producción:

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
>>> mock_function = create_autospec(function, return_value='fishy')
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
Traceback (most recent call last):
...
TypeError: <lambda>() takes exactly 3 arguments (1 given)
```

create_autospec() también se puede usar en clases, donde copia la firma del método `__init__`, y en objetos invocables, donde copia la firma del método `__call__`.

26.5.2 La clase Mock

Mock es un objeto simulado flexible, destinado a reemplazar el uso de stubs y dobles de prueba en todo tu código. Los objetos *Mock* son invocables y crean atributos en el mismo momento que se accede a ellos como nuevos objetos *Mock*¹. Acceder al mismo atributo siempre retornará el mismo objeto *Mock*. Además, registran cómo los usas, lo que te permite hacer aserciones sobre cómo tu código ha interactuado con ellos.

MagicMock es una subclase de *Mock* con todos los métodos mágicos creados previamente y listos para ser usados. También hay variantes no invocables, útiles cuando se están simulando objetos que no se pueden llamar: *NonCallableMock* y *NonCallableMagicMock*

Los decoradores *patch()* facilitan la sustitución temporal de clases en un módulo en particular con un objeto *Mock*. Por defecto, *patch()* creará un objeto *MagicMock* automáticamente. Se puede especificar una clase alternativa a *Mock* usando el argumento *new_callable* de *patch()*.

class `unittest.mock.Mock` (*spec=None, side_effect=None, return_value=DEFAULT, wraps=None, name=None, spec_set=None, unsafe=False, **kwargs*)

Creará un nuevo objeto *Mock*. *Mock* toma varios argumentos opcionales que especifican el comportamiento del objeto *Mock*:

- *spec*: Puede ser una lista de cadenas de caracteres o un objeto existente previamente (una clase o una instancia) que actúa como la especificación del objeto simulado. Si pasas un objeto, se forma una lista de cadenas llamando a la función *dir* en el objeto (excluyendo los métodos y atributos mágicos no admitidos). Acceder a cualquier atributo que no esté en esta lista generará una excepción *AttributeError*.

Si *spec* es un objeto (en lugar de una lista de cadenas de caracteres), `__class__` retorna la clase del objeto especificado. Esto permite que los objetos simulados pasen las pruebas de *isinstance()*.

- *spec_set*: Una variante más estricta de *spec*. Si se utiliza, cualquier intento de *establecer* u obtener un atributo del objeto simulado que no esté en el objeto pasado como *spec_set* lanzará una excepción *AttributeError*.
- *side_effect*: Una función que se llamará cada vez que el objeto simulado sea invocado. Consultar el atributo *side_effect* para más información. Es útil para lanzar excepciones o para cambiar dinámicamente

¹ Las únicas excepciones son los métodos y atributos mágicos (aquellos que tienen doble subrayado al principio y al final). *Mock* no los crea automáticamente, sino que lanza una excepción *AttributeError*. Esto se debe a que el intérprete a menudo solicitará implícitamente estos métodos y terminaría muy confundido si obtiene un nuevo objeto *Mock* cuando espera un método mágico. Si necesitas soporte para métodos mágicos, consulta *métodos mágicos*.

valores de retorno. La función se llama con los mismos argumentos que el objeto simulado, y a menos que retorne `DEFAULT`, su valor de retorno se utiliza como valor de retorno del propio objeto simulado.

Alternativamente `side_effect` puede ser una clase o instancia de excepción. En este caso, se lanza la excepción cuando se llama al objeto simulado.

Si `side_effect` es un iterable, cada llamada al objeto simulado retornará el siguiente valor del iterable.

Un `side_effect` se puede desactivar estableciéndolo en `None`.

- `return_value`: El valor retornado cuando se llama al objeto simulado. Por defecto, este es una nueva instancia de la clase `Mock` (creada en el primer acceso). Consultar el atributo `return_value` para más detalles.
- `unsafe`: Por defecto, si cualquier atributo comienza con `assert` o `assret` y se intenta acceder a él, se lanza una excepción `AttributeError`. Pasar `unsafe=True` permitirá el acceso a estos atributos.

Nuevo en la versión 3.5.

- `wraps`: objeto a envolver (simular) por la instancia de `Mock`. Si `wraps` no es `None`, al llamar al objeto `Mock` se pasa la llamada a través del objeto envuelto (retornando el resultado real). Acceder a un atributo del objeto simulado retornará otro objeto `Mock` que envuelve al atributo correspondiente del objeto real envuelto (de modo que intentar acceder a un atributo que no existe lanzará una excepción `AttributeError`).

Si el objeto simulado tiene un `return_value` explícito establecido, las llamadas no se pasan al objeto envuelto y `return_value` se retorna en su lugar.

- `name`: Si el objeto simulado tiene un nombre, será utilizado en la representación imprimible del mismo. Esto puede ser útil para la depuración. El nombre se propaga a los objetos simulados hijos.

Los objetos simulados también pueden ser invocados con argumentos por palabra clave arbitrarios. Estos serán utilizados para establecer atributos en el objeto simulado una vez creado. Consultar el método `configure_mock()` para más detalles.

`assert_called()`

Aserta si el objeto simulado se ha invocado al menos una vez.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

Nuevo en la versión 3.6.

`assert_called_once()`

Aserta si el objeto simulado se ha invocado exactamente una vez.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called once. Called 2 times.
```

Nuevo en la versión 3.6.

`assert_called_with(*args, **kwargs)`

Este método es una manera apropiada de asertar si la última llamada se ha realizado de una manera particular:

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

assert_called_once_with(*args, **kwargs)

Aserta si el objeto simulado se ha invocado exactamente una vez y si esa llamada se realizó con los argumentos especificados.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

assert_any_call(*args, **kwargs)

Aserta si el objeto simulado se ha invocado con los argumentos especificados.

La aserción pasa si el objeto simulado se ha invocado *en algún momento*, a diferencia de `assert_called_with()` y `assert_called_once_with()`, con los que sólo pasa la aserción si la llamada es la más reciente, y en el caso de `assert_called_once_with()` también debe ser la única llamada realizada.

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls(calls, any_order=False)

Aserta si el objeto simulado se ha invocado con las llamadas especificadas. La lista `mock_calls` se compara con la lista de llamadas.

Si `any_order` es falso entonces las llamadas deben ser secuenciales. No puede haber llamadas adicionales antes o después de las llamadas especificadas.

Si `any_order` es verdadero, las llamadas pueden estar en cualquier orden, pero deben aparecer todas en `mock_calls`.

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

assert_not_called()

Aserta si el objeto simulado nunca fue invocado.

```
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
...
AssertionError: Expected 'hello' to not have been called. Called 1 times.
```

Nuevo en la versión 3.5.

reset_mock (*, *return_value=False*, *side_effect=False*)

El método *reset_mock* restablece todos los atributos de llamada en un objeto simulado:

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

Distinto en la versión 3.6: Se añadieron dos argumentos por palabra clave a la función *reset_mock*.

Esto puede ser útil cuando se quiere hacer una serie de aserciones que reutilizan el mismo objeto. Ten en cuenta que por defecto *reset_mock()* no borra el valor de retorno, *side_effect*, ni cualquier atributo hijo que hayas establecido mediante asignación normal. En caso de que quieras restablecer *return_value* o *side_effect*, debes pasar el parámetro correspondiente como `True`. Los objetos simulados hijos y el objeto simulado que conforma el valor de retorno (si los hay) se restablecen también.

Nota: *return_value* y *side_effect* son argumentos por palabra clave exclusivamente.

mock_add_spec (*spec*, *spec_set=False*)

Agrega una especificación a un objeto simulado. *spec* puede ser un objeto o una lista de cadenas de caracteres. Sólo los atributos presentes en *spec* pueden ser obtenidos desde el objeto simulado.

Si *spec_set* es verdadero, solo los atributos de la especificación pueden ser establecidos.

attach_mock (*mock*, *attribute*)

Adjunta otro objeto simulado como un atributo de la instancia actual, substituyendo su nombre y su padre. Las llamadas al objeto simulado adjuntado se registrarán en los atributos *method_calls* y *mock_calls* del padre.

configure_mock (***kwargs*)

Establece los atributos del objeto simulado por medio de argumentos por palabra clave.

Los atributos, los valores de retorno y los efectos de colaterales se pueden configurar en los objetos simulados hijos usando la notación de punto estándar y desempaquetando un diccionario en la llamada al método:

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
...
KeyError
```

Lo mismo se puede lograr en la llamada al constructor de los objetos simulados:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
...
KeyError
```

`configure_mock()` existe con el fin de facilitar la configuración después de que el objeto simulado haya sido creado.

`__dir__()`

Los objetos `Mock` limitan los resultados de `dir(some_mock)` a resultados útiles. Para los objetos simulados con una especificación (*spec*), esto incluye todos los atributos permitidos para el mismo.

Consultar `FILTER_DIR` para conocer que hace este filtrado y la forma de desactivarlo.

`_get_child_mock(**kw)`

Crea los objetos simulados hijos para los atributos y el valor de retorno. Por defecto los objetos simulados hijos serán del mismo tipo que el padre. Las subclases de `Mock` pueden redefinir este método para personalizar la forma en la que se construye el objeto simulado hijo.

Para objetos simulados no invocables la variante invocable será utilizada (en lugar de cualquier subclase personalizada).

`called`

Un booleano que representa si el objeto simulado ha sido invocado o no:

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

`call_count`

Un entero que le indica cuántas veces el objeto simulado ha sido invocado:

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

`return_value`

Establece este atributo para configurar el valor a retornar cuando se llama al objeto simulado:

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

El valor de retorno por defecto es otro objeto simulado y se puede configurar de forma habitual:

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()' id='...'>
>>> mock.return_value.assert_called_with()
```

`return_value` también se puede establecer directamente en el constructor:

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

`side_effect`

Este atributo puede ser una función a ser llamada cuando se llame al objeto simulado, un iterable o una

excepción (clase o instancia) para ser lanzada.

Si pasas una función, será llamada con los mismos argumentos que el objeto simulado y, a menos que la función retorne el singleton `DEFAULT`, la llamada al objeto simulado retornará lo mismo que retorna la función. En cambio, si la función retorna `DEFAULT`, entonces el objeto simulado retornará su valor normal (el del atributo `return_value`).

Si pasas un iterable, se utiliza para obtener un iterador a partir del mismo que debe producir un valor en cada llamada. Este valor puede ser una instancia de la excepción a ser lanzada o un valor a retornar al llamar al objeto simulado (el manejo de `DEFAULT` es igual que en el caso en el que se pasa una función).

Un ejemplo de un objeto simulado que genera una excepción (para probar el manejo de excepciones de una API):

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
...
Exception: Boom!
```

Usando `side_effect` para retornar una secuencia de valores:

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

Usando un objeto invocable:

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

`side_effect` se puede establecer en el constructor. Aquí hay un ejemplo que suma uno al valor del objeto simulado invocado y lo retorna:

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
```

Establecer `side_effect` en `None` lo desactiva:

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
Traceback (most recent call last):
...
KeyError
>>> m.side_effect = None
>>> m()
3
```

`call_args`

Este atributo es `None` (si el objeto simulado no ha sido invocado) o los argumentos con los que se llamó por última vez. En este último caso, será una tupla con dos elementos: el primer miembro, que también es accesible a través de la propiedad `args`, son los argumentos posicionales con los que el objeto simulado

se llamó (o una tupla vacía si no se pasó ninguno) y el segundo miembro, que también es accesible mediante la propiedad `kwargs`, son los argumento por palabra clave pasados (o un diccionario vacío si no se pasó ninguno).

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()
>>> mock.call_args
call()
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock.call_args.args
(3, 4)
>>> mock.call_args.kwargs
{}
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args.args
(3, 4, 5)
>>> mock.call_args.kwargs
{'key': 'fish', 'next': 'w00t!'}
```

El atributo `call_args`, junto con los miembros de las listas `call_args_list`, `method_calls` y `mock_calls` son objetos `call`. Estos objetos son tuplas, con la finalidad de que puedan ser desempaquetadas para acceder a los argumentos individuales y hacer aserciones más complejas. Consultar [objetos call como tuplas](#) para más información.

Distinto en la versión 3.8: Added args and kwargs properties.

`call_args_list`

Este argumento es una lista de todas las llamadas consecutivas realizadas al objeto simulado (por lo que la longitud de la lista es el número de veces que se ha invocado). Previamente a que se hayan realizado llamadas es una lista vacía. El objeto `call` se puede utilizar para construir convenientemente las listas de llamadas a comparar con `call_args_list`.

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next': 'w00t!'},)]
>>> mock.call_args_list == expected
True
```

Los miembros de `call_args_list` son objetos `call`. Estos pueden ser desempaquetados como tuplas para acceder a los argumentos individuales. Consultar [objetos call como tuplas](#) para más información.

`method_calls`

Igual que realizan un seguimiento de las llamadas hechas a sí mismos, los objetos simulados también realizan un seguimiento a *sus* métodos y atributos, así como de las llamadas hechas a los mismos:

```
>>> mock = Mock()
>>> mock.method()
```

(continué en la próxima página)

(proviene de la página anterior)

```
<Mock name='mock.method()' id='... '>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='... '>
>>> mock.method_calls
[call.method(), call.property.method.attribute()]
```

Los miembros de `method_calls` son objetos `call`. Estos pueden ser desempaquetados como tuplas para acceder a los atributos individuales. Consultar *objetos call como tuplas* para más información.

mock_calls

`mock_calls` registra *todas* las llamadas al objeto simulado, sus métodos, métodos mágicos y objetos simulados del valor de retorno.

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='... '>
>>> mock.second()
<MagicMock name='mock.second()' id='... '>
>>> int(mock)
1
>>> result(1)
<MagicMock name='mock()' id='... '>
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

Los miembros de `mock_calls` son objetos `call`. Estos pueden ser desempaquetados como tuplas para acceder a los argumentos individuales. Consultar *objetos call como tuplas* para más información.

Nota: La forma como se registra el atributo `mock_calls` implica que cuando se realizan llamadas anidadas, los parámetros de las llamadas previas no se registran, por lo que siempre resultan iguales al comparar:

```
>>> mock = MagicMock()
>>> mock.top(a=3).bottom()
<MagicMock name='mock.top().bottom()' id='... '>
>>> mock.mock_calls
[call.top(a=3), call.top().bottom()]
>>> mock.mock_calls[-1] == call.top(a=-1).bottom()
True
```

__class__

Normalmente, el atributo de un objeto `__class__` retornará su tipo. Para un objeto simulado con un spec, `__class__` retorna la clase especificada en su lugar. Esto permite a los objetos simulados pasar los test de `isinstance()` para el objeto que están reemplazando / enmascarando:

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

El atributo `__class__` es asignable, esto permite al objeto simulado pasar una verificación de `isinstance()` sin verse forzado a utilizar una especificación:

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class `unittest.mock.NonCallableMock` (*spec=None, wraps=None, name=None, spec_set=None, **kwargs*)

Una versión no invocable de `Mock`. Los parámetros del constructor tienen el mismo significado que en `Mock`, con la excepción de `return_value` y `side_effect` que no tienen sentido en un objeto simulado no invocable.

Los objetos simulados que usan una clase o una instancia como `spec` o `spec_set` son capaces de pasar los test de `isinstance()`:

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

Las clases `Mock` tienen soporte para simular los métodos mágicos. Consultar la sección dedicada a los *métodos mágicos* para ver los detalles.

Las clases simuladas y los decoradores `patch()` aceptan todas argumentos por palabra clave arbitrarios para la configuración. En los decoradores `patch()` los argumentos por palabra clave se pasan al constructor del objeto simulado creado. Estos argumentos se usan para configurar los atributos del propio objeto simulado:

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

Tanto el valor de retorno como el efecto colateral del objeto simulado pueden ser establecidos de la misma manera, mediante notación de punto. En cambio, si se quieren establecer mediante el constructor, dado que no se puede utilizar notación de punto directamente en una llamada, se tiene que crear un diccionario y desempaquetarlos usando `**`:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
...
KeyError
```

Un objeto simulado invocable que fue creado con un `spec` (o un `spec_set`) introspeccionará la firma del objeto de la especificación en el momento de emparejar las llamadas al objeto simulado. Esto le permite hacer coincidir sus argumentos con los argumentos de la llamada real, independientemente de si se pasaron por posición o por nombre:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

Esto se aplica a `assert_called_with()`, `assert_called_once_with()`, `assert_has_calls()` y `assert_any_call()`. Cuando se hace uso de *Autoespecificación*, también se aplicará a las llamadas a los métodos del objeto simulado.

Distinto en la versión 3.4: Se añadió introspección de firma en objetos simulados especificados y auto-especificados.

class `unittest.mock.PropertyMock` (*args, **kwargs)

Un objeto simulado destinado a ser utilizado en una clase como una propiedad, u otro descriptor. La clase `PropertyMock` proporciona los métodos `__get__()` y `__set__()`, por lo que puedes especificar un valor de retorno para cuando su valor es requerido.

La obtención de una instancia `PropertyMock` desde un objeto ocasiona una llamada al objeto simulado, sin argumentos. Establecer su valor también llama al objeto simulado, con el valor a establecer como argumento.

```
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

Debido a la forma en que se almacenan los atributos simulados, no es posible conectar directamente un `PropertyMock` a un objeto simulado. En su lugar se puede conectar al tipo (type) del objeto simulado:

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```

class `unittest.mock.AsyncMock` (spec=None, side_effect=None, return_value=DEFAULT, wraps=None, name=None, spec_set=None, unsafe=False, **kwargs)

Una versión asíncrona de `Mock`. El objeto `AsyncMock` se comportará de tal modo que el objeto es reconocido como una función asíncrona y el resultado de su llamada es un objeto aguardable (awaitable).

```
>>> mock = AsyncMock()
>>> asyncio.iscoroutinefunction(mock)
True
>>> inspect.isawaitable(mock())
True
```

El resultado de `mock()` es una función asíncrona que proporcionará el resultado de `side_effect` o de `return_value` después de haber sido aguardada:

- si `side_effect` es una función, la función asíncrona retornará el resultado de esa función,
- si `side_effect` es una excepción, la función asíncrona lanzará la excepción,
- si `side_effect` es un iterable, la función asíncrona retornará el siguiente valor del iterable, sin embargo, si se agota la secuencia de resultados, se lanza una excepción `StopAsyncIteration` inmediatamente,
- si `side_effect` no está definido, la función asíncrona retornará el valor definido por `return_value`, por lo tanto, la función asíncrona retorna un nuevo objeto `AsyncMock` por defecto.

Establecer el argumento `spec` de un objeto `Mock` o `MagicMock` en una función asíncrona resultará en que un objeto corrutina será retornado después de la llamada al objeto.

```
>>> async def async_func(): pass
...
>>> mock = MagicMock(async_func)
>>> mock
<MagicMock spec='function' id='...'>
>>> mock()
<coroutine object AsyncMockMixin._mock_call at ...>
```

Establecer el argumento *spec* de un objeto *Mock*, *MagicMock* o *AsyncMock* en una clase que tiene simultáneamente funciones asíncronas y síncronas hará que se detecten automáticamente las funciones síncronas y las establecerá como *MagicMock* (si el objeto simulado padre es *AsyncMock* o *MagicMock*) o *Mock* (si el objeto simulado padre es *Mock*). Todas las funciones asíncronas serán *AsyncMock*.

```
>>> class ExampleClass:
...     def sync_foo():
...         pass
...     async def async_foo():
...         pass
...
>>> a_mock = AsyncMock(ExampleClass)
>>> a_mock.sync_foo
<MagicMock name='mock.sync_foo' id='...'>
>>> a_mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
>>> mock = Mock(ExampleClass)
>>> mock.sync_foo
<Mock name='mock.sync_foo' id='...'>
>>> mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
```

Nuevo en la versión 3.8.

assert_awaited()

Aserta si el objeto simulado fue aguardado al menos una vez. Ten en cuenta que, independientemente del objeto que ha sido invocado, la palabra clave *await* debe ser utilizada:

```
>>> mock = AsyncMock()
>>> async def main(coroutine_mock):
...     await coroutine_mock
...
>>> coroutine_mock = mock()
>>> mock.called
True
>>> mock.assert_awaited()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited.
>>> asyncio.run(main(coroutine_mock))
>>> mock.assert_awaited()
```

assert_awaited_once()

Aserta si el objeto simulado fue aguardado exactamente una vez.

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
>>> asyncio.run(main())
>>> mock.method.assert_awaited_once()
Traceback (most recent call last):
```

(continué en la próxima página)

(proviene de la página anterior)

```
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_awaited_with(*args, **kwargs)

Aserta si el último aguardo (await) fue con los argumentos especificados.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_with('foo', bar='bar')
>>> mock.assert_awaited_with('other')
Traceback (most recent call last):
...
AssertionError: expected call not found.
Expected: mock('other')
Actual: mock('foo', bar='bar')
```

assert_awaited_once_with(*args, **kwargs)

Aserta si que el objeto simulado se ha aguardado exactamente una vez y con los argumentos especificados.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_any_await(*args, **kwargs)

Aserta si el objeto simulado nunca se ha aguardado con los argumentos especificados.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> asyncio.run(main('hello'))
>>> mock.assert_any_await('foo', bar='bar')
>>> mock.assert_any_await('other')
Traceback (most recent call last):
...
AssertionError: mock('other') await not found
```

assert_has_awaits(calls, any_order=False)

Aserta si el objeto simulado ha sido aguardado con las llamadas especificadas. Para comprobar los aguardos (awaits) se utiliza la lista `await_args_list`.

Si `any_order` es falso, los aguardos (awaits) deben ser secuenciales. No puede haber llamadas adicionales antes o después de los aguardos especificados.

Si `any_order` es verdadero, entonces los aguardos (awaits) pueden estar en cualquier orden, pero deben aparecer todos en `await_args_list`.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
```

(continué en la próxima página)

(proviene de la página anterior)

```

...     await mock(*args, **kwargs)
...
>>> calls = [call("foo"), call("bar")]
>>> mock.assert_has_awaits(calls)
Traceback (most recent call last):
...
AssertionError: Awaits not found.
Expected: [call('foo'), call('bar')]
Actual: []
>>> asyncio.run(main('foo'))
>>> asyncio.run(main('bar'))
>>> mock.assert_has_awaits(calls)

```

assert_not_awaited()

Aserta si el objeto simulado nunca ha sido aguardado.

```

>>> mock = AsyncMock()
>>> mock.assert_not_awaited()

```

reset_mock(*args, **kwargs)

Consultar `Mock.reset_mock()`. Además, también establece `await_count` a 0, `await_args` a None y borra `await_args_list`.

await_count

Un registro entero de cuántas veces se ha aguardado el objeto simulado.

```

>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.await_count
1
>>> asyncio.run(main())
>>> mock.await_count
2

```

await_args

Este atributo es None (si el objeto simulado no se ha aguardado) o los argumentos con los que fue aguardado la última vez. Su funcionamiento es idéntico al de `Mock.call_args`.

```

>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args
>>> asyncio.run(main('foo'))
>>> mock.await_args
call('foo')
>>> asyncio.run(main('bar'))
>>> mock.await_args
call('bar')

```

await_args_list

Es una lista de todas los aguardos (awaits) realizados en el objeto simulado de forma secuencial (por lo que la longitud de la lista es el número de veces que se ha aguardado el objeto). Si no se han realizado aguardos previos, es una lista vacía.

```

>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)

```

(continué en la próxima página)

(proviene de la página anterior)

```

...
>>> mock.await_args_list
[]
>>> asyncio.run(main('foo'))
>>> mock.await_args_list
[call('foo')]
>>> asyncio.run(main('bar'))
>>> mock.await_args_list
[call('foo'), call('bar')]

```

Llamar a los objetos simulados

Los objetos Mock son invocables. La llamada a uno retornará el valor establecido en el atributo `return_value`. El valor de retorno por defecto es un nuevo objeto Mock, el cual se crea la primera vez que se accede al valor de retorno (ya sea explícitamente o llamando al objeto Mock). Una vez creado, se almacena y el mismo objeto es retornado cada vez que se solicita.

Las llamadas realizadas al objeto serán registradas en los atributos `call_args` y `call_args_list`.

Si `side_effect` se establece, será invocado después de que la llamada haya sido registrada, por lo que la llamada se registra aunque `side_effect` lance una excepción.

La forma más sencilla de hacer que un objeto simulado lance de una excepción cuando sea invocado es establecer su atributo `side_effect` como una clase o instancia de excepción:

```

>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]

```

Si `side_effect` es una función, la llamada al objeto simulado retornará lo que sea que esta función retorne. La función establecida en `side_effect` se llama con los mismos argumentos con los que el objeto simulado ha sido invocado. Esto te permite variar el valor de retorno de la llamada dinámicamente, en función de la entrada:

```

>>> def side_effect(value):
...     return value + 1
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]

```

Si se desea que el objeto simulado aún retorne el valor por defecto (un nuevo objeto simulado), o cualquier valor de retorno establecido, entonces existen dos maneras de proceder. Se puede retornar tanto el atributo `mock.return_value` como `DEFAULT` desde `side_effect`:

```
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...     return m.return_value
...
>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> m.side_effect = side_effect
>>> m()
3
```

Para eliminar un `side_effect`, volviendo al comportamiento predeterminado, establece el atributo `side_effect` en `None`:

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

El atributo `side_effect` también puede ser cualquier objeto iterable. En este caso, las llamadas repetidas al objeto simulado irán retornando valores del iterable (hasta que el iterable se agote, momento en el que se lanza una excepción *`StopIteration`*):

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
...
StopIteration
```

Si cualquier miembro del iterable es una excepción, se lanzará en lugar de retornarse:

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
...
ValueError
>>> m()
66
```

Eliminar atributos

Los objetos simulados crean atributos en demanda. Esto les permite hacerse pasar por objetos de cualquier tipo.

Es posible que desees que un objeto simulado retorne `False` al llamar a `hasattr()`, o que lance una excepción `AttributeError` cuando se intenta obtener un atributo. Puedes hacer todo esto proporcionando un objeto adecuado al atributo `spec` del objeto simulado, pero no siempre es conveniente.

Puedes «bloquear» atributos eliminándolos. Una vez eliminado, el acceso a un atributo lanzará una excepción `AttributeError`.

```
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
...
AttributeError: f
```

Los nombres de los objetos simulados y el atributo `name`

Dado que «name» es un argumento para el constructor de la clase `Mock`, si quieres que tu objeto simulado tenga un atributo «name», no puedes simplemente pasarlo al constructor en tiempo de creación. Hay dos alternativas. Una opción es usar el método `configure_mock()`:

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

Una opción más sencilla es simplemente establecer el atributo «name» después de la creación del objeto simulado:

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

Adjuntar objetos simulados como atributos

Cuando se adjunta un objeto simulado como un atributo de otro objeto simulado (o como su valor de retorno) se convierte en un «hijo» del mismo. Las llamadas a los hijos se registran en los atributos `method_calls` y `mock_calls` del padre. Esto es útil para configurar objetos simulados hijos para después adjuntarlos al padre, o para adjuntar objetos simulados a un padre que se encargará de registrar todas las llamadas a los hijos, permitiéndote hacer aserciones sobre el orden de las llamadas entre objetos simulados:

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

La excepción a lo anterior es si el objeto simulado tiene un nombre. Esto te permite evitar el comportamiento de «parentesco» explicado previamente, si por alguna razón no deseas que suceda.

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

Los objetos simulados creados automáticamente por la función `patch()` también reciben nombres automáticamente. Para adjuntar un objeto simulado con nombre a un padre debes utilizar el método `attach_mock()`:

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

26.5.3 Parcheadores

Los decoradores `patch` se utilizan para parchear los objetos sólo dentro del ámbito de la función que decoran. Se encargan automáticamente de despachar una vez terminada la prueba, incluso si se lanzan excepciones. Todas estas funciones también se pueden utilizar con declaraciones o como decoradores de clase.

`patch`

Nota: La función `patch()` es sencilla de usar. La clave es realizar la aplicación de parches en el espacio de nombres correcto. Consulta la sección [where to patch](#) para más detalles.

`unittest.mock.patch(target, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs)`

`patch()` actúa como decorador de función, decorador de clase o como gestor de contexto. Ya sea en el interior del cuerpo de una función o dentro de una declaración `with`, `target` es parcheado con un objeto `new`. Cuando la función / declaración `with` termina su ejecución el parche se deshace automáticamente.

Si se omite `new`, entonces el objetivo se reemplaza por un objeto `AsyncMock` si el objeto parcheado es una función asíncrona, o con un objeto `MagicMock` en caso contrario. Si se utiliza `patch()` como decorador y se omite `new`, el objeto simulado creado se pasa como un argumento adicional a la función decorada. Si se utiliza `patch()` como un gestor de contexto, el objeto simulado creado es retornado por el gestor de contexto.

`target` debe ser una cadena de la forma `'paquete.modulo.NombreDeLaClase'`. `target` es importado y el objeto especificado reemplazado por el objeto `new`, por lo que `target` debe ser importable desde el entorno desde el cual estás llamando a `patch()`. Hay que tener presente que `target` es importado cuando se ejecuta la función decorada, no en tiempo de decoración.

Los argumentos `spec` y `spec_set` se pasan a `MagicMock` si `patch` está creando automáticamente uno para ti.

Además, puedes pasar `spec=True` o `spec_set=True`, lo que causa que `patch` pase el objeto que está siendo simulado como el objeto `spec/spec_set`.

`new_callable` te permite especificar una clase diferente, o un objeto invocable, que será invocada para crear el objeto `new`. Por defecto se utiliza `AsyncMock` para las funciones asíncronas y `MagicMock` para el resto.

Una variante más poderosa de *spec* es *autospec*. Si estableces `autospec=True` el objeto simulado se creará con una especificación del objeto que está siendo reemplazado. Todos los atributos del objeto simulado también tendrán la especificación del atributo correspondiente del objeto que está siendo reemplazado. Los argumentos de los métodos y funciones simulados son comprobados y se lanzará una excepción *TypeError* si se les llama con la firma incorrecta. Para los objetos simulados que sustituyen a una clase, su valor de retorno (la “instancia”) tendrá la misma especificación que la clase. Consultar la función `create_autospec()` y *Autoespecificación* para más detalles.

En lugar de `autospec=True`, puedes pasar `autospec=some_object` para utilizar un objeto arbitrario como especificación en lugar del objeto reemplazado.

Por defecto, `patch()` fallará al intentar reemplazar atributos que no existen. Si pasas `create=True` y no existe el atributo, `patch` crea el atributo cuando la función se llama y lo elimina de nuevo en cuanto termina de ejecutarse. Esto es útil para implementar pruebas para atributos que tu código de producción crea en tiempo de ejecución. Está desactivado por defecto, ya que puede ser peligroso. ¡Al activarlo se pueden implementar pruebas que validen APIs que en realidad no existen!

Nota: Distinto en la versión 3.5: Si estás parcheando objetos incorporados (builtins) en un módulo, no es necesario pasar `create=True`, ya que en este caso se agrega de forma predeterminada.

`Patch` puede ser usado como un decorador de la clase `TestCase`. Funciona decorando cada uno de los métodos de prueba presentes en la clase. Esto reduce el código repetitivo cuando tus métodos de prueba comparten un conjunto de parcheo común. `patch()` encuentra las pruebas mediante la búsqueda de métodos cuyos nombres comienzan con `patch.TEST_PREFIX`. Por defecto es `'test'`, que coincide con la forma en que *unittest* busca las pruebas. Se puede especificar un prefijo alternativo estableciendo un nuevo valor para el atributo `patch.TEST_PREFIX`.

`Patch` puede ser usado como un gestor de contexto, con la declaración `with`. En este caso el parcheo se aplica al bloque sangrado inmediatamente después de la declaración `with`. Si utilizas «as», el objeto parcheado será enlazado al nombre especificado después de «as»; muy útil si `patch()` está creando un objeto simulado automáticamente.

`patch()` toma argumentos por palabra clave arbitrarios. Estos serán pasados a *Mock* (o a *new_callable*) durante la construcción.

`patch.dict(...)`, `patch.multiple(...)` y `patch.object(...)` están disponibles para casos de uso alternativos.

`patch()` como decorador de función, crea el objeto simulado por ti y lo pasa a la función decorada:

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

Parchear una clase sustituye a la clase por una *instancia* de *MagicMock*. Si la clase se instancia en el código bajo prueba, el atributo `return_value` del objeto simulado será el utilizado.

Si la clase se instancia en múltiples ocasiones, puedes utilizar el atributo `side_effect` para retornar un nuevo objeto simulado cada vez. O también puedes establecer `return_value` para que sea lo que tu quieras.

Para configurar valores de retorno de métodos en *instancias* de la clase parcheada debes hacer uso del atributo `return_value`. Por ejemplo:

```
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
```

Si utilizas `spec` o `spec_set` y `patch()` está reemplazando una *clase*, el valor de retorno del objeto simulado creado tendrá las mismas especificaciones:

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

El argumento `new_callable` es útil cuando deseas utilizar una clase por defecto alternativa a `MagicMock` para el objeto simulado creado. Por ejemplo, si quieres que se use una clase `NonCallableMock` por defecto:

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
...     thing()
...
Traceback (most recent call last):
...
TypeError: 'NonCallableMock' object is not callable
```

Otro caso de uso podría ser la sustitución de un objeto por una instancia de `io.StringIO`:

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

Cuando `patch()` crea un objeto simulado para ti, a menudo lo primero que tienes que hacer es configurar el objeto simulado recién creado. Parte de la configuración se puede hacer en la propia llamada a `patch`. Cualquier palabra clave arbitraria que pases a la llamada será utilizada para establecer los atributos del objeto simulado creado:

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

Los atributos de los objetos simulados hijos, como `return_value` y `side_effect`, también puede ser configurados en la llamada, dado que los objetos simulados hijos son atributos en sí mismos del objeto simulado padre creado. Eso sí, estos no son sintácticamente válidos para ser pasados directamente como argumentos por palabras clave a la función `patch`, pero un diccionario con ellos como claves sí que puede ser expandido en una llamada a `patch()` usando `**`:

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
...
KeyError
```

Por defecto, el intento de parchear una función en un módulo (o un método o atributo en una clase) que no existe fallará lanzando una excepción `AttributeError`:

```
>>> @patch('sys.non_existing_attribute', 42)
... def test():
...     assert sys.non_existing_attribute == 42
...
>>> test()
Traceback (most recent call last):
...
AttributeError: <module 'sys' (built-in)> does not have the attribute 'non_existing'
↪
```

pero añadir `create=True` en la llamada a `patch()` hará que el ejemplo previo funcione como se esperaba:

```
>>> @patch('sys.non_existing_attribute', 42, create=True)
... def test(mock_stdout):
...     assert sys.non_existing_attribute == 42
...
>>> test()
```

Distinto en la versión 3.8: `patch()` ahora retorna una instancia de `AsyncMock` si el objetivo es una función asíncrona.

patch.object

`patch.object(target, attribute, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs)`
 parchea el miembro `attribute` invocado de un objeto `target` con un objeto simulado.

`patch.object()` se puede utilizar como un decorador, decorador de clase o un gestor de contexto. Los argumentos `new`, `spec`, `create`, `spec_set`, `autospec` y `new_callable` tienen el mismo significado que en la función `patch()`. Al igual que `patch()`, `patch.object()` toma argumentos por palabras clave arbitrarios para la configuración del objeto simulado que crea.

Cuando se utiliza como un decorador de clase `patch.object()` se rige por `patch.TEST_PREFIX` a la hora de elegir los métodos a envolver.

Puedes llamar a `patch.object()` con tres argumentos o con dos argumentos. La forma de tres argumento toma el objeto a parchear, el nombre del atributo y el objeto con el que reemplazar el atributo.

Al llamar usando la variante de dos argumento se omite el objeto de sustitución, por lo tanto se crea un objeto simulado automáticamente y se pasa como argumento adicional a la función decorada:

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
...
>>> test()
```

`spec`, `create` y el resto de argumentos de `patch.object()` tienen el mismo significado que en la función `patch()`.

patch.dict

`patch.dict(in_dict, values=(), clear=False, **kwargs)`

Parchea un diccionario o un objeto similar a un diccionario y posteriormente lo restaura a su estado original una vez terminada la prueba.

`in_dict` puede ser un diccionario o un contenedor similar a uno. Si se trata de un objeto similar a un diccionario, debe soportar como mínimo el establecimiento, la obtención y la eliminación de elementos, además de permitir la iteración sobre las claves.

`in_dict` también puede ser una cadena que especifique el nombre del diccionario a parchear, el nombre es usado para obtener el diccionario mediante importación.

`values` puede ser otro diccionario con valores para ser añadidos al diccionario. `values` también puede ser un iterable de parejas (`clave, valor`).

Si `clear` es verdadero, el contenido del diccionario se borrará antes de establecer los nuevos valores.

La función `patch.dict()` también puede ser invocada con argumentos por palabra clave arbitrarios para establecer los valores en el diccionario.

Distinto en la versión 3.8: La función `patch.dict()` ahora retorna el diccionario parcheado cuando se utiliza como gestor de contexto.

`patch.dict()` se puede utilizar como gestor de contexto, decorador o decorador de clase:

```
>>> foo = {}
>>> @patch.dict(foo, {'newkey': 'newvalue'})
... def test():
...     assert foo == {'newkey': 'newvalue'}
>>> test()
>>> assert foo == {}
```

Cuando se utiliza `patch.dict()` como decorador de clase, se rige por `patch.TEST_PREFIX` (por defecto 'test') a la hora de elegir qué métodos envolver:

```
>>> import os
>>> import unittest
>>> from unittest.mock import patch
>>> @patch.dict('os.environ', {'newkey': 'newvalue'})
... class TestSample(unittest.TestCase):
...     def test_sample(self):
...         self.assertEqual(os.environ['newkey'], 'newvalue')
```

Si deseas utilizar un prefijo diferente para tu prueba, puede informar a los parcheadores del nuevo prefijo a usar estableciendo `patch.TEST_PREFIX`. Para más detalles sobre cómo cambiar el valor del atributo consultar [`TEST_PREFIX`](#).

`patch.dict()` puede utilizarse para agregar miembros a un diccionario, o simplemente para dejar que una prueba modifique un diccionario y asegurarte de que el diccionario será restablecido cuando finalice la misma.

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}) as patched_foo:
...     assert foo == {'newkey': 'newvalue'}
...     assert patched_foo == {'newkey': 'newvalue'}
...     # You can add, update or delete keys of foo (or patched_foo, it's the same_
...     ↪dict)
...     patched_foo['spam'] = 'eggs'
...
>>> assert foo == {}
>>> assert patched_foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

Se pueden utilizar argumentos por palabra clave en la llamada a `patch.dict()` para establecer valores en el diccionario:

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

`patch.dict()` se puede utilizar con objetos similares a un diccionario, que no son realmente diccionarios. Estos objetos como mínimo deben permitir obtener, establecer y borrar elementos, además de soportar la iteración o la prueba de pertenencia. Dichas funcionalidades se corresponden a los métodos mágicos `__getitem__()`, `__setitem__()`, `__delitem__()` e `__iter__()` o `__contains__()`.

```
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

patch.multiple

`patch.multiple(target, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs)`

Realiza múltiples parches en una sola llamada. Se toma el objeto a ser parcheado (ya sea como un objeto o una cadena de caracteres con el nombre del mismo para obtener el objeto mediante importación) y los argumentos por palabras clave para los parches:

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
    ...
```

Usa `DEFAULT` como valor si deseas que la función `patch.multiple()` cree objetos simulados automáticamente por ti. En este caso, los objetos simulados creados son pasados a la función decorada mediante palabra clave y se retorna un diccionario cuando `patch.multiple()` se utiliza como gestor de contexto.

La función `patch.multiple()` se puede utilizar como un decorador, decorador de clase o como gestor de contexto. Los argumentos `spec`, `spec_set`, `create`, `autospec` y `new_callable` tienen el mismo significado que en la función `patch()`. Estos argumentos se aplicarán a todos los parches realizados por `patch.multiple()`.

Cuando se utiliza como decorador de clase, `patch.multiple()` se rige por `patch.TEST_PREFIX` a la hora de elegir qué métodos envolver.

Si deseas que `patch.multiple()` cree objetos simulados automáticamente por ti, puedes utilizar `DEFAULT` como valor. Si utilizas `patch.multiple()` como decorador, entonces los objetos simulados creados se pasan a la función decorada por palabra clave:

```
>>> thing = object()
>>> other = object()

>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

`patch.multiple()` se puede anidar junto a otros decoradores `patch`, pero los argumentos pasados por palabra clave se deben agregar *después* de cualquier argumento estándar creado por `patch()`:

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

Si `patch.multiple()` se utiliza como un gestor de contexto, el valor retornado por el mismo es un diccionario en el que los objetos simulados creados son almacenados, usando sus nombres como claves:

```
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
```

Métodos start y stop de patch

Todos los parcheadores tienen los métodos `start()` y `stop()`. Esto facilita parchear en los métodos `setUp` o cuando deseas hacer múltiples parches sin usar decoradores anidados o declaraciones `with`.

Para utilizar estos métodos, llama a `patch()`, `patch.object()` o `patch.dict()` como haces normalmente y mantén una referencia al objeto patcher retornado. A continuación, puedes llamar al método `start()` para aplicar el parche y a `stop()` para deshacerlo.

Si estás utilizando `patch()` para crear un objeto simulado automáticamente, este será retornado por la llamada a `patcher.start()`:

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

Un uso típico de esto podría ser realizar múltiples parches en el método `setUp` de un `TestCase`:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

Prudencia: Si se utiliza esta técnica debes asegurarte de que el parcheo está «sin aplicar» llamando a `stop`. Esto puede ser más complicado de lo que parece, ya que si se produce una excepción en el `setUp` no se llama a `tearDown` a continuación. `unittest.TestCase.addCleanup()` hace que esto sea más fácil:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
... 
```

Como beneficio adicional, ya no es necesario mantener una referencia al objeto `patcher`.

También es posible detener todos los parches que han sido iniciados usando `patch.stopall()`.

`patch.stopall()`

Detiene todos los parches activos. Sólo se detienen parches iniciados con `start`.

Parchear objetos incorporados (builtins)

Puedes parchear cualquier objeto incorporado dentro de un módulo. El siguiente ejemplo parchea la función incorporada `ord()`:

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

TEST_PREFIX

Todos los parcheadores se puede utilizar como decoradores de clase. Cuando se usan de esta forma, todos los métodos a probar en la clase son envueltos. Los parcheadores reconocen los métodos que comienzan con 'test' como métodos a probar. Esta es la misma forma que la clase `unittest.TestLoader` usa para encontrar métodos de prueba por defecto.

Cabe la posibilidad de que desees utilizar un prefijo diferente para las pruebas. Puedes informar a los parcheadores del cambio de prefijo estableciendo el atributo `patch.TEST_PREFIX`:

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

Anidando decoradores patch

Si deseas aplicar múltiples parches, solo tienes que apilar los decoradores.

Puede apilar múltiples decoradores patch usando el siguiente patrón:

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

Ten en cuenta que los decoradores se aplican desde abajo hacia arriba. Esta es la manera estándar de Python para aplicar decoradores. El orden en el que los objetos simulados generados se pasan a la función de prueba coincide con este orden.

Dónde parchear

`patch()` funciona cambiando (temporalmente) el objeto al que apunta *name* con otro. Puede haber muchos nombres apuntando a cualquier objeto individual, por lo que para que el parcheo funcione, debes asegurarte de parchear el nombre utilizado para el objeto por el sistema concreto bajo prueba.

El principio básico es parchear donde un objeto es *buscado*, que no es necesariamente el mismo lugar donde está definido. Un par de ejemplos ayudarán a aclarar esto.

Imaginemos que tenemos un proyecto, que queremos probar, con la siguiente estructura:

```
a.py
-> Defines SomeClass

b.py
-> from a import SomeClass
-> some_function instantiates SomeClass
```

Ahora queremos probar `some_function`, pero queremos simular `SomeClass` utilizando `patch()`. El problema es que cuando importamos el módulo `b`, lo cual tenemos que hacer obligatoriamente, se importa `SomeClass` del módulo `a`. Si utilizamos `patch()` para simular `a.SomeClass`, no tendrá ningún efecto en nuestra prueba; el módulo `b` ya tiene una referencia a la `SomeClass` real, por lo que parece que nuestro parcheo no tuvo ningún efecto.

La clave es parchear `SomeClass` donde se utiliza (o donde es buscado). En este caso `some_function` realmente va a buscar `SomeClass` en el módulo `b`, donde lo hemos importado. La aplicación de parches debe ser similar a:

```
@patch('b.SomeClass')
```

Sin embargo, ten en cuenta el escenario alternativo donde en el módulo `b` en lugar de `from a import SomeClass` se hace `import a` y `some_function` usa `a.SomeClass`. Ambas formas de importación son comunes. En este caso, la clase que queremos parchear está siendo buscada en el módulo, por lo que tenemos que parchear `a.SomeClass`:

```
@patch('a.SomeClass')
```

Parcheando descriptores y objetos proxy

Tanto `patch` como `patch.object` parchean descriptores correctamente y los restauran posteriormente: métodos de clase, métodos estáticos y propiedades. Los descriptores deben ser parcheados en la *clase* y no en una instancia. También funcionan con *algunos* objetos que actúan como proxy en el acceso a atributos, como el objeto de configuraciones de django (django settings object).

26.5.4 MagicMock y el soporte de métodos mágicos

Simular métodos mágicos

`Mock` soporta la simulación de los métodos de protocolo de Python, también conocidos como «métodos mágicos». Esto permite a los objetos simulados reemplazar contenedores u otros objetos que implementan protocolos de Python.

Dado que los métodos mágicos se buscan de manera diferente a los métodos normales², este soporte ha sido especialmente implementado. Esto significa que sólo es compatible con métodos mágicos específicos. La lista de métodos soportados incluye *casi* todos los existentes. Si hay alguno que falta y que consideras necesario, por favor hánznoslo saber.

Puedes simular métodos mágicos estableciendo el método que te interesa en una función o en una instancia simulada. Si utilizas una función, *debe* aceptar `self` como primer argumento³.

² Los métodos mágicos *deben* ser buscados en la clase en lugar de en la instancia. Diferentes versiones de Python son inconsistentes sobre la aplicación de esta regla. Los métodos de protocolo soportados deben trabajar con todas las versiones de Python.

³ La función está básicamente conectada a la clase, pero cada instancia `Mock` se mantiene aislada de las demás.

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

Un caso de uso para esto es simular objetos usados como gestores de contexto en una declaración `with`:

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

Las llamadas a métodos mágicos no aparecen registradas en `method_calls`, aunque sí se registran en `mock_calls`.

Nota: Si se utiliza el argumento por palabra clave `spec` para crear un objeto simulado, intentar después establecer un método mágico que no está en la especificación lanzará una excepción `AttributeError`.

La lista completa de los métodos mágicos soportados es la siguiente:

- `__hash__`, `__sizeof__`, `__repr__` y `__str__`
- `__dir__`, `__format__` y `__subclasses__`
- `__round__`, `__floor__`, `__trunc__` y `__ceil__`
- Comparaciones: `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__` y `__ne__`
- Métodos de contenedores: `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`, `__len__`, `__iter__`, `__reversed__` y `__missing__`
- Administrador de contexto: `__enter__`, `__exit__`, `__aenter__` y `__aexit__`
- Métodos numéricos unarios: `__neg__`, `__pos__` y `__invert__`
- Los métodos numéricos (incluyendo los métodos estándar y las variantes `in-place`): `__add__`, `__sub__`, `__mul__`, `__matmul__`, `__div__`, `__truediv__`, `__floordiv__`, `__mod__`, `__divmod__`, `__lshift__`, `__rshift__`, `__and__`, `__xor__`, `__or__`, y `__pow__`
- Métodos de conversión numérica: `__complex__`, `__int__`, `__float__` y `__index__`
- Métodos de descriptores: `__get__`, `__set__` y `__delete__`
- Pickling: `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` y `__setstate__`

- Representación de rutas del sistema de archivos: `__fspath__`
- Métodos de iteración asíncronos: `__aiter__` y `__anext__`

Distinto en la versión 3.8: Se añadió soporte para `os.PathLike.__fspath__()`.

Distinto en la versión 3.8: Se añadió soporte para `__aenter__`, `__aexit__`, `__aiter__` y `__anext__`.

Los siguientes métodos existen pero *no* están soportados, ya sea porque son usados por el propio objeto simulado, porque no se pueden establecer dinámicamente o porque pueden causar problemas:

- `__getattr__`, `__setattr__`, `__init__` y `__new__`
- `__prepare__`, `__instancecheck__`, `__subclasscheck__` y `__del__`

Magic Mock

Hay dos variantes de `MagicMock`: `MagicMock` y `NonCallableMagicMock`.

class `unittest.mock.MagicMock(*args, **kw)`

`MagicMock` es una subclase de `Mock` con implementaciones por defecto de la mayoría de los métodos mágicos. Puedes utilizar `MagicMock` para crear objetos simulados sin tener que establecer los métodos mágicos por ti mismo.

Los parámetros del constructor tienen el mismo significado que en `Mock`.

Si utilizas los argumentos `spec` o `spec_set` entonces *solo* los métodos mágicos que existan en la especificación serán creados.

class `unittest.mock.NonCallableMagicMock(*args, **kw)`

Una versión no invocable de `MagicMock`.

Los parámetros del constructor tienen el mismo significado que en `MagicMock`, con la excepción de `return_value` y `side_effect` que no tienen significado en un objeto simulado no invocable.

Los métodos mágicos están implementados mediante objetos `MagicMock`, por lo que puedes configurarlos y utilizarlos de la forma habitual:

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

Por defecto, muchos de los métodos de protocolo están obligados a retornar objetos de un tipo específico. Estos métodos están preconfigurados con un valor de retorno por defecto, de manera que puedan ser utilizados sin tener que hacer nada más, siempre que no estés interesado en el valor de retorno. En todo caso, puedes *establecer* el valor de retorno de forma manual si deseas cambiar el valor predeterminado.

Métodos y sus valores por defecto:

- `__lt__`: `NotImplemented`
- `__gt__`: `NotImplemented`
- `__le__`: `NotImplemented`
- `__ge__`: `NotImplemented`
- `__int__`: `1`
- `__contains__`: `False`
- `__len__`: `0`
- `__iter__`: `iter([])`
- `__exit__`: `False`

- `__aexit__`: False
- `__complex__`: 1j
- `__float__`: 1.0
- `__bool__`: True
- `__index__`: 1
- `__hash__`: hash predeterminado del objeto simulado
- `__str__`: str predeterminado del objeto simulado
- `__sizeof__`: sizeof predeterminado del objeto simulado

Por ejemplo:

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

Los dos métodos de igualdad, `__eq__()` y `__ne__()`, son especiales. Realizan la comparación de igualdad predeterminada basada en la identidad, utilizando el atributo `side_effect`, a menos que cambies su valor de retorno para que retorne alguna otra cosa:

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
>>> mock == 3
True
```

El valor de retorno de `MagicMock.__iter__()` puede ser cualquier objeto iterable, no se requiere que sea un iterador:

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

Si el valor de retorno es un iterador, iterar sobre él una vez que se consume, y cualquier iteración posterior, resultará en una lista vacía:

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

`MagicMock` tiene todos los métodos mágicos soportados configurados a excepción de algunos de los más oscuros y obsoletos. De todas formas, puedes configurar estos también si lo deseas.

Los métodos mágicos que son compatibles, pero que no están configurados por defecto en `MagicMock` son:

- `__subclasses__`

- `__dir__`
- `__format__`
- `__get__`, `__set__` y `__delete__`
- `__reversed__` y `__missing__`
- `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` y `__setstate__`
- `__getformat__` y `__setformat__`

26.5.5 Ayudantes

sentinel

`unittest.mock.sentinel`

El objeto `sentinel` proporciona una manera conveniente de proporcionar objetos únicos para tus pruebas.

Los atributos se crean a demanda, en el instante en el que se accede a ellos por su nombre por primera vez. El acceso al mismo atributo siempre retornará el mismo objeto. Los objetos retornados tienen una reproducción (`repr`) apropiada para que los mensajes de error de la prueba sean legibles.

Distinto en la versión 3.7: Los atributos `sentinel` ahora preservan su identidad cuando son *copiados* o *serializados con pickle*.

A veces, cuando implementas pruebas, necesitas probar que un determinado objeto se pasa como argumento a otro método, o se retorna. Crear objetos centinela con un nombre para probar esto puede ser una práctica común. `sentinel` proporciona una manera conveniente de crear y probar la identidad de este tipo de objetos.

En el siguiente ejemplo, parcheamos `method` para retornar `sentinel.some_object`:

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> sentinel.some_object
sentinel.some_object
```

DEFAULT

`unittest.mock.DEFAULT`

El objeto `DEFAULT` es un centinela pre-creado (actualmente `sentinel.DEFAULT`). Puede ser usado por las funciones *side_effect* para indicar que el valor de retorno normal debe ser utilizado.

call

`unittest.mock.call(*args, **kwargs)`

`call()` es un objeto ayudante que puede usarse para hacer aserciones simples, para comparar con `call_args`, `call_args_list`, `mock_calls` y `method_calls`. `call()` y también se puede utilizar con `assert_has_calls()`.

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]
True
```

`call.call_list()`

Para un objeto `call` que representa múltiples llamadas, el método `call_list()` retorna una lista con todas las llamadas intermedias, así como la llamada final.

`call_list` es particularmente útil para hacer aserciones sobre «llamadas encadenadas». Una llamada encadenada es varias llamadas en una sola línea de código. Esto resulta en múltiples entradas en el atributo `mock_calls` de un objeto simulado. Construir manualmente la secuencia de llamadas puede ser tedioso.

`call_list()` puede construir la secuencia de llamadas desde la misma llamada encadenada:

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1),
 call().method(arg='foo'),
 call().method().other('bar'),
 call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

Un objeto `call`, dependiendo de cómo fuera construido, puede ser una tupla de la forma (argumentos posicionales, argumentos por palabras clave) o bien de la forma (nombre, argumentos posicionales, argumentos por palabras clave). Esto no es particularmente interesante cuando los construyes por ti mismo, pero la introspección de los objetos `call` que conforman los atributos `Mock.call_args`, `Mock.call_args_list` y `Mock.mock_calls` si pueden ser de utilidad para acceder a los argumentos individuales que contienen.

Los objetos `call` en `Mock.call_args` y `Mock.call_args_list` están conformados por dos tuplas (argumentos posicionales, argumentos por palabra clave) mientras que los objetos `call` en `Mock.mock_calls`, junto con los que construyas por ti mismo, constan de tres tuplas (nombre, argumentos posicionales, argumentos por palabra clave).

Puedes utilizar esta presentación en forma de tuplas para obtener los argumentos individuales, con la finalidad de llevar a cabo introspección y aserciones más complejas. Los argumentos posicionales son una tupla (vacía si no hay ninguno) y los argumentos por palabra clave son un diccionario:

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> kall.args
(1, 2, 3)
>>> kall.kwargs
{'arg': 'one', 'arg2': 'two'}
>>> kall.args is kall[0]
True
>>> kall.kwargs is kall[1]
True
```

```
>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg': 'two', 'arg2': 'three'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

`unittest.mock.create_autospec(spec, spec_set=False, instance=False, **kwargs)`

Crea un nuevo objeto simulado utilizando otro objeto (*spec*) como especificación. Los atributos del objeto simulado utilizarán el atributo correspondiente del objeto *spec* como su especificación.

Los argumentos de las funciones o métodos simulados se validarán para asegurarse de que son invocados con la firma correcta.

Si *spec_set* es `True`, tratar de establecer atributos que no existen en el objeto especificado lanzará una excepción `AttributeError`.

Si se utiliza una clase como especificación, el valor de retorno del objeto simulado (la instancia de la clase) tendrá la misma especificación. Se puede utilizar una clase como especificación de una instancia pasando *instance=True*. El objeto simulado retornado sólo será invocable si las instancias del objeto simulado son también invocables.

`create_autospec()` también acepta argumentos por palabra clave arbitrarios, que son pasados al constructor del objeto simulado creado.

Consultar [Autoespecificación](#) para ver ejemplos de cómo utilizar la autoespecificación mediante `create_autospec()` y el argumento *autospec* de `patch()`.

Distinto en la versión 3.8: `create_autospec()` ahora retorna un objeto `AsyncMock` si el objetivo a simular es una función asíncrona.

ANY

`unittest.mock.ANY`

A veces puede que necesites hacer aserciones sobre *algunos* de los argumentos de una llamada al objeto simulado, pero sin preocuparte por el resto, o puede que quieras hacer uso de ellos de forma individual fuera de `call_args` y hacer aserciones más complejas con ellos.

Para ignorar ciertos argumentos puedes pasar objetos que se comparan como iguales con *cualquier cosa*. En este caso, las llamadas a `assert_called_with()` y `assert_called_once_with()` tendrán éxito sin importar lo que se haya pasado realmente.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

`ANY` también se puede utilizar en las comparaciones con las listas de llamadas, como `mock_calls`:

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

FILTER_DIR

`unittest.mock.FILTER_DIR`

`FILTER_DIR` es una variable definida a nivel de módulo que controla la forma en la que los objetos simulados responden a `dir()` (sólo para Python 2.6 y en adelante). El valor predeterminado es `True`, que utiliza el filtrado descrito a continuación, con la finalidad de mostrar solo los miembros considerados como útiles. Si no te gusta este filtrado, o necesitas desactivarlo con fines de diagnóstico, simplemente establece `mock.FILTER_DIR = False`.

Con el filtrado activado, `dir(some_mock)` mostrará solo atributos útiles y además incluirá cualquier atributo creado dinámicamente, que normalmente no se mostraría. Si el objeto simulado fue creado con un *spec* (o *autospec*) se muestran todos los atributos del objeto original, incluso si no se ha accedido a ellos todavía:

```
>>> dir(Mock())
['assert_any_call',
 'assert_called',
 'assert_called_once',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'assert_not_called',
 'attach_mock',
 ...
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...]
```

Muchos de los atributos con subrayado y doble subrayado no muy útiles (atributos privados de *Mock* y no del objeto real que se está simulando) se han filtrado del resultado de llamar a `dir()` en un objeto *Mock*. Si no te gusta este comportamiento, puedes desactivarlo estableciendo el modificador a nivel de módulo `FILTER_DIR`:

```
>>> from unittest import mock
>>> mock.FILTER_DIR = False
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...]
```

Como alternativa, puedes usar `vars(my_mock)` (miembros de instancia) y `dir(type(my_mock))` (miembros de tipo) para omitir el filtrado con independencia del valor de `mock.FILTER_DIR`.

mock_open

`unittest.mock.mock_open(mock=None, read_data=None)`

Una función auxiliar que permite crear un objeto simulado con la finalidad de reemplazar el uso de `open()`. Funciona para simular llamadas directas a `open()` y para aquellos casos en los que es utilizada como gestor de contexto.

El argumento *mock* es el objeto simulado a configurar. Si es `None` (por defecto) un objeto *MagicMock* se creará para ti, con la API limitada a métodos o atributos disponibles en los gestores de fichero estándares.

read_data es una cadena de caracteres para los métodos `read()`, `readline()` y `readlines()` del gestor de fichero a retornar. Las llamadas a los métodos tomarán datos de *read_data* hasta que se agote. El objeto

simulado de estos métodos es bastante simple: cada vez que el *mock* se llama, *read_data* se rebobina hasta el principio. Si necesitas más control sobre los datos que estás proporcionando al código de prueba tendrás que personalizar el objeto simulado. Cuando eso no sea suficiente, alguno de los paquetes que implementan sistemas de archivos en memoria disponible en [PyPI](#) puede ofrecer un sistema de archivos realista para usar en las pruebas.

Distinto en la versión 3.4: Se añadió soporte para *readline()* y *readlines()*. El objeto simulado de *read()* ahora consume datos de *read_data*, en lugar de retornarlo en cada llamada.

Distinto en la versión 3.5: *read_data* ahora es restablecido en cada llamada al *mock*.

Distinto en la versión 3.8: Se añadió el método *__iter__()* a la implementación, de manera que la iteración (como ocurre en los bucles) consume apropiadamente *read_data*.

Usar *open()* como gestor de contexto es una buena manera de asegurarse de que los gestores de archivos se cierren correctamente al terminar y se está convirtiendo en una práctica común:

```
with open('/some/path', 'w') as f:
    f.write('something')
```

El problema es que, incluso si simulas la llamada a *open()*, es el *objeto retornado* el que se utiliza como gestor de contexto (lo que conlleva que sus métodos *__enter__()* y *__exit__()* son invocados).

Simular gestores de contexto con un *MagicMock* es lo suficientemente común y complicado como para que una función auxiliar sea útil:

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

Y para la lectura de archivos:

```
>>> with patch('__main__.open', mock_open(read_data='bibble')) as m:
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

Autoespecificación

La autoespecificación se basa en la característica *spec* ya existente en el objeto simulado. Limita la API de los objetos simulados a la API del objeto original (la especificación), pero es recursiva (implementada de forma perezosa), de modo que los atributos del objeto simulado sólo tienen la misma API que los atributos de la especificación. Además, las funciones / métodos simuladas tienen la misma firma de llamada que la original y lanzan una excepción *TypeError* si se les llama incorrectamente.

Antes de explicar cómo funciona la autoespecificación, he aquí por qué se necesita.

Mock es un objeto muy potente y flexible, pero adolece de dos defectos cuando se utiliza para simular objetos de un sistema bajo prueba. Uno de estos defectos es específico de la API de *Mock* y el otro es un problema más genérico referente al uso de objetos simulados.

En primer lugar, el problema específico a `Mock.Mock` tiene dos métodos de aserción que son extremadamente útiles: `assert_called_with()` y `assert_called_once_with()`.

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
Traceback (most recent call last):
...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

Debido a que los objetos simulados crean automáticamente atributos según demanda y además permiten que se les llame con argumentos arbitrarios, si escribes mal uno de estos métodos de aserción, la utilidad de esa aserción desaparece:

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assret_called_once_with(4, 5, 6)
```

Tus pruebas pueden pasar silenciosamente y de forma incorrecta debido al error tipográfico.

El segundo problema es algo más general en las simulaciones. Si refactorizas parte de tu código, cambias el nombre de los miembros, etc., las pruebas para el código que siguen utilizando la *antigua API*, pero utilizan objetos simulados en lugar de los objetos reales, todavía pasarán las pruebas. Esto significa que tus pruebas pueden validarlo todo sin problemas, a pesar de que el código esté roto.

Ten en cuenta que esta es otra razón por la que necesitas pruebas de integración además de pruebas unitarias. Probar todo de forma aislada está muy bien, pero si no pruebas cómo están «conectadas entre sí» tus unidades, todavía hay mucho espacio para errores que las pruebas deberían haber detectado.

`mock` ya proporciona una característica para ayudar con esto, llamada especificación. Si se utiliza una clase o instancia como atributo `spec` de un objeto simulado, entonces solo puedes acceder a los atributos del mismo que existe en la clase real:

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with
Traceback (most recent call last):
...
AttributeError: Mock object has no attribute 'assret_called_with'
```

La especificación sólo se aplica al propio objeto simulado, por lo que aún tenemos el mismo problema con cualquiera de los métodos del mismo:

```
>>> mock.has_data()
<mock.Mock object at 0x...>
>>> mock.has_data.assret_called_with()
```

La autoespecificación resuelve este problema. Puedes pasar `autospec=True` a `patch()` / `patch.object()` o utilizar la función `create_autospec()` para crear un objeto simulado con una especificación. Si utilizas el argumento `autospec=True` de `patch()`, el objeto que se va a reemplazar se utiliza como objeto de especificación. Debido a que la especificación se hace «perezosamente» (la especificación se crea en el instante en el que se accede a los atributos del objeto simulado, no antes), se puede utilizar con objetos muy complejos o anidados (como módulos que importan módulos que, a su vez, importan otros módulos) sin un gran impacto en el rendimiento.

He aquí un ejemplo de ello en acción:

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
```

(continué en la próxima página)

(proviene de la página anterior)

```
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

Se puede ver que `request.Request` tiene una especificación. `request.Request` toma dos argumentos en el constructor (uno de los cuales es *self*). Esto es lo que sucede si tratamos de llamarlo de forma incorrecta:

```
>>> req = request.Request()
Traceback (most recent call last):
...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

La especificación también se aplica a las clases instanciadas (es decir, el valor de retorno de los objetos simulados especificados):

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='...'>
```

Los objetos de la clase `Request` no son invocables, por lo que el valor de retorno de una instancia de nuestro objeto simulado de `request.Request` no es invocable. Con la especificación, en cambio, cualquier error tipográfico en nuestras aserciones generará el error correcto:

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assert_called_with
Traceback (most recent call last):
...
AttributeError: Mock object has no attribute 'assert_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

En muchos casos, podrás simplemente añadir `autospec=True` a tus llamadas `patch()` existentes y así estar protegido contra errores tipográficos y cambios de la API.

Además de poder usar *autospec* a través de `patch()`, existe la función `create_autospec()` para crear directamente objetos simulados autoespecificados:

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

Sin embargo, este no es el comportamiento predeterminado, ya que no está exento de advertencias y restricciones. Con el fin de conocer qué atributos están disponibles en el objeto especificado, *autospec* tiene que hacer uso de introspección sobre la especificación (acceder a los atributos). A medida que recorres los atributos en el objeto simulado, se produce paralelamente y de forma soterrada un recorrido del objeto original. Si alguno de tus objetos especificados tienen propiedades o descriptores que puedan desencadenar la ejecución de código, quizás no deberías utilizar la autoespecificación. De todas formas, es siempre mucho mejor diseñar tus objetos de modo que la introspección sea segura⁴.

Un problema más serio es que es común que los atributos de instancia sean creados en el método `__init__()` y que no existan a nivel de clase. *autospec* no puede tener conocimiento de ningún atributo creado dinámicamente, por lo que restringe la API a atributos visibles:

```
>>> class Something:
...     def __init__(self):
...         self.a = 33
... 
```

(continué en la próxima página)

⁴ Esto sólo se aplica a las clases u objetos ya instanciados. Llamar a una clase simulada para crear una instancia simulada *no* crea una instancia real. Solo se llevan a cabo las búsqueda de atributos (mediante llamadas a `dir()`).

(proviene de la página anterior)

```
>>> with patch('__main__.Something', autospec=True):
...     thing = Something()
...     thing.a
...
Traceback (most recent call last):
...
AttributeError: Mock object has no attribute 'a'
```

Hay diferentes formas de resolver este problema. La forma más sencilla, pero no necesariamente la menos molesta, es simplemente establecer los atributos necesarios en el objeto simulado después de la creación del mismo. El hecho de que *autospec* no te permita buscar atributos que no existen en la especificación no impide que los establezcas manualmente después:

```
>>> with patch('__main__.Something', autospec=True):
...     thing = Something()
...     thing.a = 33
...
...
Traceback (most recent call last):
...
AttributeError: Mock object has no attribute 'a'
```

Existe una versión más agresiva de *spec* y *autospec* que impide establecer atributos inexistentes. Esto es útil si deseas asegurarte de que tu código solo *establece* atributos válidos, pero obviamente evitando este escenario en particular:

```
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...     thing = Something()
...     thing.a = 33
...
Traceback (most recent call last):
...
AttributeError: Mock object has no attribute 'a'
```

Probablemente la mejor manera de resolver el problema es añadir atributos de clase como valores por defecto para los miembros de instancia inicializados en el método `__init__()`. Ten en cuenta que, si sólo estás estableciendo atributos predeterminados en `__init__()`, proporcionarlos a través de atributos de clase (que, por supuesto, son compartidos entre instancias) también es más rápido.

```
class Something:
    a = 33
```

Esto nos plantea otro problema. Es relativamente común proporcionar un valor predeterminado de `None` para los miembros que posteriormente se convertirán en objetos de un tipo diferente. `None` es inútil como especificación dado que no permite acceder a *ningún* atributo o método. Como `None` *nunca* será útil como especificación, y probablemente indica un miembro que normalmente será de algún otro tipo en el futuro, la autoespecificación no usa una especificación para aquellos miembros configurados con `None` como valor. Estos serán simplemente objetos simulados ordinarios (MagicMocks en realidad):

```
>>> class Something:
...     member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

Si modificar tus clases en producción para agregar valores predeterminados no es de tu agrado, dispones de otras opciones. Una de ellas es simplemente usar una instancia como especificación en lugar de la clase. La otra es crear una subclase de la clase en producción y agregar los valores predeterminados a la subclase, sin afectar a la clase en producción. Ambas alternativas requieren que uses un objeto alternativo como especificación. Afortunadamente *patch()* admite esto, simplemente tienes que pasar el objeto alternativo mediante el argumento *autospec*:

```
>>> class Something:
...     def __init__(self):
```

(continué en la próxima página)

(proviene de la página anterior)

```

...     self.a = 33
...
>>> class SomethingForTest(Something):
...     a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>

```

Sellar objetos simulados

`unittest.mock.seal(mock)`

Seal desactivará la creación automática de objetos simulados al acceder a un atributo del objeto simulado que está siendo sellado, o a cualquiera de sus atributos que ya sean objetos simulados de forma recursiva.

Si una instancia simulada con un nombre o una especificación es asignada a un atributo no será considerada en la cadena de sellado. Esto permite evitar que seal fije partes del objeto simulado.

```

>>> mock = Mock()
>>> mock.submock.attribute1 = 2
>>> mock.not_submock = mock.Mock(name="sample_name")
>>> seal(mock)
>>> mock.new_attribute # This will raise AttributeError.
>>> mock.submock.attribute2 # This will raise AttributeError.
>>> mock.not_submock.attribute2 # This won't raise.

```

Nuevo en la versión 3.7.

26.6 unittest.mock — getting started

Nuevo en la versión 3.3.

26.6.1 Usando Mock

Mock Patching Methods

Common uses for *Mock* objects include:

- Patching methods
- Recording method calls on objects

You might want to replace a method on an object to check that it is called with the correct arguments by another part of the system:

```

>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>

```

Once our mock has been used (`real.method` in this example) it has methods and attributes that allow you to make assertions about how it has been used.

Nota: In most of these examples the *Mock* and *MagicMock* classes are interchangeable. As the *MagicMock* is the more capable class it makes a sensible one to use by default.

(proviene de la página anterior)

```
...     result = some_function()
...     assert result == 'the result'
```

Naming your mocks

It can be useful to give your mocks a name. The name is shown in the repr of the mock and can be helpful when the mock appears in test failure messages. The name is also propagated to attributes or methods of the mock:

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

Tracking all Calls

Often you want to track more than a single call to a method. The `mock_calls` attribute records all calls to child attributes of the mock - and also to their children.

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

If you make an assertion about `mock_calls` and any unexpected methods have been called, then the assertion will fail. This is useful because as well as asserting that the calls you expected have been made, you are also checking that they were made in the right order and with no additional calls:

You use the `call` object to construct lists for comparing with `mock_calls`:

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

However, parameters to calls that return mocks are not recorded, which means it is not possible to track nested calls where the parameters used to create ancestors are important:

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

Setting Return Values and Attributes

Setting the return values on a mock object is trivially easy:

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

Of course you can do the same for methods on the mock:

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

The return value can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

If you need an attribute setting on your mock, just do it:

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

Sometimes you want to mock up a more complex situation, like for example `mock.connection.cursor().execute("SELECT 1")`. If we wanted this call to return a list, then we have to configure the result of the nested call.

We can use `call` to construct the set of calls in a «chained call» like this for easy assertion afterwards:

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

It is the call to `.call_list()` that turns our call object into a list of calls representing the chained calls.

Raising exceptions with mocks

A useful attribute is `side_effect`. If you set this to an exception class or instance then the exception will be raised when the mock is called.

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
...
Exception: Boom!
```

Side effect functions and iterables

`side_effect` can also be set to a function or an iterable. The use case for `side_effect` as an iterable is where your mock is going to be called several times, and you want each call to return a different value. When you set `side_effect` to an iterable every call to the mock returns the next value from the iterable:

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

For more advanced use cases, like dynamically varying the return values depending on what the mock is called with, `side_effect` can be a function. The function will be called with the same arguments as the mock. Whatever the function returns is what the call returns:

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
1
>>> mock(2, 3)
2
```

Mocking asynchronous iterators

Since Python 3.8, `AsyncMock` and `MagicMock` have support to mock async-iterators through `__aiter__`. The `return_value` attribute of `__aiter__` can be used to set the return values to be used for iteration.

```
>>> mock = MagicMock() # AsyncMock also works here
>>> mock.__aiter__.return_value = [1, 2, 3]
>>> async def main():
...     return [i async for i in mock]
...
>>> asyncio.run(main())
[1, 2, 3]
```

Mocking asynchronous context manager

Since Python 3.8, `AsyncMock` and `MagicMock` have support to mock async-context-managers through `__aenter__` and `__aexit__`. By default, `__aenter__` and `__aexit__` are `AsyncMock` instances that return an async function.

```
>>> class AsyncContextManager:
...     async def __aenter__(self):
...         return self
...     async def __aexit__(self, exc_type, exc, tb):
...         pass
...
>>> mock_instance = MagicMock(AsyncContextManager()) # AsyncMock also works here
>>> async def main():
...     async with mock_instance as result:
...         pass
...
>>>
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> asyncio.run(main())
>>> mock_instance.__aenter__.assert_awaited_once()
>>> mock_instance.__aexit__.assert_awaited_once()
```

Creating a Mock from an Existing Object

One problem with over use of mocking is that it couples your tests to the implementation of your mocks rather than your real code. Suppose you have a class that implements `some_method`. In a test for another class, you provide a mock of this object that *also* provides `some_method`. If later you refactor the first class, so that it no longer has `some_method` - then your tests will continue to pass even though your code is now broken!

`Mock` allows you to provide an object as a specification for the mock, using the `spec` keyword argument. Accessing methods / attributes on the mock that don't exist on your specification object will immediately raise an attribute error. If you change the implementation of your specification, then tests that use that class will start failing immediately without you having to instantiate the class in those tests.

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
...
AttributeError: object has no attribute 'old_method'
```

Using a specification also enables a smarter matching of calls made to the mock, regardless of whether some parameters were passed as positional or named arguments:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, 3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(a=1, b=2, c=3)
```

If you want this smarter matching to also work with method calls on the mock, you can use *auto-speccking*.

If you want a stronger form of specification that prevents the setting of arbitrary attributes as well as the getting of them then you can use `spec_set` instead of `spec`.

26.6.2 Patch Decorators

Nota: With `patch()` it matters that you patch objects in the namespace where they are looked up. This is normally straightforward, but for a quick guide read *where to patch*.

A common need in tests is to patch a class attribute or a module attribute, for example patching a builtin or patching a class in a module to test that it is instantiated. Modules and classes are effectively global, so patching on them has to be undone after the test or the patch will persist into other tests and cause hard to diagnose problems.

`mock` provides three convenient decorators for this: `patch()`, `patch.object()` and `patch.dict()`. `patch` takes a single string, of the form `package.module.Class.attribute` to specify the attribute you are patching. It also optionally takes a value that you want the attribute (or class or whatever) to be replaced with. “`patch.object`” takes an object and the name of the attribute you would like patched, plus optionally the value to patch it with.

`patch.object`:

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
```

(continúe en la próxima página)

(proviene de la página anterior)

```

... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original

>>> @patch('package.module.attribute', sentinel.attribute)
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()

```

If you are patching a module (including *builtins*) then use *patch()* instead of *patch.object()*:

```

>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"

```

The module name can be “dotted”, in the form *package.module* if needed:

```

>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()

```

A nice pattern is to actually decorate test methods themselves:

```

>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original

```

If you want to patch with a Mock, you can use *patch()* with only one argument (or *patch.object()* with two arguments). The mock will be created for you and passed into the test function / method:

```

>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()

```

You can stack up multiple patch decorators using this pattern:

```

>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1, MockClass1)
...         self.assertIs(package.module.ClassName2, MockClass2)

```

(continué en la próxima página)

(proviene de la página anterior)

```
...
>>> MyTest('test_something').test_something()
```

When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *Python* order that decorators are applied). This means from the bottom up, so in the example above the mock for `test_module.ClassName2` is passed in first.

There is also `patch.dict()` for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

`patch`, `patch.object` and `patch.dict` can all be used as context managers.

Where you use `patch()` to create a mock for you, you can get a reference to the mock using the «as» form of the with statement:

```
>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

As an alternative patch, `patch.object` and `patch.dict` can be used as class decorators. When used in this way it is the same as applying the decorator individually to every method whose name starts with «test».

26.6.3 Further Examples

Here are some more examples for some slightly more advanced scenarios.

Mocking chained calls

Mocking chained calls is actually straightforward with mock once you understand the `return_value` attribute. When a mock is called for the first time, or you fetch its `return_value` before it has been called, a new *Mock* is created.

This means that you can see how the object returned from a call to a mocked object has been used by interrogating the `return_value` mock:

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

From here it is a simple step to configure and then make assertions about chained calls. Of course another alternative is writing your code in a more testable way in the first place...

So, suppose we have some code that looks a little bit like this:


```
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
...         response = self.backend.get_endpoint('foobar').create_call('spam',
↪ 'eggs').start_call()
...         # more code
```

Assuming that `BackendProvider` is already well tested, how do we test `method()`? Specifically, we want to test that the code section `# more code` uses the response object in the correct way.

As this chain of calls is made from an instance attribute we can monkey patch the `backend` attribute on a `Something` instance. In this particular case we are only interested in the return value from the final call to `start_call` so we don't have much configuration to do. Let's assume the object it returns is "file-like", so we'll ensure that our response object uses the builtin `open()` as its `spec`.

To do this we create a mock instance as our mock backend and create a mock response object for it. To set the response as the return value for that final `start_call` we could do this:

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_
↪ value = mock_response
```

We can do that in a slightly nicer way using the `configure_mock()` method to directly set the return value for us:

```
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call.
↪ return_value': mock_response}
>>> mock_backend.configure_mock(**config)
```

With these we monkey patch the «mock backend» in place and can make the real call:

```
>>> something.backend = mock_backend
>>> something.method()
```

Using `mock_calls` we can check the chained call with a single assert. A chained call is several calls in one line of code, so there will be several entries in `mock_calls`. We can use `call.call_list()` to create this list of calls for us:

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

Partial mocking

In some tests I wanted to mock out a call to `datetime.date.today()` to return a known date, but I didn't want to prevent the code under test from creating new date objects. Unfortunately `datetime.date` is written in C, and so I couldn't just monkey-patch out the static `date.today()` method.

I found a simple way of doing this that involved effectively wrapping the date class with a mock, but passing through calls to the constructor to the real class (and returning real instances).

The `patch decorator` is used here to mock out the `date` class in the module under test. The `side_effect` attribute on the mock date class is then set to a lambda function that returns a real date. When the mock date class is called a real date will be constructed and returned by `side_effect`.

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
```

(continué en la próxima página)

(proviene de la página anterior)

```

...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)

```

Note that we don't patch `datetime.date` globally, we patch `date` in the module that *uses* it. See [where to patch](#).

When `date.today()` is called a known date is returned, but calls to the `date(...)` constructor still return normal dates. Without this you can find yourself having to calculate an expected result using exactly the same algorithm as the code under test, which is a classic testing anti-pattern.

Calls to the `date` constructor are recorded in the `mock_date` attributes (`call_count` and `friends`) which may also be useful for your tests.

An alternative way of dealing with mocking dates, or other builtin classes, is discussed in [this blog entry](#).

Mocking a Generator Method

A Python generator is a function or method that uses the `yield` statement to return a series of values when iterated over¹.

A generator method / function is called to return the generator object. It is the generator object that is then iterated over. The protocol method for iteration is `__iter__()`, so we can mock this using a [MagicMock](#).

Here's an example class with an «`iter`» method implemented as a generator:

```

>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]

```

How would we mock this class, and in particular its «`iter`» method?

To configure the values returned from the iteration (implicit in the call to `list`), we need to configure the object returned by the call to `foo.iter()`.

```

>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]

```

Applying the same patch to every test method

If you want several patches in place for multiple test methods the obvious way is to apply the patch decorators to every method. This can feel like unnecessary repetition. For Python 2.6 or more recent you can use `patch()` (in all its various forms) as a class decorator. This applies the patches to all test methods on the class. A test method is identified by methods whose names start with `test`:

```

>>> @patch('mymodule.SomeClass')
... class MyTest(unittest.TestCase):
...
...     def test_one(self, MockSomeClass):

```

(continué en la próxima página)

¹ There are also generator expressions and more advanced uses of generators, but we aren't concerned about them here. A very good introduction to generators and how powerful they are is: [Generator Tricks for Systems Programmers](#).

(proviene de la página anterior)

```

...     self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def test_two(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def not_a_test(self):
...         return 'something'
...
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'

```

An alternative way of managing patches is to use the *Métodos start y stop de patch*. These allow you to move the patching into your `setUp` and `tearDown` methods.

```

>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()

```

If you use this technique you must ensure that the patching is «undone» by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the `setUp` then `tearDown` is not called. `unittest.TestCase.addCleanup()` makes this easier:

```

>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
>>> MyTest('test_foo').run()

```

Mocking Unbound Methods

Whilst writing tests today I needed to patch an *unbound method* (patching the method on the class rather than on the instance). I needed `self` to be passed in as the first argument because I want to make asserts about which objects were calling this particular method. The issue is that you can't patch with a mock for this, because if you replace an unbound method with a mock it doesn't become a bound method when fetched from the instance, and so it doesn't get `self` passed in. The workaround is to patch the unbound method with a real function instead. The `patch()` decorator makes it so simple to patch out methods with a mock that having to create a real function becomes a nuisance.

If you pass `autospec=True` to `patch` then it does the patching with a *real* function object. This function object has the same signature as the one it is replacing, but delegates to a mock under the hood. You still get your mock auto-created in exactly the same way as before. What it means though, is that if you use it to patch out an unbound method on a class the mocked function will be turned into a bound method if it is fetched from an instance. It will have `self` passed in as the first argument, which is exactly what I wanted:

```
>>> class Foo:
...     def foo(self):
...         pass
...
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...     mock_foo.return_value = 'foo'
...     foo = Foo()
...     foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

If we don't use `autospec=True` then the unbound method is patched out with a `Mock` instance instead, and isn't called with `self`.

Checking multiple calls with mock

`mock` has a nice API for making assertions about how your mock objects are used.

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

If your mock is only being called once you can use the `assert_called_once_with()` method that also asserts that the `call_count` is one.

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
...
AssertionError: Expected to be called once. Called 2 times.
```

Both `assert_called_with` and `assert_called_once_with` make assertions about the *most recent* call. If your mock is going to be called several times, and you want to make assertions about *all* those calls you can use `call_args_list`:

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

The `call` helper makes it easy to make assertions about these calls. You can build up a list of expected calls and compare it to `call_args_list`. This looks remarkably similar to the repr of the `call_args_list`:

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

Coping with mutable arguments

Another situation is rare, but can bite you, is when your mock is called with mutable arguments. `call_args` and `call_args_list` store *references* to the arguments. If the arguments are mutated by the code under test then you can no longer make assertions about what the values were when the mock was called.

Here's some example code that shows the problem. Imagine the following functions defined in “mymodule”:

```
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

When we try to test that `grob` calls `frob` with the correct argument look what happens:

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

One possibility would be for mock to copy the arguments you pass in. This could then cause problems if you do assertions that rely on object identity for equality.

Here's one solution that uses the `side_effect` functionality. If you provide a `side_effect` function for a mock then `side_effect` will be called with the same args as the mock. This gives us an opportunity to copy the arguments and store them for later assertions. In this example I'm using *another* mock to store the arguments so that I can use the mock methods for doing the assertion. Again a helper function sets this up for me.

```
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

`copy_call_args` is called with the mock that will be called. It returns a new mock that we do the assertion on. The `side_effect` function makes a copy of the args and calls our `new_mock` with the copy.

Nota: If your mock is only going to be used once there is an easier way of checking arguments at the point they are called. You can simply do the checking inside a `side_effect` function.

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
...
AssertionError
```

An alternative approach is to create a subclass of `Mock` or `MagicMock` that copies (using `copy.deepcopy()`) the arguments. Here's an example implementation:

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, /, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super().__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
...
AssertionError: Expected call: mock({1})
Actual call: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

When you subclass `Mock` or `MagicMock` all dynamically created attributes, and the `return_value` will use your subclass automatically. That means all children of a `CopyingMock` will also have the type `CopyingMock`.

Nesting Patches

Using `patch` as a context manager is nice, but if you do multiple patches you can end up with nested `with` statements indenting further and further to the right:

```
>>> class MyTest(unittest.TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

With unittest cleanup functions and the *Métodos start y stop de patch* we can achieve the same effect without the

nested indentation. A simple helper method, `create_patch`, puts the patch in place and returns the created mock for us:

```
>>> class MyTest(unittest.TestCase):
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

Mocking a dictionary with `MagicMock`

You may want to mock a dictionary, or other container object, recording all access to it whilst having it still behave like a dictionary.

We can do this with `MagicMock`, which will behave like a dictionary, and using `side_effect` to delegate dictionary access to a real underlying dictionary that is under our control.

When the `__getitem__()` and `__setitem__()` methods of our `MagicMock` are called (normal dictionary access) then `side_effect` is called with the key (and in the case of `__setitem__` the value too). We can also control what is returned.

After the `MagicMock` has been used we can use attributes like `call_args_list` to assert about how the dictionary was used:

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
...     return my_dict[name]
...
>>> def setitem(name, val):
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

Nota: An alternative to using `MagicMock` is to use `Mock` and *only* provide the magic methods you specifically want:

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

A *third* option is to use `MagicMock` but passing in `dict` as the `spec` (or `spec_set`) argument so that the `MagicMock` created only has dictionary magic methods available:

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

With these side effect functions in place, the `mock` will behave like a normal dictionary but recording the access. It even raises a `KeyError` if you try to access a key that doesn't exist.

```
>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
Traceback (most recent call last):
...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

After it has been used you can make assertions about the access using the normal mock methods and attributes:

```
>>> mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'b': 'fish', 'c': 3, 'd': 'eggs'}
```

Mock subclasses and their attributes

There are various reasons why you might want to subclass `Mock`. One reason might be to add helper methods. Here's a silly example:

```
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has_been_called()
False
>>> mymock()
>>> mymock.has_been_called()
True
```

The standard behaviour for `Mock` instances is that attributes and the return value mocks are of the same type as the mock they are accessed on. This ensures that `Mock` attributes are `Mocks` and `MagicMock` attributes are `MagicMocks`². So if you're subclassing to add helper methods then they'll also be available on the attributes and return value mock of instances of your subclass.

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
```

(continué en la próxima página)

² An exception to this rule are the non-callable mocks. Attributes use the callable variant because otherwise non-callable mocks couldn't have callable methods.

(proviene de la página anterior)

```
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has_been_called()
True
```

Sometimes this is inconvenient. For example, [one user](#) is subclassing mock to create a [Twisted adaptor](#). Having this applied to attributes too actually causes errors.

Mock (in all its flavours) uses a method called `_get_child_mock` to create these «sub-mocks» for attributes and return values. You can prevent your subclass being used for attributes by overriding this method. The signature is that it takes arbitrary keyword arguments (`**kwargs`) which are then passed onto the mock constructor:

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, /, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

Mocking imports with `patch.dict`

One situation where mocking can be hard is where you have a local import inside a function. These are harder to mock because they aren't using an object from the module namespace that we can patch out.

Generally local imports are to be avoided. They are sometimes done to prevent circular dependencies, for which there is *usually* a much better way to solve the problem (refactor the code) or to prevent «up front costs» by delaying the import. This can also be solved in better ways than an unconditional local import (store the module as a class or module attribute and only do the import on first use).

That aside there is a way to use `mock` to affect the results of an import. Importing fetches an *object* from the `sys.modules` dictionary. Note that it fetches an *object*, which need not be a module. Importing a module for the first time results in a module object being put in `sys.modules`, so usually when you import something you get a module back. This need not be the case however.

This means you can use `patch.dict()` to temporarily put a mock in place in `sys.modules`. Any imports whilst this patch is active will fetch the mock. When the patch is complete (the decorated function exits, the with statement body is complete or `patcher.stop()` is called) then whatever was there previously will be restored safely.

Here's an example that mocks out the “fooble” module.

```
>>> import sys
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     import fooble
...     fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

As you can see the `import fooble` succeeds, but on exit there is no “fooble” left in `sys.modules`.

This also works for the `from module import name` form:

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     from fooble import blob
...     blob.blip()
...
<Mock name='mock.blob.blip()' id='... '>
>>> mock.blob.blip.assert_called_once_with()
```

With slightly more work you can also mock package imports:

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
...     fooble()
...
<Mock name='mock.module.fooble()' id='... '>
>>> mock.module.fooble.assert_called_once_with()
```

Tracking order of calls and less verbose call assertions

The `Mock` class allows you to track the *order* of method calls on your mock objects through the `method_calls` attribute. This doesn't allow you to track the order of calls between separate mock objects, however we can use `mock_calls` to achieve the same effect.

Because mocks track calls to child mocks in `mock_calls`, and accessing an arbitrary attribute of a mock creates a child mock, we can create our separate mocks from a parent one. Calls to those child mock will then all be recorded, in order, in the `mock_calls` of the parent:

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar
```

```
>>> mock_foo.something()
<Mock name='mock.foo.something()' id='... '>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='... '>
```

```
>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]
```

We can then assert about the calls, including the order, by comparing with the `mock_calls` attribute on the manager mock:

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

If patch is creating, and putting in place, your mocks then you can attach them to a manager mock using the `attach_mock()` method. After attaching calls will be recorded in `mock_calls` of the manager.

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
<MagicMock name='mock.MockClass1().foo()' id='... '>
```

(continué en la próxima página)

(proviene de la página anterior)

```
<MagicMock name='mock.MockClass2().bar()' id='... '>
>>> manager.mock_calls
[call.MockClass1(),
call.MockClass1().foo(),
call.MockClass2(),
call.MockClass2().bar()]
```

If many calls have been made, but you're only interested in a particular sequence of them then an alternative is to use the `assert_has_calls()` method. This takes a list of calls (constructed with the `call` object). If that sequence of calls are in `mock_calls` then the assert succeeds.

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='... '>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='... '>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

Even though the chained call `m.one().two().three()` aren't the only calls that have been made to the mock, the assert still succeeds.

Sometimes a mock may have several calls made to it, and you are only interested in asserting about *some* of those calls. You may not even care about the order. In this case you can pass `any_order=True` to `assert_has_calls`:

```
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)
```

More complex argument matching

Using the same basic concept as *ANY* we can implement matchers to do more complex assertions on objects used as arguments to mocks.

Suppose we expect some object to be passed to a mock that by default compares equal based on object identity (which is the Python default for user defined classes). To use `assert_called_with()` we would need to pass in the exact same object. If we are only interested in some of the attributes of this object then we can create a matcher that will check these attributes for us.

You can see in this example how a “standard” call to `assert_called_with` isn't sufficient:

```
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
Traceback (most recent call last):
...
AssertionError: Expected: call(<__main__.Foo object at 0x...>)
Actual call: call(<__main__.Foo object at 0x...>)
```

A comparison function for our `Foo` class might look something like this:

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

And a matcher object that can use comparison functions like this for its equality operation would look something like this:

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

Putting all this together:

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

The `Matcher` is instantiated with our compare function and the `Foo` object we want to compare against. In `assert_called_with` the `Matcher` equality method will be called, which compares the object the mock was called with against the one we created our matcher with. If they match then `assert_called_with` passes, and if they don't an `AssertionError` is raised:

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

With a bit of tweaking you could have the comparison function raise the `AssertionError` directly and provide a more useful failure message.

As of version 1.5, the Python testing library `PyHamcrest` provides similar functionality, that may be useful here, in the form of its equality matcher (`hamcrest.library.integration.match_equality`).

26.7 2to3 - Traducción de código Python 2 a 3

2to3 es un programa hecho en Python que lee código fuente en Python 2.x y aplica una serie de *fixers* para transformarlo en código Python 3.x válido. La librería estándar contiene un buen conjunto de *fixers* que se encargarán de casi todo el código. La librería soporte de 2to3 `lib2to3` es, de todas maneras, una librería flexible y genérica. Por lo cual es posible escribir sus propios *fixers* para 2to3. `lib2to3` también puede adaptarse a aplicaciones propias en las cuales el código en Python necesite editarse automáticamente.

26.7.1 Usando 2to3

2to3 generalmente estará instalada con el interprete de Python como un *script*. También se encuentra ubicada en el directorio `Tools/scripts` en la raíz de Python.

Los argumentos básicos de 2to3 son una lista de archivos o directorios a convertir. Los directorios se recorren recursivamente en búsqueda de archivos en Python.

Este es un ejemplo de un archivo en Python 2.x, :file: `example.py`:

```
def greet (name) :
    print "Hello, {0}!".format (name)
print "What's your name?"
name = raw_input()
greet (name)
```

Puede ser convertido a Python 3.x vía 2to3 desde la línea de comandos:

```
$ 2to3 example.py
```

Se imprime un *diff* del archivo fuente original. 2to3 también puede escribir las modificaciones necesarias directamente en el archivo fuente. (Se hace una copia de respaldo del archivo original a menos que se proporcione `-n`.) La escritura de los cambios se habilita con la opción `-w`:

```
$ 2to3 -w example.py
```

Después de la conversión, `example.py` se ve de la siguiente manera:

```
def greet (name) :
    print ("Hello, {0}!".format (name))
print("What's your name?")
name = input()
greet (name)
```

Los comentarios y la indentación exacta se conservan durante todo el proceso de conversión.

Por defecto, 2to3 corre un conjunto de *fixers predefinidos*. La opción `-l` lista todos los *fixers* posibles. Se puede pasar un conjunto explícito de *fixers* con la opción `-x`. Asimismo la opción `-x` deshabilita el *fixer* que se explicita. El siguiente ejemplo corre solo los *fixers* `imports` y `has_key`:

```
$ 2to3 -f imports -f has_key example.py
```

Este comando corre todos los *fixers* excepto el *apply fixer*:

```
$ 2to3 -x apply example.py
```

Algunos *fixers* son explícitos, esto quiere decir que no corren por defecto y deben ser listados en la línea de comando para que se ejecuten. Aquí, además de los *fixers* por defectos, se ejecuta el *fixer* `idioms`:

```
$ 2to3 -f all -f idioms example.py
```

Puede observarse que pasar `all` habilita todos los *fixers* por defecto.

Algunas veces 2to3 va a encontrar algo en su código que necesita ser modificado, pero 2to3 no puede hacerlo automáticamente. En estos casos, 2to3 va a imprimir una advertencia debajo del *diff* del archivo. Deberá tomar nota de la advertencia para obtener un código compatible con 3.x.

2to3 también puede refactorizar *doctest*. Para habilitar este modo, use la opción `-d`. Tenga en cuenta que *solo* los *doctest* serán refactorizados. Esto tampoco requiere que el módulo sea válido en Python. Por ejemplo, *doctest* de ejemplo en un documento reST también pueden ser refactorizados con esta opción.

La opción `-v` habilita la salida de más información en el proceso de conversión.

Como algunas declaraciones *print* pueden ser pasadas como llamadas a funciones o declaraciones, 2to3 no siempre lee archivos que contienen la función `print`. Cuando 2to3 detecta la presencia de la directiva de compilación `from __future__ import print_function`, modifica su gramática interna para interpretar `print()` como una función. Este cambio también puede habilitarse manualmente con la opción `-p`. Use la opción `-p` para ejecutar los *fixers* en el código al que ya se le han convertido sus declaraciones *print*.

La opción `-o` o la opción `--output-dir` permiten designar un directorio alternativo para que se guarden los archivos procesados. La opción `-n` es necesaria ya que los archivos de respaldo no tienen sentido cuando no se sobrescriben los archivos originales.

Nuevo en la versión 3.2.3: Se agregó la opción `:option:'!-o'`.

La opción `-W` o `--write-unchanged-files` le dice a 2to3 que siempre escriba archivos de salida, incluso si no se requieren hacer cambios en el archivo. Esto es muy útil con la opción `-o` para que copie el árbol completo de código Python con su conversión de un directorio a otro. Esta opción incluye a la opción `-w` ya que no tendría sentido de otra manera.

Nuevo en la versión 3.2.3: Se agregó la opción `-W`.

La opción `--add-suffix` agrega un texto al final de todos los nombres de archivo. La opción `-n` es necesaria, ya que las copias de respaldo no son necesarias cuando escribimos a un archivo con distinto nombre. Ejemplo:

```
$ 2to3 -n -W --add-suffix=3 example.py
```

Hará que se escriba un archivo convertido con el nombre `example.py3`.

Nuevo en la versión 3.2.3: Se agrega la opción `--add-suffix`.

Para convertir un proyecto entero de un árbol de directorios a otro use:

```
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
```

26.7.2 Fixers

Cada paso de la transformación del código es encapsulado en un *fixer*. El comando `2to3 -l` los lista. Como se *explicó arriba*, cada uno de estos puede habilitarse o deshabilitarse individualmente. En esta sección se los describe más detalladamente.

apply

Elimina el uso de `apply()`. Por ejemplo `apply(function, *args, **kwargs)` es convertido a `function(*args, **kwargs)`.

asserts

Reemplaza los nombre de método *unittest* en desuso por los correctos.

De	A
<code>failUnlessEqual(a, b)</code>	<code>assertEqual(a, b)</code>
<code>assertEquals(a, b)</code>	<code>assertEqual(a, b)</code>
<code>failIfEqual(a, b)</code>	<code>assertNotEqual(a, b)</code>
<code>assertNotEquals(a, b)</code>	<code>assertNotEqual(a, b)</code>
<code>failUnless(a)</code>	<code>assertTrue(a)</code>
<code>assert_(a)</code>	<code>assertTrue(a)</code>
<code>failIf(a)</code>	<code>assertFalse(a)</code>
<code>failUnlessRaises(exc, cal)</code>	<code>assertRaises(exc, cal)</code>
<code>failUnlessAlmostEqual(a, b)</code>	<code>assertAlmostEqual(a, b)</code>
<code>assertAlmostEquals(a, b)</code>	<code>assertAlmostEqual(a, b)</code>
<code>failIfAlmostEqual(a, b)</code>	<code>assertNotAlmostEqual(a, b)</code>
<code>assertNotAlmostEquals(a, b)</code>	<code>assertNotAlmostEqual(a, b)</code>

basestring

Convierte *basestring* a *str*.

buffer

Convierte `buffer` a `memoryview`. Este *fixer* es opcional porque la API `memoryview` es similar pero no exactamente la misma que la del `buffer`.

dict

Corrige los métodos de iteración del diccionario, `dict.iteritems()` es convertido a `dict.items()`, `dict.iterkeys()` a `dict.keys()`, y `dict.itervalues()` a `dict.values()`. Del mismo modo, `dict.viewitems()`, `dict.viewkeys()` y `dict.viewvalues()` son convertidos respectivamente a `dict.items()`, `dict.keys()` y `dict.values()`. También incluye los usos existentes de `dict.items()`, `dict.keys()`, y `dict.values()` en una llamada a `list`.

except

Convierte `except X, T` a `except X as T`.

exec

Convierte la declaración `exec` a la función `exec()`.

execfile

Elimina el uso de la función `execfile()`. El argumento para `execfile()` es encapsulado para las funciones `open()`, `compile()`, y `exec()`.

exitfunc

Cambia la declaración de `sys.exitfunc` para usar el módulo `atexit`.

filter

Encapsula la función `filter()` usando una llamada para la clase `list`.

funcattrs

Corrige los atributos de la función que fueron renombrados. Por ejemplo, `my_function.func_closure` es convertido a `my_function.__closure__`.

future

Elimina la declaración `from __future__ import new_feature`.

getcwdu

Renombra la función `os.getcwdu()` a `os.getcwd()`.

has_key

Cambia `dict.has_key(key)` a `key in dict`.

idioms

Este *fixer* opcional ejecuta varias transformaciones que tornan el código Python más idiomático. Comparaciones de tipo como `type(x) is SomeClass` y `type(x) == SomeClass` son convertidas a `isinstance(x, SomeClass)`. “`while 1`” cambia a `while True`. Este *fixer* también intenta hacer uso de `sorted()` en los lugares apropiados. Por ejemplo, en este bloque:

```
L = list(some_iterable)
L.sort()
```

es convertido a

```
L = sorted(some_iterable)
```

import

Detecta las importaciones entre hermanos y las convierte en importaciones relativas.

imports

Maneja los cambios de nombre de módulo en la librería estándar.

imports2

Maneja otros cambios de nombre de módulo en la biblioteca estándar. Está separada del *fixer* `imports` solo por motivos de limitaciones técnicas.

input

Convierte `input(prompt)` a `eval(input(prompt))`.

intern

Convierte `intern()` a `sys.intern()`.

isinstance

Corrige tipos duplicados en el segundo argumento de `isinstance()`. Por ejemplo, “`isinstance(x, (int, int))`” es convertido a `isinstance(x, int)` y `isinstance(x, (int, float, int))` es convertido a `isinstance(x, (int, float))`.

itertools_imports

Elimina importaciones de `itertools.ifilter()`, `itertools.izip()`, y `itertools.imap()`. Importación de `itertools.ifilterfalse()` también se cambian a `itertools.filterfalse()`.

itertools

Cambia el uso de `itertools.ifilter()`, `itertools.izip()`, y `itertools.imap()` para sus equivalentes integrados `itertools.ifilterfalse()` es cambiado a `itertools.filterfalse()`.

long

Renombra `long` a `int`.

map

Encapsula `map()` en una llamada a `list`. También cambia `map(None, x)` a `list(x)`. Usando “`from future_builtins import map`” se deshabilita este *fixer*.

metaclass

Convierte la vieja sintaxis de metaclasses (`__metaclass__ = Meta` en el cuerpo de la clase) a la nueva sintaxis (`class X(metaclass=Meta)`).

methodattrs

Corrige nombres de atributos de métodos antiguos. Por ejemplo, `meth.im_func` es convertido a `meth.__func__`.

ne

Convierte la antigua sintaxis no-igual, `<>`, a `!=`.

next

Convierte el uso de métodos iteradores `next()` para la función `next()`. También renombra métodos `next()` a `__next__()`.

nonzero

Renombra el método `__nonzero__()` a `__bool__()`.

numliterals

Convierte literales octales a la nueva sintaxis.

operator

Convierte llamadas para varias funciones en el módulo `operator` a otras, pero equivalentes, llamadas de funciones. Cuando es necesario, se agregan las declaraciones `import` apropiadas, por ejemplo `import collections.abc`. Se realizan los siguientes mapeos:

De	A
<code>operator.isCallable(obj)</code>	<code>callable(obj)</code>
<code>operator.sequenceIncludes(obj)</code>	<code>operator.contains(obj)</code>
<code>operator.isSequenceType(obj)</code>	<code>isinstance(obj, collections.abc.Sequence)</code>
<code>operator.isMappingType(obj)</code>	<code>isinstance(obj, collections.abc.Mapping)</code>
<code>operator.isNumberType(obj)</code>	<code>isinstance(obj, numbers.Number)</code>
<code>operator.repeat(obj, n)</code>	<code>operator.mul(obj, n)</code>
<code>operator.irepeat(obj, n)</code>	<code>operator.imul(obj, n)</code>

paren

Agrega los paréntesis extra donde sean necesarios en las listas de comprensión. Por ejemplo `[x for x in 1, 2]` se convierte en `[x for x in (1, 2)]`.

print

Convierte la declaración `print` en la función `print()`.

raise

Convierte `raise E, V` a `raise E(V)`, y `raise E, V, T` a `raise E(V).with_traceback(T)`. SI `E` es una tupla, la conversión será incorrecta porque sustituir tuplas por excepciones fue eliminado en Python 3.0.

raw_input

Conviertes `raw_input()` to `input()`.

reduce

Maneja el movimiento de `reduce()` a `functools.reduce()`.

reload

Convierte `reload()` a `importlib.reload()`.

renames

Cambia `sys.maxint` a `sys.maxsize`.

repr

Sustituye el *backtick* `repr` por la función `repr()`.

set_literal

Sustituye el uso de la clase constructora `set` por su literal. Este *fixer* es opcional.

standarderror

Renombra `StandardError` a `Exception`.

sys_exc

Cambia los `sys.exc_value`, `sys.exc_type`, `sys.exc_traceback` en desuso para usar la función `sys.exc_info()`.

throw

Corrige el cambio de la API en el método generador `throw()`.

tuple_params

Elimina el desempaquetamiento implícito del parámetro de tupla. Este *fixer* inserta variables temporarias.

types

Corrige el código roto por la remoción de algunos miembros en el módulo `types`.

unicode

Renombra `unicode` a `str`.

urllib

Maneja el renombramiento de los módulos `urllib` y `urllib2` para el paquete `urllib`.

ws_comma

Remueve el exceso de espacios blancos de los ítems separados por coma. Este *fixer* es opcional.

xrange

Renombra `xrange()` a `range()` y encapsula la llamada a la función existente `range()` con `list`.

xreadlines

Cambia `for x in file.xreadlines()` por `for x in file`.

zip

Encapsula el uso de la función `zip()` en una llamada a la clase `list`. Esto está deshabilitado cuando `from future_builtins import zip` aparece.

26.7.3 `lib2to3` - librería 2to3

Código fuente: [Lib/lib2to3/](#)

Nota: La API del módulo `lib2to3` debe considerarse inestable y puede cambiar drásticamente en el futuro.

26.8 `test` — Paquete de pruebas de regresión para Python

Nota: El paquete `test` está destinado solo para uso interno de Python. Está documentado para beneficio de los desarrolladores principales de Python. Se desaconseja el uso de este paquete fuera de la biblioteca estándar de Python, ya que el código mencionado aquí puede cambiar o eliminarse sin previo aviso entre versiones de Python.

El paquete `test` contiene todas las pruebas de regresión para Python, así como los módulos `test.support` y `test.regrtest`. Se utiliza `test.support` para mejorar sus pruebas, mientras que `test.regrtest` maneja el conjunto de pruebas.

Cada módulo en el paquete `test` cuyo nombre comienza con `test_` es un conjunto de pruebas para un módulo o característica específica. Todas las pruebas nuevas deben escribirse usando el módulo `unittest` o `doctest`. Algunas pruebas anteriores se escriben utilizando un estilo de prueba «tradicional» que compara la salida impresa con `sys.stdout`; este estilo de prueba se considera obsoleto.

Ver también:

Módulo `unittest` Escritura de pruebas de regresión de PyUnit.

Módulo `doctest` Pruebas integradas en cadenas de caracteres de documentación.

26.8.1 Escritura de pruebas unitarias para el paquete `test`

Se prefiere que las pruebas que utilizan el módulo `unittest` sigan algunas pautas. Una es nombrar el módulo de prueba comenzándolo con `test_` y terminarlo con el nombre del módulo que se está probando. Los métodos de prueba en el módulo de prueba deben comenzar con `test_` y terminar con una descripción de lo que el método está probando. Esto es necesario para que el controlador de prueba reconozca los métodos como métodos de prueba. Por lo tanto, no se debe incluir una cadena de caracteres de documentación para el método. Se debe usar un comentario (como `# Tests function returns only True or False`) para proporcionar documentación para los métodos de prueba. Esto se hace porque las cadenas de documentación se imprimen si existen y, por lo tanto, no se indica qué prueba se está ejecutando.

A menudo se usa una plantilla básica:

```
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 ...
```

(continúe en la próxima página)

(proviene de la página anterior)

```

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 ...

if __name__ == '__main__':
    unittest.main()

```

Este patrón de código permite que el conjunto de pruebas sea ejecutado por `test.regrtest`, como un script que admite la CLI `unittest`, o mediante la CLI `python -m unittest`.

El objetivo de las pruebas de regresión es intentar romper el código. Esto lleva a algunas pautas a seguir:

- El conjunto de pruebas debe ejercer todas las clases, funciones y constantes. Esto incluye no solo la API externa que se presentará al mundo exterior sino también el código «privado».
- Se prefiere la prueba de caja blanca (examinar el código que se prueba cuando se escriben las pruebas). Las pruebas de caja negra (probar solo la interfaz de usuario publicada) no son lo suficientemente completas como para garantizar que se prueben todos los casos límite y límite.
- Asegúrese de que todos los valores posibles son probados, incluidos los no válidos. Esto asegura que no solo todos los valores válidos sean aceptables, sino que los valores incorrectos se manejen correctamente.
- Agote tantas rutas de código cómo sea posible. Pruebe donde se produce la ramificación y, por lo tanto, adapte la entrada para asegurarse de que se toman muchas rutas diferentes a través del código.
- Añada una prueba explícita para cualquier error descubierto para el código probado. Esto asegurará que el error no vuelva a aparecer si el código se cambia en el futuro.
- Asegúrese de limpiar después de sus pruebas (así como cerrar y eliminar todos los archivos temporales).
- Si una prueba depende de una condición específica del sistema operativo, verifique que la condición ya existe antes de intentar la prueba.
- Importe la menor cantidad de módulos posible y hágalo lo antes posible. Esto minimiza las dependencias externas de las pruebas y también minimiza el posible comportamiento anómalo de los efectos secundarios de importar un módulo.
- Intente maximizar la reutilización del código. En ocasiones, las pruebas variarán en algo tan pequeño como qué tipo de entrada se utiliza. Minimice la duplicación de código usando como clase base una clase de prueba básica con una clase que especifique la entrada:

```

class TestFuncAcceptsSequencesMixin:

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = 'abc'

```

(continué en la próxima página)

(proviene de la página anterior)

```
class AcceptTuples(TestFuncAcceptsSequencesMixin, unittest.TestCase):  
    arg = (1, 2, 3)
```

Cuando use este patrón, recuerde que todas las clases que heredan `unittest.TestCase` se ejecutan como pruebas. La clase `Mixin` en el ejemplo anterior no tiene ningún dato y, por lo tanto, no se puede ejecutar solo, por lo tanto, no hereda de `unittest.TestCase`.

Ver también:

Desarrollo dirigido por pruebas (*Test Driven Development*) Un libro de *Kent Beck* sobre pruebas de escritura antes del código.

26.8.2 Ejecución de pruebas utilizando la interfaz de línea de comandos

El paquete `test` puede ejecutarse como un script para controlar el conjunto de pruebas de regresión de Python, gracias a la opción `-m`: **`python -m test`**. Internamente, se utiliza `test.regrtest`; la llamada **`python -m test.regrtest`** utilizada en versiones anteriores de Python todavía funciona. Ejecuta el script por sí mismo automáticamente y comienza a ejecutar todas las pruebas de regresión en el paquete `test.regrtest`. Lo hace buscando todos los módulos en el paquete cuyo nombre comienza con `test_`, importándolos y ejecutando la función `test_main()` si está presente o cargando las pruebas a través de `unittest.TestLoader.loadTestsFromModule` si `test_main` no existe. Los nombres de las pruebas a ejecutar también se pueden pasar al script. La especificación de una prueba de regresión única (**`python -m test test_spam`**) minimizará la salida y solo imprimirá si la prueba pasó o no.

Ejecutando `test` directamente permite establecer qué recursos están disponibles para que se usen las pruebas. Para ello, use la opción de línea de comandos `-u`. Al especificar `all` como el valor para la opción `-u` se habilitan todos los recursos posibles: **`python -m test -uall`**. Si se desean todos los recursos menos uno (un caso más común), se puede enumerar una lista de recursos separados por comas que no se desean después de `all`. El comando **`python -m test -uall,-audio,-largefile`** ejecutará `test` con todos los recursos excepto los recursos `audio` y `largefile`. Para obtener una lista de todos los recursos y más opciones de línea de comandos, ejecute **`python -m test -h`**.

Algunas otras formas de ejecutar las pruebas de regresión dependen de en qué plataforma se ejecuten las pruebas. En Unix, puede ejecutar **`make test`** en el directorio de nivel superior donde se construyó Python. En Windows, ejecutar **`rt.bat`** desde su directorio `PCbuild` ejecutará todas las pruebas de regresión.

26.9 `test.support` — Utilidades para el conjunto de pruebas de Python

El módulo `test.support` proporciona soporte para el conjunto de pruebas de regresión de Python.

Nota: `test.support` no es un módulo público. Está documentado aquí para ayudar a los desarrolladores de Python a escribir pruebas. La API de este módulo está sujeta a cambios sin problemas de compatibilidad con versiones anteriores entre versiones.

Este módulo define las siguientes excepciones:

exception `test.support.TestFailed`

Excepción que se lanzará cuando una prueba falle. Esto está en desuso a favor de pruebas basadas en `unittest` en métodos de aserción `unittest.TestCase`

exception `test.support.ResourceDenied`

La subclase de `unittest.SkipTest`. Se lanza cuando un recurso (como una conexión de red) no está disponible. Lanzado por la función `requires()`.

El módulo `test.support` define las siguientes constantes:

`test.support.verbose`

True cuando la salida detallada está habilitada. Debe verificarse cuando se desea información más detallada sobre una prueba en ejecución. `verbose` está establecido por `test.regrtest`.

`test.support.is_jython`

True si el intérprete en ejecución es Jython.

`test.support.is_android`

True si el sistema es Android.

`test.support.unix_shell`

Ruta del shell si no está en Windows; de otra manera None.

`test.support.FS_NONASCII`

Un carácter no codificable ASCII codificable por `os.fsencode()`.

`test.support.TESTFN`

Establecido a un nombre que sea seguro de usar como nombre de un archivo temporal. Cualquier archivo temporal que se cree debe cerrarse y desvincularse (eliminarse).

`test.support.TESTFN_UNICODE`

Establecido un nombre que no sea ASCII para un archivo temporal.

`test.support.TESTFN_ENCODING`

Establecido en `sys.getfilesystemencoding()`.

`test.support.TESTFN_UNENCODABLE`

Establecido un nombre de archivo (tipo `str`) que no se pueda codificar mediante la codificación del sistema de archivos en modo estricto. Puede ser None si no es posible generar dicho nombre de archivo.

`test.support.TESTFN_UNDECODABLE`

Establecido un nombre de archivo (tipo de bytes) que no pueda decodificarse mediante la codificación del sistema de archivos en modo estricto. Puede ser None si no es posible generar dicho nombre de archivo.

`test.support.TESTFN_NONASCII`

Establecido un nombre de archivo que contiene el carácter `FS_NONASCII`.

`test.support.IPV6_ENABLED`

Establecido True si `IPV6` está habilitado en este host, de lo contrario False.

`test.support.SAVEDCWD`

Establecido `os.getcwd()`.

`test.support.PGO`

Establecido cuando se pueden omitir las pruebas cuando no son útiles para `PGO`.

`test.support.PIPE_MAX_SIZE`

Una constante que probablemente sea más grande que el tamaño del búfer de la tubería (*pipe*) del sistema operativo subyacente, para bloquear las escrituras.

`test.support.SOCK_MAX_SIZE`

Una constante que probablemente sea más grande que el tamaño del búfer del socket del sistema operativo subyacente para bloquear las escrituras.

`test.support.TEST_SUPPORT_DIR`

Establecido en el directorio de nivel superior que contiene `test.support`.

`test.support.TEST_HOME_DIR`

Establecido en el directorio de nivel superior para el paquete de prueba.

`test.support.TEST_DATA_DIR`

Establecido en el directorio data dentro del paquete de prueba.

`test.support.MAX_Py_ssize_t`

Establecido `sys.maxsize` para pruebas de gran memoria.

`test.support.max_memuse`

Establecido por `set_memlimit()` como límite de memoria para pruebas de memoria grande. Limitado por `MAX_Py_ssize_t`.

`test.support.real_max_memuse`

Establecido por `set_memlimit()` como límite de memoria para pruebas de memoria grande. No limitado por `MAX_Py_ssize_t`.

`test.support.MISSING_C_DOCSTRINGS`

Retorna True si se ejecuta en CPython, no en Windows, y la configuración no está configurada con `WITH_DOC_STRINGS`.

`test.support.HAVE_DOCSTRINGS`

Verifica la presencia de cadenas de documentos (*docstrings*).

`test.support.TEST_HTTP_URL`

Define la URL de un servidor HTTP dedicado para las pruebas de red.

`test.support.ALWAYS_EQ`

Objeto que es igual a cualquier cosa. Se utiliza para probar la comparación de tipos mixtos.

`test.support.LARGEST`

Objeto que es mayor que cualquier cosa (excepto a sí mismo). Se utiliza para probar la comparación de tipos mixtos.

`test.support.SMALLEST`

Objeto que es menor que cualquier cosa (excepto él mismo). Se utiliza para probar la comparación de tipos mixtos.

El módulo `test.support` define las siguientes funciones:

`test.support.forget(module_name)`

Elimina el módulo llamado `module_name` de `sys.modules` y elimina los archivos compilados por bytes del módulo.

`test.support.unload(name)`

Elimina `name` de `sys.modules`.

`test.support.unlink(filename)`

Llama a `os.unlink()` en `filename`. En plataformas Windows, esto está envuelto con un ciclo de espera que verifica la existencia del archivo.

`test.support.rmdir(filename)`

Llama a `os.rmdir()` en `filename`. En las plataformas Windows, esto está envuelto con un ciclo de espera que verifica la existencia del archivo.

`test.support.rmtree(path)`

Llama a `shutil.rmtree()` en `path` o `os.lstat()` y `os.rmdir()` para eliminar una ruta y su contenido. En las plataformas Windows, esto está envuelto con un ciclo de espera que verifica la existencia de los archivos.

`test.support.make_legacy_pyc(source)`

Mueve un archivo `pyc` de **PEP 3147/PEP 488** a su ubicación heredada y retorna la ruta del sistema de archivos al archivo de `pyc` heredado. El valor de origen es la ruta del sistema de archivos al archivo fuente. No es necesario que exista, sin embargo, debe existir el archivo **PEP 3147/488** `pyc`.

`test.support.is_resource_enabled(resource)`

Retorna True si `resource` está habilitado y disponible. La lista de recursos disponibles solo se establece cuando `test.regrtest` está ejecutando las pruebas.

`test.support.python_is_optimized()`

Retorna True si Python no fue construido con `-O0` o `-Og`.

`test.support.with_pymalloc()`

Retorna `_testcapi.WITH_PYMALLOC`.

`test.support.requires(resource, msg=None)`

Lanza `ResourceDenied` si `resource` no está disponible. `msg` es el argumento para `ResourceDenied` si se lanza. Siempre retorna `True` si es invocado por una función cuyo `__name__` es `'__main__'`. Se usa cuando se ejecutan pruebas por `test.regrtest`.

`test.support.system_must_validate_cert(f)`

Lanza `unittest.SkipTest` en fallos de validación de certificación `TLS`.

`test.support.sortdict(dict)`

Retorna una representación del *diccionario* con las claves ordenadas.

`test.support.findfile(filename, subdir=None)`

Retorna la ruta al archivo llamado `filename`. Si no se encuentra ninguna coincidencia, se retorna `filename`. Esto no equivale a un fallo, ya que podría ser la ruta al archivo.

La configuración `subdir` indica una ruta relativa a utilizar para encontrar el archivo en lugar de buscar directamente en los directorios de ruta.

`test.support.create_empty_file(filename)`

Crea un archivo vacío con `filename`. Si ya existe, truncarlo.

`test.support.fd_count()`

Cuenta el número de descriptores de archivo abiertos.

`test.support.match_test(test)`

Hace coincidir `test` con los patrones establecidos en `set_match_tests()`.

`test.support.set_match_tests(patterns)`

Define una prueba de coincidencia con una expresión regular `patterns`.

`test.support.run_unittest(*classes)`

Ejecuta subclases `unittest.TestCase` pasadas a la función. La función escanea las clases en busca de métodos que comiencen con el prefijo `test_` y ejecuta las pruebas individualmente.

También está permitido pasar cadenas de caracteres como parámetros; Estas deberían ser claves en `sys.modules`. Cada módulo asociado será escaneado por `unittest.TestLoader.loadTestsFromModule()`. Esto generalmente se ve en la siguiente función `test_main()`:

```
def test_main():
    support.run_unittest(__name__)
```

Esto ejecutará todas las pruebas definidas en el módulo nombrado.

`test.support.run_doctest(module, verbosity=None, optionflags=0)`

Ejecuta `doctest.testmod()` en `module` dado. Retorna `(failure_count, test_count)`.

Si `verbosity` es `None`, la `doctest.testmod()` se ejecuta con `verbosity` establecido en `verbose`. De lo contrario, se ejecuta con verbosidad establecida en `None`. `optionflags` se pasa como `optionflags` to `doctest.testmod()`.

`test.support.setswitchinterval(interval)`

Establecido `sys.setswitchinterval()` en el `intervalo` dado. Define un intervalo mínimo para los sistemas Android para evitar que el sistema se cuelgue.

`test.support.check_impl_detail(**guards)`

Usa esta comprobación para proteger las pruebas específicas de implementación de CPython o para ejecutarlas solo en las implementaciones protegidas por los argumentos:

```
check_impl_detail()           # Only on CPython (default).
check_impl_detail(jython=True) # Only on Jython.
check_impl_detail(cpython=False) # Everywhere except CPython.
```

`test.support.check_warnings(*filters, quiet=True)`

Un envoltorio de conveniencia para `warnings.catch_warnings()` que hace que sea más fácil probar que una advertencia se lanzó correctamente. Es aproximadamente equivalente a llamar a `warnings`.

`catch_warnings(record=True)` con `warnings.simplefilter()` establecido en `always` y con la opción de validar automáticamente los resultados que se registran.

`check_warnings` acepta 2 tuplas de la forma `("mensaje regexp", WarningCategory)` como argumentos posicionales. Si se proporcionan uno o más *filters*, o si el argumento opcional de palabra clave *quiet* es `False`, se verifica para asegurarse de que las advertencias sean las esperadas: cada filtro especificado debe coincidir con al menos una de las advertencias lanzadas por el código adjunto o la prueba falla, y si se lanzan advertencias que no coinciden con ninguno de los filtros especificados, la prueba falla. Para deshabilitar la primera de estas comprobaciones, configure *quiet* en `True`.

Si no se especifican argumentos, el valor predeterminado es:

```
check_warnings(("", Warning), quiet=True)
```

En este caso, se capturan todas las advertencias y no se lanzan errores.

En la entrada al administrador de contexto, se retorna una instancia de `WarningRecorder`. La lista de advertencias subyacentes de `catch_warnings()` está disponible a través del atributo `warnings` del objeto del registrador. Como conveniencia, también se puede acceder directamente a los atributos del objeto que representa la advertencia más reciente a través del objeto grabador (vea el ejemplo a continuación). Si no se ha lanzado ninguna advertencia, cualquiera de los atributos que de otro modo se esperarían en un objeto que representa una advertencia retornará `None`.

El objeto grabador (*recorder object*) también tiene un método `reset()`, que borra la lista de advertencias.

El administrador de contexto está diseñado para usarse así:

```
with check_warnings(("assertion is always true", SyntaxWarning),
                   ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

En este caso, si no se generó ninguna advertencia, o si surgió alguna otra advertencia, `check_warnings()` lanzaría un error.

Cuando una prueba necesita analizar más profundamente las advertencias, en lugar de simplemente verificar si ocurrieron o no, se puede usar un código como este:

```
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
```

Aquí se capturarán todas las advertencias, y el código de prueba prueba las advertencias capturadas directamente.

Distinto en la versión 3.2: Nuevos argumentos opcionales *filters* y *quiet*.

`test.support.check_no_resource_warning(testcase)`

Gestor de contexto para comprobar que no se ha lanzado un `ResourceWarning`. Debe eliminar el objeto que puede emitir `ResourceWarning` antes del final del administrador de contexto.

`test.support.set_memlimit(limit)`

Establece los valores para `max_memuse` y `real_max_memuse` para pruebas de memoria grande.

`test.support.record_original_stdout(stdout)`

Almacene el valor de `stdout`. Está destinado a mantener el `stdout` en el momento en que comenzó el regtest.

`test.support.get_original_stdout()`

Retorna el `stdout` original establecido por `record_original_stdout()` o `sys.stdout` si no está configurado.

`test.support.strip_python_stderr(stderr)`

Elimine el *stderr* de un proceso de Python de la salida de depuración potencial emitida por el intérprete. Esto normalmente se ejecutará en el resultado de `subprocess.Popen.communicate()`.

`test.support.args_from_interpreter_flags()`

Retorna una lista de argumentos de línea de comandos que reproducen la configuración actual en `sys.flags` y `sys.warnoptions`.

`test.support.optim_args_from_interpreter_flags()`

Retorna una lista de argumentos de línea de comandos que reproducen la configuración de optimización actual en `sys.flags`.

`test.support.captured_stdin()`

`test.support.captured_stdout()`

`test.support.captured_stderr()`

Un administrador de contexto que reemplaza temporalmente la secuencia nombrada con un objeto *io.StringIO*.

Ejemplo de uso con flujos de salida:

```
with captured_stdout() as stdout, captured_stderr() as stderr:
    print("hello")
    print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"
```

Ejemplo de uso con flujo de entrada:

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

`test.support.temp_dir(path=None, quiet=False)`

Un administrador de contexto que crea un directorio temporal en *path* y produce el directorio.

Si *path* es *None*, el directorio temporal se crea usando `tempfile.mkdtemp()`. Si *quiet* es *False*, el administrador de contexto lanza una excepción en caso de error. De lo contrario, si se especifica *path* y no se puede crear, solo se emite una advertencia.

`test.support.change_cwd(path, quiet=False)`

Un administrador de contexto que cambia temporalmente el directorio de trabajo actual a *path* y produce el directorio.

Si *quiet* es *False*, el administrador de contexto lanza una excepción en caso de error. De lo contrario, solo emite una advertencia y mantiene el directorio de trabajo actual igual.

`test.support.temp_cwd(name='tempcwd', quiet=False)`

Un administrador de contexto que crea temporalmente un nuevo directorio y cambia el directorio de trabajo actual (*CWD*).

El administrador de contexto crea un directorio temporal en el directorio actual con el nombre *name* antes de cambiar temporalmente el directorio de trabajo actual. Si *name* es *None*, el directorio temporal se crea usando `tempfile.mkdtemp()`.

Si *quiet* es *False* y no es posible crear o cambiar el *CWD*, se lanza un error. De lo contrario, solo se lanza una advertencia y se utiliza el *CWD* original.

`test.support.temp_umask(umask)`

Un administrador de contexto que establece temporalmente el proceso *umask*.

`test.support.transient_internet(resource_name, *, timeout=30.0, errnos=())`

Un administrador de contexto que plantea *ResourceDenied* cuando varios problemas con la conexión a

Internet se manifiestan como excepciones.

`test.support.disable_fault_handler()`

Un administrador de contexto que reemplaza `sys.stderr` con `sys.__stderr__`.

`test.support.gc_collect()`

Se fuerza la mayor cantidad posible de objetos para ser recolectados. Esto es necesario porque el recolector de basura no garantiza la desasignación oportuna. Esto significa que los métodos `__del__` pueden llamarse más tarde de lo esperado y las referencias débiles pueden permanecer vivas por más tiempo de lo esperado.

`test.support.disable_gc()`

Un administrador de contexto que deshabilita el recolector de basura al entrar y lo vuelve a habilitar al salir.

`test.support.swap_attr(obj, attr, new_val)`

Administrador de contexto para intercambiar un atributo con un nuevo objeto.

Uso:

```
with swap_attr(obj, "attr", 5):  
    ...
```

Esto establecerá `obj.attr` en 5 durante la duración del bloque `with`, restaurando el valor anterior al final del bloque. Si `attr` no existe en `obj`, se creará y luego se eliminará al final del bloque.

El valor anterior (o `None` si no existe) se asignará al objetivo de la cláusula «como», si existe.

`test.support.swap_item(obj, attr, new_val)`

Administrador de contexto para intercambiar un elemento con un nuevo objeto.

Uso:

```
with swap_item(obj, "item", 5):  
    ...
```

Esto establecerá `obj["item"]` a 5 durante la duración del bloque `with`, restaurando el valor anterior al final del bloque. Si `item` no existe en `obj`, se creará y luego se eliminará al final del bloque.

El valor anterior (o `None` si no existe) se asignará al objetivo de la cláusula «como», si existe.

`test.support.wait_threads_exit(timeout=60.0)`

El administrador de contexto debe esperar hasta que salgan todos los hilos creados en la declaración `with`.

`test.support.start_threads(threads, unlock=None)`

Administrador de contexto para iniciar `threads`. Intenta unir los hilos al salir.

`test.support.calcobjsize(fmt)`

Retorna `struct.calcsize()` para `nP{fmt}0n` o, si `gettotalrefcount` existe, `2PnP{fmt}0P`.

`test.support.calcvobjsize(fmt)`

Retorna `struct.calcsize()` para `nPnP{fmt}0n` o, si `gettotalrefcount` existe, `2PnPnP{fmt}0P`.

`test.support.checksizeof(test, o, size)`

Para el caso de prueba (*testcase*), se aserciona que el `sys.getsizeof` para `o` más el tamaño del encabezado GC es igual a *size*.

`test.support.can_symlink()`

Retorna `True` si el sistema operativo admite links simbólicos, de lo contrario `False`.

`test.support.can_xattr()`

Retorna `True` si el sistema operativo admite *xattr*, de lo contrario `False`.

`@test.support.skip_unless_symlink`

Un decorador para ejecutar pruebas que requieren soporte para enlaces simbólicos.

`@test.support.skip_unless_xattr`

Un decorador para ejecutar pruebas que requieren soporte para *xattr*.

`@test.support.skip_unless_bind_unix_socket`

Un decorador para ejecutar pruebas que requieren un enlace `bind()` para *sockets Unix*.

`@test.support.anticipate_failure (condition)`

Un decorador para marcar condicionalmente las pruebas con `unittest.expectedFailure()`. Cualquier uso de este decorador debe tener un comentario asociado que identifique el problema relevante del rastreador.

`@test.support.run_with_locale (catstr, *locales)`

Un decorador para ejecutar una función en una configuración regional diferente, restableciéndola correctamente una vez que ha finalizado. `catstr` es la categoría de configuración regional como una cadena (por ejemplo, "LC_ALL"). Las *locales* aprobadas se probarán secuencialmente y se utilizará la primera configuración regional válida.

`@test.support.run_with_tz (tz)`

Un decorador para ejecutar una función en una zona horaria específica, restableciéndola correctamente una vez que haya finalizado.

`@test.support.requires_freebsd_version (*min_version)`

Decorador para la versión mínima cuando se ejecuta la prueba en FreeBSD. Si la versión de FreeBSD es inferior al mínimo, aumente `unittest.SkipTest`.

`@test.support.requires_linux_version (*min_version)`

Decorador para la versión mínima cuando se ejecuta la prueba en Linux. Si la versión de Linux es inferior al mínimo, aumente `unittest.SkipTest`.

`@test.support.requires_mac_version (*min_version)`

Decorador para la versión mínima cuando se ejecuta la prueba en Mac OS X. Si la versión MAC OS X es menor que la mínima, genere `unittest.SkipTest`.

`@test.support.requires_ieee_754`

Decorador para omitir pruebas en plataformas que no son *IEEE 754*.

`@test.support.requires_zlib`

Decorador para omitir pruebas si `zlib` no existe.

`@test.support.requires_gzip`

Decorador para omitir pruebas si `gzip` no existe.

`@test.support.requires_bz2`

Decorador para omitir pruebas si `bz2` no existe.

`@test.support.requires_lzma`

Decorador para omitir pruebas si `lzma` no existe.

`@test.support.requires_resource (resource)`

Decorador para omitir pruebas si `resource` no está disponible.

`@test.support.requires_docstrings`

Decorador para ejecutar solo la prueba si `HAVE_DOCSTRINGS`.

`@test.support.cpython_only (test)`

Decorador para pruebas solo aplicable a CPython.

`@test.support.impl_detail (msg=None, **guards)`

Decorador para invocar `check_impl_detail()` en *guards*. Si eso retorna `False`, entonces usa `msg` como la razón para omitir la prueba.

`@test.support.no_tracing (func)`

Decorador para desactivar temporalmente el seguimiento durante la duración de la prueba.

`@test.support.refcount_test (test)`

Decorador para pruebas que implican conteo de referencias. El decorador no ejecuta la prueba si CPython no la ejecuta. Cualquier función de rastreo no se establece durante la duración de la prueba para evitar conteos de referencia (*refcounts*) inesperados causados por la función de rastreo.

`@test.support.reap_threads` (*func*)

Decorador para garantizar que los hilos se limpien incluso si la prueba falla.

`@test.support.bigmemtest` (*size, memuse, dry_run=True*)

Decorador para pruebas *bigmem*.

size es un tamaño solicitado para la prueba (en unidades arbitrarias interpretadas por la prueba). *memuse* es el número de bytes por unidad para la prueba, o una buena estimación de la misma. Por ejemplo, una prueba que necesita dos búfers de byte, de 4 GiB cada uno, podría decorarse con `@bigmemtest(size=_4G, memuse=2)`.

El argumento *size* normalmente se pasa al método de prueba decorado como un argumento adicional. Si *dry_run* es `True`, el valor pasado al método de prueba puede ser menor que el valor solicitado. Si *dry_run* es `False`, significa que la prueba no admite ejecuciones ficticias cuando no se especifica `-M`.

`@test.support.bigaddrspacetest` (*f*)

Decorador para pruebas que llenan el espacio de direcciones. *f* es la función para envolver.

`test.support.make_bad_fd` ()

Se crea un descriptor de archivo no válido abriendo y cerrando un archivo temporal y retornando su descriptor.

`test.support.check_syntax_error` (*testcase, statement, errtext="", *, lineno=None, offset=None*)

Prueba los errores de sintaxis en *statement* intentando compilar *statement*. *testcase* es la instancia `unittest` para la prueba. *errtext* es la expresión regular que debe coincidir con la representación de cadena de caracteres que es lanzada en `SyntaxError`. Si *lineno* no es `None`, se compara con la línea de la excepción. Si *offset* no es `None`, se compara con el desplazamiento de la excepción.

`test.support.check_syntax_warning` (*testcase, statement, errtext="", *, lineno=1, offset=None*)

Prueba la advertencia de sintaxis en *statement* intentando compilar *statement*. Pruebe también que `SyntaxWarning` se emite solo una vez, y que se convertirá en `SyntaxError` cuando se convierta en error. *testcase* es la instancia `unittest` para la prueba. *errtext* es la expresión regular que debe coincidir con la representación de cadena del emitido `SyntaxWarning` y lanza `SyntaxError`. Si *lineno* no es `None`, se compara con la línea de advertencia y excepción. Si *offset* no es `None`, se compara con el desplazamiento de la excepción.

Nuevo en la versión 3.8.

`test.support.open_urlresource` (*url, *args, **kw*)

Abre *url*. Si la apertura falla, se lanza `TestFailed`.

`test.support.import_module` (*name, deprecated=False, *, required_on()*)

Esta función importa y retorna el módulo nombrado. A diferencia de una importación normal, esta función lanza `unittest.SkipTest` si el módulo no se puede importar.

Los mensajes de deprecación de módulos y paquetes se suprimen durante esta importación si *deprecated* es `True`. Si se requiere un módulo en una plataforma pero es opcional para otros, establezca *required_on* en un iterable de prefijos de plataforma que se compararán con `sys.platform`.

Nuevo en la versión 3.1.

`test.support.import_fresh_module` (*name, fresh=(), blocked=(), deprecated=False*)

Esta función importa y retorna una copia nueva del módulo Python nombrado eliminando el módulo nombrado de `sys.modules` antes de realizar la importación. Tenga en cuenta que a diferencia de `reload()`, el módulo original no se ve afectado por esta operación.

fresh es un iterable de nombres de módulos adicionales que también se eliminan del caché `sys.modules` antes de realizar la importación.

bloqueado es un iterable de nombres de módulos que se reemplazan con `None` en la memoria caché del módulo durante la importación para garantizar que los intentos de importarlos generen `ImportError`.

El módulo nombrado y los módulos nombrados en los parámetros *fresh* y *bloqueado* se guardan antes de comenzar la importación y luego se vuelven a insertar en `sys.modules` cuando se completa la importación fresca.

Los mensajes de deprecación de módulos y paquetes se suprimen durante esta importación si *deprecated* es `True`.

Esta función lanzará `ImportError` si el módulo nombrado no puede importarse.

Ejemplo de uso:

```
# 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'])
```

Nuevo en la versión 3.1.

`test.support.modules_setup()`
Retorna una copia de `sys.modules`.

`test.support.modules_cleanup(oldmodules)`
Elimina los módulos a excepción de `oldmodules` y encodings para preservar la memoria caché interna.

`test.support.threading_setup()`
Retorna el recuento del hilo actual y copia de subprocessos colgantes.

`test.support.threading_cleanup(*original_values)`
Se limpia los hilos no especificados en `original_values`. Diseñado para emitir una advertencia si una prueba deja hilos en ejecución en segundo plano.

`test.support.join_thread(thread, timeout=30.0)`
Se une un `thread` dentro de `timeout`. Lanza un `AssertionError` si el hilo sigue vivo después de unos segundos de `timeout`.

`test.support.reap_children()`
Se utiliza esto al final de `test_main` siempre que se inicien subprocessos. Esto ayudará a garantizar que ningún proceso hijo adicional (zombies) se quede para acumular recursos y crear problemas al buscar `refleaks`.

`test.support.get_attribute(obj, name)`
Obtiene un atributo, lanzando `unittest.SkipTest` si `AttributeError` está activado.

`test.support.bind_port(sock, host=HOST)`
Se enlaza el `socket` a un puerto libre y retorna el número de puerto. Se basa en puertos efímeros para garantizar que estemos utilizando un puerto independiente. Esto es importante ya que muchas pruebas pueden ejecutarse simultáneamente, especialmente en un entorno *buildbot*. Este método lanza una excepción si `sock.family` es `AF_INET` y `sock.type` es `SOCK_STREAM`, y el `socket` tiene `SO_REUSEADDR` o `SO_REUSEPORT` establecido en él. Las pruebas nunca deben configurar estas opciones de `socket` para los `sockets TCP/IP`. El único caso para configurar estas opciones es probar la multidifusión (*multicasting*) a través de múltiples `sockets UDP`.

Además, si la opción de `socket` `SO_EXCLUSIVEADDRUSE` está disponible (es decir, en Windows), se establecerá en el `socket`. Esto evitará que otras personas se vinculen a nuestro `host/puerto` mientras dure la prueba.

`test.support.bind_unix_socket(sock, addr)`
Enlace un `socket` Unix, lanzando `unittest.SkipTest` si `PermissionError` es lanzado.

`test.support.catch_threading_exception()`
El administrador de contexto captura `threading.Thread` excepción usando `threading.excepthook()`.

Atributos establecidos cuando se detecta una excepción:

- `exc_type`
- `exc_value`
- `exc_traceback`
- `thread`

Consulte la documentación para `threading.excepthook()`.

Estos atributos se eliminan en la salida del administrador de contexto.

Uso:

```
with support.catch_threading_exception() as cm:
    # code spawning a thread which raises an exception
    ...

    # check the thread exception, use cm attributes:
    # exc_type, exc_value, exc_traceback, thread
    ...

# exc_type, exc_value, exc_traceback, thread attributes of cm no longer
# exists at this point
# (to avoid reference cycles)
```

Nuevo en la versión 3.8.

`test.support.catch_unraisable_exception()`

El administrador de contexto detectando excepciones imposibles de evaluar usando `sys.unraisablehook()`.

El almacenamiento del valor de excepción (`cm.unraisable.exc_value`) crea un ciclo de referencia. El ciclo de referencia se interrumpe explícitamente cuando sale el administrador de contexto.

El almacenamiento del objeto (`cm.unraisable.object`) puede resucitarlo si se establece en un objeto que se está finalizando. Salir del administrador de contexto borra el objeto almacenado.

Uso:

```
with support.catch_unraisable_exception() as cm:
    # code creating an "unraisable exception"
    ...

    # check the unraisable exception: use cm.unraisable
    ...

# cm.unraisable attribute no longer exists at this point
# (to break a reference cycle)
```

Nuevo en la versión 3.8.

`test.support.find_unused_port(family=socket.AF_INET, socktype=socket.SOCK_STREAM)`

Retorna un puerto no utilizado que debería ser adecuado para el enlace. Esto se logra creando un *socket* temporal con la misma familia y tipo que el parámetro *sock* (el valor predeterminado es `AF_INET`, `SOCK_STREAM`) y vinculándolo a la dirección de host especificada (por defecto es `0.0.0.0`) con el puerto establecido en 0, provocando un puerto efímero no utilizado del sistema operativo. El *socket* temporal se cierra y se elimina, y se retorna el puerto efímero.

Este método o `bind_port()` debe usarse para cualquier prueba en la que un *socket* del servidor deba estar vinculado a un puerto en particular durante la duración de la prueba. Cuál usar depende de si el código de llamada está creando un *socket* Python, o si un puerto no utilizado debe proporcionarse en un constructor o pasar a un programa externo (es decir, el argumento `-accept` al modo *s_server de openssl*). Siempre es preferible `bind_port()` sobre `find_unused_port()` donde sea posible. Se desaconseja el uso de un puerto codificado ya que puede hacer que varias instancias de la prueba sean imposibles de ejecutar simultáneamente, lo cual es un problema para los *buildbots*.

`test.support.load_package_tests(pkg_dir, loader, standard_tests, pattern)`

La implementación genérica del protocolo `unittest` `load_tests` para usar en paquetes de prueba. `pkg_dir` es el directorio raíz del paquete; `loader`, `standard_tests` y `pattern` son los argumentos esperados por `load_tests`. En casos simples, el paquete de prueba `__init__`. Py puede ser el siguiente:

```
import os
from test.support import load_package_tests
```

(continué en la próxima página)

(proviene de la página anterior)

```
def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

`test.support.fs_is_case_insensitive(directory)`

Retorna True si el sistema de archivos para *directory* no distingue entre mayúsculas y minúsculas.

`test.support.detect_api_mismatch(ref_api, other_api, *, ignore=())`

Retorna el conjunto de atributos, funciones o métodos de *ref_api* que no se encuentra en *other_api* *, *excepto por una lista definida de elementos que se ignorarán en esta comprobación especificada en *ignore*.

De forma predeterminada, omite los atributos privados que comienzan con “_” pero incluye todos los métodos mágicos, es decir, los que comienzan y terminan en “__”.

Nuevo en la versión 3.5.

`test.support.patch(test_instance, object_to_patch, attr_name, new_value)`

Se anula *object_to_patch.attr_name* con *new_value*. Se agrega también el procedimiento de limpieza a *test_instance* para restaurar *object_to_patch* para *attr_name*. *Attr_name* debe ser un atributo válido para *object_to_patch*.

`test.support.run_in_subinterp(code)`

Ejecuta *code* en el subinterpretador. Lanza `unittest.SkipTest` si `tracemalloc` está habilitado.

`test.support.check_free_after_iterating(test, iter, cls, args=())`

Aserciona que *iter* se desasigna después de iterar.

`test.support.missing_compiler_executable(cmd_names=[])`

Verifica la existencia de los ejecutables del compilador cuyos nombres figuran en *cmd_names* o todos los ejecutables del compilador cuando *cmd_names* está vacío y retorna el primer ejecutable faltante o None cuando no se encuentra ninguno.

`test.support.check__all__(test_case, module, name_of_module=None, extra=(), blacklist=())`

Aserciona que la variable `__all__` de *module* contiene todos los nombres públicos.

Los nombres públicos del módulo (su API) se detectan automáticamente en función de si coinciden con la convención de nombres públicos y se definieron en *module*.

El argumento *name_of_module* puede especificar (como una cadena o tupla del mismo) qué módulo(s) se podría definir una API para ser detectada como una API pública. Un caso para esto es cuando *module* importa parte de su API pública desde otros módulos, posiblemente un *backend de C* (como `csv` y su `_csv`).

El argumento *extra* puede ser un conjunto de nombres que de otro modo no se detectarían automáticamente como «públicos» («*public*»), como objetos sin un atributo adecuado `__module__`. Si se proporciona, se agregará a los detectados automáticamente.

El argumento *blacklist* puede ser un conjunto de nombres que no deben tratarse como parte de la API pública aunque sus nombres indiquen lo contrario.

Ejemplo de uso:

```
import bar
import foo
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        blacklist = {'baz'} # Undocumented name.
```

(continué en la próxima página)

(proviene de la página anterior)

```
# bar imports part of its API from _bar.
support.check__all__(self, bar, ('bar', '_bar'),
                       extra=extra, blacklist=blacklist)
```

Nuevo en la versión 3.6.

`test.support.adjust_int_max_str_digits(max_digits)`

This function returns a context manager that will change the global `sys.set_int_max_str_digits()` setting for the duration of the context to allow execution of test code that needs a different limit on the number of digits when converting between an integer and string.

Nuevo en la versión 3.8.14.

El módulo `test.support` define las siguientes clases:

class `test.support.TransientResource` (*exc*, ***kwargs*)

Las instancias son un administrador de contexto que lanza `ResourceDenied` si se lanza el tipo de excepción especificado. Cualquier argumento de palabra clave se trata como pares de atributo / valor para compararlo con cualquier excepción lanzada dentro de `with`. Solo si todos los pares coinciden correctamente con los atributos de la excepción se lanza `ResourceDenied`.

class `test.support.EnvironmentVarGuard`

Clase utilizada para establecer o deshabilitar temporalmente las variables de entorno. Las instancias se pueden usar como un administrador de contexto y tienen una interfaz de diccionario completa para consultar/modificar `os.environ` subyacente. Después de salir del administrador de contexto, todos los cambios en las variables de entorno realizados a través de esta instancia se revertirán.

Distinto en la versión 3.1: Añadido una interfaz de diccionario.

`EnvironmentVarGuard.set(envvar, value)`

Se establece temporalmente la variable de entorno `envvar` en el valor de `value`.

`EnvironmentVarGuard.unset(envvar)`

Deshabilita temporalmente la variable de entorno `envvar`.

class `test.support.SuppressCrashReport`

Un administrador de contexto suele intentar evitar ventanas emergentes de diálogo de bloqueo en las pruebas que se espera que bloqueen un subproceso.

En Windows, deshabilita los cuadros de diálogo de Informe de errores de Windows usando `SetErrorMode`.

En UNIX, `resource.setrlimit()` se usa para establecer `resource.RLIMIT_CORE` del límite flexible a 0 para evitar la creación de archivos `coredump`.

En ambas plataformas, el valor anterior se restaura mediante `__exit__()`.

class `test.support.CleanImport` (**module_names*)

Un administrador de contexto para forzar la importación para que retorne una nueva referencia de módulo. Esto es útil para probar comportamientos a nivel de módulo, como la emisión de una Advertencia de desaprobación en la importación. Ejemplo de uso:

```
with CleanImport('foo'):
    importlib.import_module('foo') # New reference.
```

class `test.support.DirsOnSysPath` (**paths*)

Un administrador de contexto para agregar temporalmente directorios a `sys.path`.

Esto hace una copia de `sys.path`, agrega cualquier directorio dado como argumento posicional, luego re-verte `sys.path` a la configuración copiada cuando finaliza el contexto.

Tenga en cuenta que *all* `sys.path` produce modificaciones en el cuerpo del administrador de contexto, incluida la sustitución del objeto, se revertirán al final del bloque.

class `test.support.SaveSignals`

Clase para guardar y restaurar manejadores (*handlers*) de señal registrados por el manejador de señal Python.

class `test.support.Matcher`

matches (*self*, *d*, ****kwargs**)

Intenta hacer coincidir una sola sentencia con los argumentos proporcionados.

match_value (*self*, *k*, *dv*, *v*)

Intenta hacer coincidir un único valor almacenado (*dv*) con un valor proporcionado (*v*).

class `test.support.WarningsRecorder`

La clase utilizada para registrar advertencias para pruebas unitarias. Consulte la documentación de `check_warnings()` arriba para obtener más detalles.

class `test.support.BasicTestRunner`

run (*test*)

Retorna *test* y retorna el resultado..

class `test.support.TestHandler` (`logging.handlers.BufferingHandler`)

Clase para soporte de *logging*.

class `test.support.FakePath` (*path*)

Simple *path-like object*. Se implementa el método `__fspath__()` que simplemente retorna el argumento *path*. Si *path* es una excepción, se lanzará en `__fspath__()`.

26.10 `test.support.script_helper` —Utilidades para las pruebas de ejecución de Python

El módulo `test.support.script_helper` proporciona soporte para las pruebas de ejecución de script de Python.

`test.support.script_helper.interpreter_requires_environment()`

Retorna *True* si el `sys.executable` interpreter requiere variables de entorno para poder ejecutarse.

Esto está diseñado para usarse con `@unittest.skipIf()` para anotar pruebas que necesitan usar una función `assert_python*` para iniciar un modo aislado (`-I`) o sin entorno proceso de subinterpretador de modo (`-E`).

Una compilación y prueba normal no se encuentra en esta situación, pero puede suceder cuando se intenta ejecutar el conjunto de pruebas de biblioteca estándar desde un intérprete que no tiene un directorio principal obvio con la lógica de búsqueda de directorio principal actual de Python.

La configuración `PYTHONHOME` es una forma de hacer que la mayoría del *testuite* se ejecute en esa situación. `PYTHONPATH` o `PYTHONUSERSTEST` son otras variables de entorno comunes que pueden afectar si el intérprete puede o no comenzar.

`test.support.script_helper.run_python_until_end(*args, **env_vars)`

Configura el entorno basado en *env_vars* para ejecutar el intérprete en un subprocesso. Los valores pueden incluir `__isolated`, `__cleanenv`, `__cwd` y `TERM`.

`test.support.script_helper.assert_python_ok(*args, **env_vars)`

Aserción de que ejecutar el intérprete con *arg* y variables de entorno opcionales *env_vars* tiene éxito (`rc == 0`) y retorna una tupla (código de retorno, `stdout`, `stderr`).

Si se establece la palabra clave `__cleanenv`, *env_vars* se usa como un entorno nuevo.

Python se inicia en modo aislado (opción de línea de comando `-I`), excepto si la palabra clave `__isolated` se establece en `False`.

`test.support.script_helper.assert_python_failure(*args, **env_vars)`

Aserciona que la ejecución del intérprete con *args* y variables de entorno opcionales *env_vars* falla (`rc != 0`) y retorna una tupla (`return code`, `stdout`, `stderr`).

Consulte `assert_python_ok()` para más opciones.

```
test.support.script_helper.spawn_python(*args, stdout=subprocess.PIPE,
                                         stderr=subprocess.STDOUT, **kw)
```

Ejecuta un subprocesso de Python con los argumentos dados.

kw es un argumento adicional de palabras clave para pasar a `subprocess.Popen()`. Retorna un objeto a `subprocess.Popen`.

```
test.support.script_helper.kill_python(p)
```

Ejecuta el proceso dado `subprocess.Popen` hasta que finalice y retorne *stdout*.

```
test.support.script_helper.make_script(script_dir, script_basename, source,
                                       omit_suffix=False)
```

Crea un script que contiene *source* en la ruta *script_dir* y *script_basename*. Si *omit_suffix* es `False`, agregue `.py` al nombre. Retorna la ruta completa del script.

```
test.support.script_helper.make_zip_script(zip_dir, zip_basename, script_name,
                                           name_in_zip=None)
```

Crea un archivo zip en *zip_dir* y *zip_basename* con la extensión `zip` que contiene los archivos en *script_name*. *name_in_zip* es el nombre del archivo. Retorna una tupla que contiene (ruta completa, ruta completa del nombre del archivo).

```
test.support.script_helper.make_pkg(pkg_dir, init_source="")
```

Crea un directorio llamado *pkg_dir* que contiene un archivo `__init__` con *init_source* como su contenido.

```
test.support.script_helper.make_zip_pkg(zip_dir, zip_basename, pkg_name,
                                         script_basename, source, depth=1, compiled=False)
```

Crea un directorio de paquete zip con una ruta de *zip_dir* y *zip_basename* que contiene un archivo `__init__` vacío y un archivo *script_basename* que contiene el *source*. Si *compilado* es `True`, ambos archivos fuente se compilarán y se agregarán al paquete zip. Retorna una tupla de la ruta zip completa y el nombre de archivo para el archivo zip.

Consulta también el modo de desarrollo de Python: la opción `-X dev` y la variable de entorno `PYTHONDEVMODE`.

Depuración y perfilado

Estas bibliotecas le ayudan con el desarrollo de Python: el depurador le permite recorrer paso a paso el código, analizar marcos de pila y establecer puntos de interrupción, etc., y los perfiladores ejecutan código y le proporcionan un desglose detallado de los tiempos de ejecución, lo que le permite identificar cuellos de botella en sus programas. Los eventos de auditoría proporcionan visibilidad de los comportamientos en tiempo de ejecución que, de lo contrario, requerirían depuración o parches intrusivos.

27.1 Tabla de auditoría de eventos

This table contains all events raised by `sys.audit()` or `PySys_Audit()` calls throughout the CPython runtime and the standard library. These calls were added in 3.8.0 or later (see [PEP 578](#)).

Ver `sys.addaudithook()` y `PySys_AddAuditHook()` para informarse en el manejo de estos eventos.

CPython implementation detail: Esta tabla es generada desde la documentación de CPython, y no debe guardar eventos lanzados por otras implementaciones. Ver la documentación de tus especificaciones en tiempo de ejecución para los eventos lanzados recientemente.

Audit event	Arguments
<code>array.__new__</code>	<code>typecode, initializer</code>
<code>builtins.breakpoint</code>	<code>breakpointhook</code>
<code>builtins.id</code>	<code>id</code>
<code>builtins.input</code>	<code>prompt</code>
<code>builtins.input/result</code>	<code>result</code>
<code>code.__new__</code>	<code>code, filename, name, argcount, posonlyargcount, kwnonlyargcount, nloc</code>
<code>compile</code>	<code>source, filename</code>
<code>cpython.PyInterpreterState_Clear</code>	
<code>cpython.PyInterpreterState_New</code>	
<code>cpython._PySys_ClearAuditHooks</code>	
<code>cpython.run_command</code>	<code>command</code>
<code>cpython.run_file</code>	<code>filename</code>
<code>cpython.run_interactivehook</code>	<code>hook</code>
<code>cpython.run_module</code>	<code>module-name</code>
<code>cpython.run_startup</code>	<code>filename</code>
<code>cpython.run_stdin</code>	

Tabla 1 – proviene de la página anterior

Audit event	Arguments
ctypes.addressof	obj
ctypes.call_function	func_pointer, arguments
ctypes.cdata	address
ctypes.cdata/buffer	pointer, size, offset
ctypes.create_string_buffer	init, size
ctypes.create_unicode_buffer	init, size
ctypes.dlopen	name
ctypes.dlsym	library, name
ctypes.dlsym/handle	handle, name
ctypes.get_errno	
ctypes.get_last_error	
ctypes.seh_exception	code
ctypes.set_errno	errno
ctypes.set_last_error	error
ctypes.string_at	address, size
ctypes.wstring_at	address, size
ensurepip.bootstrap	root
exec	code_object
fcntl.fcntl	fd, cmd, arg
fcntl.flock	fd, operation
fcntl.ioctl	fd, request, arg
fcntl.lockf	fd, cmd, len, start, whence
ftplib.connect	self, host, port
ftplib.sendcmd	self, cmd
function.__new__	code
gc.get_objects	generation
gc.get_referents	objs
gc.get_referrers	objs
glob.glob	pathname, recursive
imaplib.open	self, host, port
imaplib.send	self, data
import	module, filename, sys.path, sys.meta_path, sys.path_hooks
mmap.__new__	fileno, length, access, offset
msvcrt.get_osfhandle	fd
msvcrt.locking	fd, mode, nbytes
msvcrt.open_osfhandle	handle, flags
nntplib.connect	self, host, port
nntplib.putline	self, line
object.__delattr__	obj, name
object.__getattr__	obj, name
object.__setattr__	obj, name, value
open	file, mode, flags
os.add_dll_directory	path
os.chdir	path
os.chflags	path, flags
os.chmod	path, mode, dir_fd
os.chown	path, uid, gid, dir_fd
os.exec	path, args, env
os.fork	
os.forkpty	
os.getxattr	path, attribute
os.kill	pid, sig
os.killpg	pgid, sig

Tabla 1 – proviene de la página anterior

Audit event	Arguments
os.link	src, dst, src_dir_fd, dst_dir_fd
os.listdir	path
os.listdirattr	path
os.lockf	fd, cmd, len
os.mkdir	path, mode, dir_fd
os.posix_spawn	path, argv, env
os.putenv	key, value
os.remove	path, dir_fd
os.removexattr	path, attribute
os.rename	src, dst, src_dir_fd, dst_dir_fd
os.rmdir	path, dir_fd
os.scandir	path
os.setxattr	path, attribute, value, flags
os.spawn	mode, path, args, env
os.startfile	path, operation
os.symlink	src, dst, dir_fd
os.system	command
os.truncate	fd, length
os.unsetenv	key
os.utime	path, times, ns, dir_fd
pdb.Pdb	
pickle.find_class	module, name
poplib.connect	self, host, port
poplib.putline	self, line
pty.spawn	argv
resource.prlimit	pid, resource, limits
resource.setrlimit	resource, limits
setopencodehook	
shutil.chown	path, user, group
shutil.copyfile	src, dst
shutil.copymode	src, dst
shutil.copystat	src, dst
shutil.copytree	src, dst
shutil.make_archive	base_name, format, root_dir, base_dir
shutil.move	src, dst
shutil.rmtree	path
shutil.unpack_archive	filename, extract_dir, format
signal.pthread_kill	thread_id, signalnum
smtplib.connect	self, host, port
smtplib.send	self, data
socket.__new__	self, family, type, protocol
socket.bind	self, address
socket.connect	self, address
socket.getaddrinfo	host, port, family, type, protocol
socket.gethostbyaddr	ip_address
socket.gethostbyname	hostname
socket.gethostname	
socket.getnameinfo	sockaddr
socket.getservbyname	servicename, protocolname
socket.getservbyport	port, protocolname
socket.sendmsg	self, address
socket.sendto	self, address
socket.sethostname	name

Tabla 1 – proviene de la página anterior

Audit event	Arguments
sqlite3.connect	database
subprocess.Popen	executable, args, cwd, env
sys._current_frames	
sys._getframe	
sys.addaudithook	
sys.excepthook	hook, type, value, traceback
sys.set_asyncgen_hooks_finalizer	
sys.set_asyncgen_hooks_firstiter	
sys.setprofile	
sys.settrace	
sys.unraisablehook	hook, unraisable
syslog.closelog	
syslog.openlog	ident, logoption, facility
syslog.setlogmask	maskpri
syslog.syslog	priority, message
telnetlib.Telnet.open	self, host, port
telnetlib.Telnet.write	self, buffer
tempfile.mkdtemp	fullpath
tempfile.mkstemp	fullpath
urllib.Request	fullurl, data, headers, method
webbrowser.open	url
winreg.ConnectRegistry	computer_name, key
winreg.CreateKey	key, sub_key, access
winreg.DeleteKey	key, sub_key, access
winreg.DeleteValue	key, value
winreg.DisableReflectionKey	key
winreg.EnableReflectionKey	key
winreg.EnumKey	key, index
winreg.EnumValue	key, index
winreg.ExpandEnvironmentStrings	str
winreg.LoadKey	key, sub_key, file_name
winreg.OpenKey	key, sub_key, access
winreg.OpenKey/result	key
winreg.PyHKEY.Detach	key
winreg.QueryInfoKey	key
winreg.QueryReflectionKey	key
winreg.QueryValue	key, sub_key, value_name
winreg.SaveKey	key, file_name
winreg.SetValue	key, sub_key, type, value

The following events are raised internally and do not correspond to any public API of CPython:

Audit event	Arguments
<code>_winapi.CreateFile</code>	<code>file_name, desired_access, share_mode, creation_disposition, flags_and_attributes</code>
<code>_wina-pi.CreateJunction</code>	<code>src_path, dst_path</code>
<code>_wina-pi.CreateNamedPipe</code>	<code>name, open_mode, pipe_mode</code>
<code>_winapi.CreatePipe</code>	
<code>_wina-pi.CreateProcess</code>	<code>application_name, command_line, current_directory</code>
<code>_wina-pi.OpenProcess</code>	<code>process_id, desired_access</code>
<code>_wina-pi.TerminateProcess</code>	<code>handle, exit_code</code>
<code>ctypes.PyObj_FromPtr</code>	<code>obj</code>

27.2 bdb — Framework de depuración

Source code: [Lib/bdb.py](#)

El módulo `bdb` maneja las funciones básicas del depurador, como establecer puntos de interrupción o gestionar la ejecución a través del mismo.

La siguiente excepción es definida:

exception `bdb.BdbQuit`

Excepción lanzada por la clase `Bdb` al salir del depurador.

El módulo `bdb` también define dos clases:

class `bdb.Breakpoint` (*self, file, line, temporary=0, cond=None, funcname=None*)

Esta clase implementa puntos de interrupción temporales, permitiendo ignorar recuentos, desactivar y (re-)activar puntos de interrupción y definir condiciones de interrupción para los mismos.

Los puntos de interrupción se indexan numéricamente mediante una lista llamada `bpynumber` y por pares (`archivo`, `línea`) mediante `bplist`. En el primer caso se apunta a una única instancia de `Breakpoint`. En el segundo caso se apunta a una lista de tales instancias, ya que puede haber más de un punto de interrupción por línea.

Al crear un punto de interrupción, el nombre del archivo asociado debe estar en su forma canónica. Si se define un nombre de función `funcname`, se contará el punto de interrupción cuando se ejecute la primera línea de esa función. Los puntos de interrupción condicionales siempre cuentan los alcances.

Las instancias de la clase `Breakpoint` tienen los siguientes métodos:

deleteMe ()

Elimina el punto de interrupción de la lista asociada a un archivo/línea. Si es el último punto de interrupción en esa posición, también borra la entrada correspondiente del archivo/línea correspondiente.

enable ()

Marca el punto de interrupción como habilitado.

disable ()

Marca el punto de interrupción como deshabilitado.

bpformat ()

Retorna una cadena de caracteres que contiene toda la información sobre el punto de interrupción, apropiadamente formateada:

- El número del punto de interrupción.
- Si es temporal o no.
- Su archivo, la posición de la línea.
- La condición que causa la interrupción.
- Si debe ignorarse las próximas N veces.
- El recuento de alcances del punto de interrupción.

Nuevo en la versión 3.2.

bpprint (*out=None*)

Imprime la salida de *bpformat()* en el archivo *out*, o en la salida estándar si es *None*.

class `bdb.Bdb` (*skip=None*)

La clase *Bdb* actúa como clase base del depurador genérico de Python.

Esta clase se encarga de los detalles de la funcionalidad de seguimiento; una clase derivada debería encargarse de implementar la interacción con el usuario. Un ejemplo es la clase de depuración estándar (*pdb.Pdb*).

El argumento *skip*, si se proporciona, debe ser un iterable con patrones de nombre de archivo, al estilo del módulo *glob*. El depurador no entrará en aquellos marcos de ejecución que se originen en un módulo que coincida con uno de estos patrones. Para determinar si un marco de ejecución se origina en un módulo determinado, se hace uso de `__name__` en las variables globales del marco de ejecución dado.

Nuevo en la versión 3.1: El argumento *skip*.

Los siguientes métodos de la clase *Bdb* normalmente no necesitan ser redefinidos.

canonic (*filename*)

Método auxiliar para obtener el nombre del archivo en su forma canónica, es decir, como una ruta absoluta normalizada entre mayúsculas y minúsculas (en sistemas de archivos que no distinguen entre ambas) y despojada de los corchetes angulares circundantes.

reset ()

Establece los atributos *botframe*, *stopframe*, *returnframe* y *quitting* con valores preparados para comenzar la depuración.

trace_dispatch (*frame, event, arg*)

Esta función se instala como función de seguimiento de los marcos de ejecución depurados. Su valor de retorno es la nueva función de seguimiento a usar (en la mayoría de los casos, ella misma).

La implementación predeterminada decide cómo despachar un marco de ejecución, dependiendo del tipo de evento (pasado como una cadena) que está a punto de ejecutarse. *event* puede tomar uno de los siguientes valores:

- "line": Se va a ejecutar una nueva línea de código.
- "call": Está a punto de llamarse a una función o se ha entrado en otro bloque de código.
- "return": Una función u otro bloque de código está a punto de retornar.
- "exception": Ha ocurrido una excepción.
- "c_call": Una función de C está a punto de llamarse .
- "c_return": Una función de C ha retornado.
- "c_exception": Una función de C ha lanzado una excepción.

Para los eventos de Python, son llamadas funciones especializadas (ver más abajo). En cambio, para los eventos de C no se realiza ninguna acción.

El parámetro *arg* depende del evento previo.

Consulta la documentación de *sys.settrace()* para obtener más información sobre la función de seguimiento. Para obtener más información sobre los objetos código y los objetos marco consulte *types*.

dispatch_line (*frame*)

Si el depurador tiene que detenerse en la línea actual, invoca el método `user_line()` (que debe ser redefinido en las subclases). Genera una excepción `BdbQuit` si se establece el flag `Bdb.quitting` (que se puede establecer mediante `user_line()`). Retorna una referencia al método `trace_dispatch()` para realizar un seguimiento adicional en ese ámbito.

dispatch_call (*frame*, *arg*)

Si el depurador tiene que detenerse en esta llamada de función, invoca el método `user_call()` (que debe ser redefinido en las subclases). Lanza una excepción `BdbQuit` si se establece el flag `Bdb.quitting` (que se puede establecer mediante `user_call()`). Retorna una referencia al método `trace_dispatch()` para realizar un seguimiento adicional en ese ámbito.

dispatch_return (*frame*, *arg*)

Si el depurador tiene que detenerse en el retorno de esta función, invoca el método `user_return()` (que debe ser redefinido en las subclases). Lanza una excepción `BdbQuit` si se establece el flag `Bdb.quitting` (que se puede establecer mediante `user_return()`). Retorna una referencia al método `trace_dispatch()` para realizar un seguimiento adicional en ese ámbito.

dispatch_exception (*frame*, *arg*)

Si el depurador tiene que detenerse en esta excepción, invoca el método `user_exception()` (que debe ser redefinido en las subclases). Lanza una excepción `BdbQuit` si se establece el flag `Bdb.quitting` (que se puede establecer mediante `user_exception()`). Retorna una referencia al método `trace_dispatch()` para realizar un seguimiento adicional en ese ámbito.

Las clases derivadas normalmente no necesitan redefinir los siguientes métodos, pero pueden hacerlo si necesitan redefinir la definición de parada y los puntos de interrupción.

stop_here (*frame*)

Este método comprueba si el *frame* está en cualquier posición debajo de `botframe` en la pila de llamadas. `botframe` es el marco de ejecución en el que comenzó la depuración.

break_here (*frame*)

Este método comprueba si hay un punto de interrupción en el nombre de archivo y la línea pertenecientes a *frame* o, al menos, en la función actual. Si el punto de interrupción es temporal, este método lo elimina.

break_anywhere (*frame*)

Este método comprueba si hay un punto de interrupción en el nombre de archivo del marco de ejecución actual.

Las clases derivadas deben redefinir estos métodos para adquirir el control sobre las operaciones de depuración.

user_call (*frame*, *argument_list*)

Este método se llama desde `dispatch_call()` cuando existe la posibilidad de que sea necesaria una interrupción en cualquier lugar dentro de la función llamada.

user_line (*frame*)

Este método se llama desde `dispatch_line()` cuando `stop_here()` o `break_here()` generan `True`.

user_return (*frame*, *return_value*)

Este método se llama desde `dispatch_return()` cuando `stop_here()` genera `True`.

user_exception (*frame*, *exc_info*)

Este método se llama desde `dispatch_exception()` cuando `stop_here()` genera `True`.

do_clear (*arg*)

Maneja cómo un punto de interrupción debe ser eliminado cuando es temporal.

Este método debe ser implementado por las clases derivadas.

Las clases derivadas y los clientes pueden llamar a los siguientes métodos para influir en el estado de transición.

set_step ()

Se detiene después de una línea de código.

set_next (*frame*)

Se detiene en la siguiente línea del marco de ejecución dado o en la de uno inferior.

set_return (*frame*)

Se detiene cuando se retorna desde el marco de ejecución dado.

set_until (*frame*)

Se detiene cuando se alcanza un número de línea superior al de la línea actual o se retorna desde el marco de ejecución actual.

set_trace ([*frame*])

Inicia la depuración desde el marco de ejecución *frame*. Si *frame* no se especifica, la depuración se inicia desde el marco de ejecución que produce la llamada.

set_continue ()

Se detiene solo en los puntos de interrupción o cuando haya terminado. Si no hay puntos de interrupción, se configura la función de seguimiento del sistema en `None`.

set_quit ()

Establece el atributo `quitting` en `True`. Esto lanza una excepción `BdbQuit` en la siguiente llamada a uno de los métodos `dispatch_*()` que tenga lugar.

Las clases derivadas y los clientes pueden llamar a los siguientes métodos para manipular los puntos de interrupción. Estos métodos retornan una cadena de caracteres que contiene un mensaje de error si algo fue mal, o `None` si todo fue correctamente.

set_break (*filename*, *lineno*, *temporary=0*, *cond*, *funcname*)

Establece un nuevo punto de interrupción. Si la línea en la posición *lineno* no existe en el archivo con nombre *filename* pasado como argumento, se retorna un mensaje de error. *filename* debe estar en su forma canónica, tal como se describe en el método `canonic()`.

clear_break (*filename*, *lineno*)

Elimina el punto de interrupción en el archivo *filename* y número de línea *lineno*. Si no se ha establecido ninguno, se retorna un mensaje de error.

clear_bpbynumber (*arg*)

Elimina el punto de interrupción que tiene el índice *arg* en `Breakpoint.bpbynumber`. Si *arg* no es un valor numérico o es un índice fuera de rango, se retorna un mensaje de error.

clear_all_file_breaks (*filename*)

Elimina todos los puntos de interrupción en el archivo *filename*. Si no se ha establecido ninguno, se retorna un mensaje de error.

clear_all_breaks ()

Elimina todos los puntos de interrupción que existen.

get_bpbynumber (*arg*)

Retorna un punto de interrupción especificado por el número dado. Si *arg* es una cadena de caracteres, será convertida en un número. Si *arg* es una cadena no numérica, o el punto de interrupción dado nunca existió o ya ha sido eliminado, se lanza una excepción `ValueError`.

Nuevo en la versión 3.2.

get_break (*filename*, *lineno*)

Comprueba si hay un punto de interrupción en la línea número *lineno* del archivo *filename*.

get_breaks (*filename*, *lineno*)

Retorna todos los puntos de interrupción en la línea número *lineno* del archivo *filename*, o una lista vacía si no se ha establecido ninguno.

get_file_breaks (*filename*)

Retorna todos los puntos de interrupción en el archivo *filename*, o una lista vacía si no hay ninguno establecido.

get_all_breaks ()

Retorna todos los puntos de interrupción establecidos.

Las clases derivadas y clientes pueden llamar los siguientes métodos para obtener una estructura de datos que representa un seguimiento de pila.

get_stack (*f*, *t*)

Obtiene una lista de registros para un marco de ejecución y todos los marcos de ejecución superiores (que han ocasionado la llamadas previas) e inferiores, además del tamaño de la parte superior.

format_stack_entry (*frame_lineno*, *lprefix*=':')

Retorna una cadena con información sobre una entrada de pila, identificada por una tupla de la forma (cuadro de ejecución, número de línea):

- La forma canónica del nombre del archivo que contiene el marco de ejecución.
- El nombre de la función o "<lambda>".
- Los argumentos de entrada.
- El valor de retorno.
- La línea de código (si existe).

Los dos métodos descritos a continuación pueden ser llamados por los clientes para usar un depurador que se encargue de depurar un *statement*, proporcionado como una cadena de caracteres.

run (*cmd*, *globals*=None, *locals*=None)

Depura una sentencia ejecutada a través de la función `exec()`. *globals* por defecto es `__main__`. `dict__`, mientras que *locals* es *globals* por defecto.

runeval (*expr*, *globals*=None, *locals*=None)

Depura una expresión ejecutada mediante la función `eval()`. *globals* y *locals* tienen el mismo significado que en `run()`.

runctx (*cmd*, *globals*, *locals*)

Definido solo por compatibilidad con versiones anteriores. Llama al método `run()`.

runcall (*func*, **args*, ***kws*)

Depura una única llamada a función y retorna su resultado.

Por último, el módulo también define las siguientes funciones:

`bdb.checkfuncname` (*b*, *frame*)

Comprueba si se debería interrumpir la ejecución en éste punto, dependiendo de la forma en que se estableció el punto de interrupción *b*.

Si se estableció usando el número de línea, verifica si `b.line` es el mismo que el del marco de ejecución que también se pasó como argumento. Si el punto de interrupción se estableció mediante el nombre de la función, tenemos que comprobar que estamos en el cuadro de ejecución correcto (la función correcta) y si estamos en su primera línea ejecutable.

`bdb.effective` (*file*, *line*, *frame*)

Determina si hay un punto de interrupción efectivo (activo) en esta línea de código. Retorna una tupla con el punto de interrupción y un valor booleano que indica si es correcto eliminar un punto de interrupción temporal. Retorna (None, None) si no hay un punto de interrupción coincidente.

`bdb.set_trace` ()

Inicia la depuración usando una instancia de `Bdb`, partiendo desde el marco de ejecución que realiza la llamada.

27.3 `faulthandler` — Volcar el rastreo de Python

Nuevo en la versión 3.3.

Este módulo contiene funciones para volcar los rastreos de Python explícitamente, en un fallo, después de un tiempo de espera o en una señal del usuario. Llame a `faulthandler.enable()` para instalar los gestores de fallos para las señales `SIGSEGV`, `SIGFPE`, `SIGABRT`, `SIGBUS`, y `SIGILL`. También puede activarlos al inicio estableciendo la variable de entorno `PYTHONFAULTHANDLER` o usando la opción de línea de comandos `-X faulthandler`.

El gestor de fallos es compatible con el gestor de fallos del sistema como Apport o el gestor de fallos de Windows. El módulo utiliza una pila alternativa para los gestores de señales si la función `sigaltstack()` está disponible. Esto le permite volcar el rastreo incluso en un desbordamiento de pila.

El gestor de fallos se llama en casos catastróficos y, por lo tanto, solo puede utilizar funciones seguras en señales (por ejemplo, no puede asignar memoria en el *heap*). Debido a esta limitación, el volcado del rastreo es mínimo comparado a los rastreos normales de Python:

- Solo se soporta ASCII. El gestor de errores `backslashreplace` se utiliza en la codificación.
- Cada cadena de caracteres está limitada a 500 caracteres.
- Solo se muestran el nombre de archivo, el nombre de la función y el número de línea. (sin código fuente)
- Está limitado a 100 *frames* y 100 hilos.
- El orden se invierte: la llamada más reciente se muestra primero.

Por defecto, el rastreo de Python se escribe en `sys.stderr`. Para ver los rastreos, las aplicaciones deben ejecutarse en la terminal. Alternativamente se puede pasar un archivo de registro a `faulthandler.enable()`.

El módulo está implementado en C, así los rastreos se pueden volcar en un fallo o cuando Python está en bloqueo mutuo.

27.3.1 Volcar el rastreo

`faulthandler.dump_traceback(file=sys.stderr, all_threads=True)`

Vuelca los rastreos de todos los hilos en el archivo *file*. Si *all_threads* es `False`, vuelca solo el hilo actual.

Distinto en la versión 3.5: Se añadió soporte para pasar el descriptor de archivo a esta función.

27.3.2 Estado del gestor de fallos

`faulthandler.enable(file=sys.stderr, all_threads=True)`

Activa el gestor de fallos: instala gestores para las señales `SIGSEGV`, `SIGFPE`, `SIGABRT`, `SIGBUS` y `SIGILL` para volcar el rastreo de Python. Si *all_threads* es `True`, produce rastreos por cada hilo activo. De lo contrario, vuelca solo el hilo actual.

El archivo *file* se debe mantener abierto hasta que se desactive el gestor de fallos: ver [problema con descriptors de archivo](#).

Distinto en la versión 3.5: Se añadió soporte para pasar el descriptor de archivo a esta función.

Distinto en la versión 3.6: En Windows, también se instaló un gestor para la excepción de Windows.

`faulthandler.disable()`

Desactiva el gestor de fallos: desinstala los gestores de señales instalados por `enable()`.

`faulthandler.is_enabled()`

Comprueba si el gestor de fallos está activado.

27.3.3 Volcar los rastreos después de un tiempo de espera

`faulthandler.dump_traceback_later` (*timeout*, *repeat=False*, *file=sys.stderr*, *exit=False*)

Vuelca los rastreos de todos los hilos, después de un tiempo de espera de *timeout* segundos, o cada *timeout* segundos si *repeat* es `True`. Si *exit* es `True`, llama a `_exit()` con `status=1` después de volcar los rastreos. (Nota: `_exit()` termina el proceso inmediatamente, lo que significa que no hace ninguna limpieza como vaciar los buffers de archivos.) Si la función se llama dos veces, la nueva llamada reemplaza los parámetros previos y reinicia el tiempo de espera. El temporizador tiene una resolución de menos de un segundo.

El archivo *file* se debe mantener abierto hasta que se vuelque el rastreo o se llame a `cancel_dump_traceback_later()`: ver [problema con descriptores de archivo](#).

Esta función está implementada utilizando un hilo vigilante.

Distinto en la versión 3.7: Ahora esta función está siempre disponible.

Distinto en la versión 3.5: Se añadió soporte para pasar el descriptor de archivo a esta función.

`faulthandler.cancel_dump_traceback_later()`

Cancela la última llamada a `dump_traceback_later()`.

27.3.4 Volcar el rastreo en una señal del usuario

`faulthandler.register` (*signum*, *file=sys.stderr*, *all_threads=True*, *chain=False*)

Registra una señal del usuario: instala un gestor para la señal *signum* para volcar el rastreo de todos los hilos, o del hilo actual si *all_threads* es `False`, en el archivo *file*. Llama al gestor previo si *chain* es `True`.

El archivo *file* se debe mantener abierto hasta que la señal sea anulada por `unregister()`: ver [problema con descriptores de archivo](#).

No está disponible en Windows.

Distinto en la versión 3.5: Se añadió soporte para pasar el descriptor de archivo a esta función.

`faulthandler.unregister` (*signum*)

Anula una señal del usuario: desinstala el gestor de la señal *signum* instalada por `register()`. Retorna `True` si la señal fue registrada, `False` en otro caso.

No está disponible en Windows.

27.3.5 Problema con descriptores de archivo

`enable()`, `dump_traceback_later()` y `register()` guardan el descriptor de archivo de su argumento *file*. Si se cierra el archivo y su descriptor de archivo es reutilizado por un nuevo archivo, o si se usa `os.dup2()` para reemplazar el descriptor de archivo, el rastreo se escribirá en un archivo diferente. Llame a estas funciones nuevamente cada vez que se reemplace el archivo.

27.3.6 Ejemplo

Ejemplo de un fallo de segmentación en Linux con y sin activar el gestor de fallos:

```
$ python3 -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python3 -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700 (most recent call first):
```

(continué en la próxima página)

(proviene de la página anterior)

```
File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
File "<stdin>", line 1 in <module>
Segmentation fault
```

27.4 pdb — El Depurador de Python

Código fuente: [Lib/unittest/mock.py](#)

El módulo `pdb` define un depurador de código fuente interactivo para programas Python. Soporta el establecimiento de puntos de ruptura (condicionales) y pasos sencillos a nivel de línea de código fuente, inspección de marcos de pila, listado de código fuente, y evaluación de código Python arbitrario en el contexto de cualquier marco de pila. También soporta depuración post-mortem y puede ser llamado bajo control del programa.

El depurador es extensible – en realidad se define como la clase `Pdb`. Esto no está actualmente documentado pero se entiende fácilmente leyendo la fuente. La interfaz de extensión usa los módulos `bdb` y `cmd`.

El mensaje del depurador es `(Pdb)`. El uso típico para ejecutar un programa bajo el control del depurador es:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

Distinto en la versión 3.3: La finalización de tabulación a través del módulo `readline` está disponible para comandos y argumentos de comando, por ejemplo. Los nombres globales y locales actuales se ofrecen como argumentos del comando `p`.

`pdb.py` también puede ser invocado como un script para depurar otros scripts. Por ejemplo:

```
python3 -m pdb myscript.py
```

Cuando se invoca como script, `pdb` entrará automáticamente en la depuración post-mortem si el programa que se depura sale de forma anormal. Después de la depuración post-mortem (o después de la salida normal del programa), `pdb` reiniciará el programa. El reinicio automático preserva el estado de `pdb` (como los puntos de ruptura) y en la mayoría de los casos es más útil que abandonar el depurador al salir del programa.

Nuevo en la versión 3.2: `pdb.py` ahora acepta una opción `-c` que ejecuta comandos como si se dieran en un archivo `.pdbrc`, ver [Comandos del depurador](#).

Nuevo en la versión 3.7: `pdb.py` ahora acepta una opción `-m` que ejecuta módulos similares a los que `python3 -m` hace. Como con un script, el depurador detendrá la ejecución justo antes de la primera línea del módulo.

El uso típico para entrar en el depurador de un programa en ejecución es insertar:

```
import pdb; pdb.set_trace()
```

En el lugar donde quieres ingresar en el depurador. Entonces puedes pasar a través del código siguiendo esta declaración, y continuar corriendo sin el depurador usando el comando `continue`.

Nuevo en la versión 3.7: La `breakpoint()` incorporada cuando se llama con los valores por defecto, puede ser usado en lugar del `import pdb; pdb.set_trace()`.

El uso típico para inspeccionar un programa que se ha estrellado es:

```

>>> 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)

```

El módulo define las siguientes funciones; cada una de ellas entra en el depurador de una manera ligeramente diferente:

`pdb.run(statement, globals=None, locals=None)`

Ejecutar el *statement* (dado como una cadena o un objeto de código) bajo el control del depurador. El prompt del depurador aparece antes de que se ejecute cualquier código; puedes establecer puntos de ruptura y escribir *continue*, o puedes pasar por la declaración usando *step* o *next* (todos estos comandos se explican más abajo). Los argumentos opcionales *globals* y *locals* especifican el entorno en el que se ejecuta el código; por defecto se utiliza el diccionario del módulo `__main__`. (Ver la explicación de las funciones incorporadas `exec()` o `eval()`).

`pdb.runeval(expression, globals=None, locals=None)`

Evalúa la *expression* (dada como una cadena o un objeto de código) bajo el control del depurador. Cuando vuelve `runeval()`, retorna el valor de la expresión. En caso contrario, esta función es similar a `run()`.

`pdb.runcall(function, *args, **kwargs)`

Llama a la *function* (un objeto de función o método, no una cadena) con los argumentos dados. Cuando `runcall()` retorna, retorna lo que sea que la llamada de la función haya retornado. El aviso del depurador aparece tan pronto como se introduce la función.

`pdb.set_trace(*, header=None)`

Entra en el depurador en el marco de la pila de llamadas. Esto es útil para codificar duramente un punto de interrupción en un punto dado de un programa, incluso si el código no se depura de otra manera (por ejemplo, cuando una afirmación falla). Si se da, se imprime el *header* en la consola justo antes de que comience la depuración.

Distinto en la versión 3.7: El argumento de la palabra clave *header*.

`pdb.post_mortem(traceback=None)`

Ingresa a la depuración post-mortem del objeto *traceback* dado. Si no se da un *traceback*, utiliza el de la excepción que se está manejando actualmente (una excepción debe ser manejada si se va a utilizar el pre-terminado).

`pdb.pm()`

Ingresa a la depuración post-mortem de la traza encontrada en `sys.last_traceback`.

Las funciones `run*` y `set_trace()` son alias para instanciar la clase `Pdb` y llamar al método del mismo nombre. Si quieres acceder a más funciones, tienes que hacerlo tú mismo:

`class pdb.Pdb(completekey='tab', stdin=None, stdout=None, skip=None, nosigint=False, readrc=True)`

`Pdb` es la clase de depuración.

Los argumentos *completekey*, *stdin* y *stdout* se pasan a la subyacente `cmd.Cmd` class; ver la descripción allí.

El argumento *skip*, si se da, debe ser un iterable de los patrones de nombre de los módulos de estilo global. El depurador no entrará en marcos que se originen en un módulo que coincida con uno de estos patrones.¹

¹ Si se considera que un marco se origina en un determinado módulo se determina por el `__name__` en el marcos globales.

Por defecto, Pdb establece un manejador para la señal de SIGINT (que se envía cuando el usuario presiona `Ctrl-C` en la consola) cuando da un comando de `continue`. Esto te permite entrar en el depurador de nuevo presionando `Ctrl-C`. Si quieres que Pdb no toque el manejador de SIGINT, pon `nosigint` en `true`.

El argumento `* readrc *` por defecto es verdadero y controla si Pdb cargará archivos `.pdbrc` desde el sistema de archivos.

Ejemplo de llamada para permitir el rastreo con `skip`:

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

Levanta una `auditing event` `pdb.Pdb` sin argumentos.

Nuevo en la versión 3.1: El argumento `skip`.

Nuevo en la versión 3.2: El argumento del `nosigint`. Anteriormente, un manejador SIGINT nunca fue establecido por Pdb.

Distinto en la versión 3.6: El argumento `readrc`.

```
run(statement, globals=None, locals=None)
runeval(expression, globals=None, locals=None)
runcall(function, *args, **kwargs)
set_trace()
```

Véase la documentación para las funciones explicadas anteriormente.

27.4.1 Comandos del depurador

Los comandos reconocidos por el depurador se enumeran a continuación. La mayoría de los comandos pueden abreviarse a una o dos letras como se indica; por ejemplo, `h` (`elp`) significa que se puede usar `h` o `help` para introducir el comando de ayuda (pero no `he` o `hel`, ni `H` o `Help` o `HELP`). Los argumentos de los comandos deben estar separados por espacios en blanco (espacios o tabulaciones). Los argumentos opcionales están encerrados entre corchetes (`[]`) en la sintaxis del comando; los corchetes no deben escribirse. Las alternativas en la sintaxis de los comandos están separadas por una barra vertical (`|`).

Introducir una línea en blanco repite el último comando introducido. Excepción: si el último comando fue un `list` comando, las siguientes 11 líneas están listadas.

Se supone que los comandos que el depurador no reconoce son declaraciones en Python y se ejecutan en el contexto del programa que se está depurando. Las declaraciones en Python también pueden ser precedidas por un signo de exclamación (`!`). Esta es una manera poderosa de inspeccionar el programa que se está depurando; incluso es posible cambiar una variable o llamar a una función. Cuando se produce una excepción en una sentencia de este tipo, se imprime el nombre de la excepción pero no se cambia el estado del depurador.

El depurador soporta `aliases`. Los alias pueden tener parámetros que permiten un cierto nivel de adaptabilidad al contexto que se está examinando.

Se pueden introducir múltiples comandos en una sola línea, separados por `;;`. (No se usa un solo `;` ya que es el separador de múltiples comandos en una línea que se pasa al analizador de Python). No se aplica ninguna inteligencia para separar los comandos; la entrada se divide en el primer par `;;`, incluso si está en medio de una cadena citada.

Si un archivo `.pdbrc` existe en el directorio principal del usuario o en el directorio actual, se lee y se ejecuta como si se hubiera escrito en el prompt del depurador. Esto es particularmente útil para los alias. Si ambos archivos existen, el del directorio principal se lee primero y los alias definidos allí pueden ser anulados por el archivo local.

Distinto en la versión 3.2: `.pdbrc` puede ahora contener comandos que continúan depurando, como `continue` o `next`. Anteriormente, estos comandos no tenían ningún efecto.

h(elp) [`command`]

Sin argumento, imprime la lista de comandos disponibles. Con un `command` como argumento, imprime la ayuda sobre ese comando. `help pdb` muestra la documentación completa (el docstring del módulo `pdb`). Como el argumento `command` debe ser un identificador, hay que introducir `help exec` para obtener ayuda sobre el comando `!`.

w (here)

Imprime un rastro de la pila (*stack trace*), con el marco más reciente en la parte inferior. Una flecha indica el marco actual, que determina el contexto de la mayoría de los comandos.

d (own) [*count*]

Mueve los niveles del marco actual *count* (por defecto uno) hacia abajo en el trazado de la pila (*stack trace*) (a un marco más nuevo).

u (p) [*count*]

Mueve el marco actual *count* (por defecto uno) niveles hacia arriba en el trazado de la pila (*stack trace*) (a un marco más antiguo).

b (reak) [[([*filename*:]*lineno* | *function*) [, *condition*]]

Con un argumento *lineno*, establece una ruptura allí en el archivo actual. Con un argumento *function*, establece una ruptura en la primera declaración ejecutable dentro de esa función. El número de línea puede ir precedido de un nombre de archivo y dos puntos, para especificar un punto de interrupción en otro archivo (probablemente uno que aún no se haya cargado). El archivo se busca en *sys.path*. Observe que a cada punto de interrupción se le asigna un número al que se refieren todos los demás comandos de puntos de interrupción.

Si un segundo argumento está presente, es una expresión que debe ser evaluada como verdadera antes de que el punto de ruptura sea honrado.

Sin argumento, enumere todas las interrupciones, incluyendo para cada punto de interrupción, el número de veces que se ha alcanzado ese punto de interrupción, el conteo de ignorancia actual y la condición asociada, si la hay.

tbreak [[([*filename*:]*lineno* | *function*) [, *condition*]]

Punto de interrupción temporal, que se elimina automáticamente cuando se ejecuta por primera vez. Los argumentos son los mismos que para *break*.

cl (ear) [*filename:lineno* | *bpnumber* [*bpnumber* ...]]

Con el argumento *filename:lineno*, despeja todos los puntos de interrupción en esta línea. Con una lista de números de puntos de interrupción separados por espacios, despeja esos puntos de interrupción. Sin el argumento, despeja todos los puntos de interrupción (pero primero pide confirmación).

disable [*bpnumber* [*bpnumber* ...]]

Deshabilitar los puntos de ruptura interrupción como una lista de números de puntos de ruptura separados por espacios. Desactivar un punto de interrupción significa que no puede hacer que el programa detenga la ejecución, pero a diferencia de borrar un punto de interrupción, permanece en la lista de puntos de interrupción y puede ser (re)activado.

enable [*bpnumber* [*bpnumber* ...]]

Habilitar los puntos de interrupción especificados.

ignore *bpnumber* [*count*]

Establece el conteo de ignorar el número de punto de interrupción dado. Si se omite el recuento, el recuento de ignorar se establece en 0. Un punto de interrupción se activa cuando el recuento de ignorar es cero. Cuando no es cero, el conteo se decrementa cada vez que se alcanza el punto de ruptura y el punto de ruptura no se desactiva y cualquier condición asociada se evalúa como verdadera.

condition *bpnumber* [*condition*]

Establece una nueva *condition* para el punto de interrupción, una expresión que debe evaluarse como verdadera antes de que el punto de ruptura sea honrado. Si la condición está ausente, se elimina cualquier condición existente, es decir, el punto de ruptura se hace incondicional.

commands [*bpnumber*]

Especifique una lista de comandos para el número del punto de interrupción *bpnumber*. Los comandos mismos aparecen en las siguientes líneas. Escriba una línea que contenga sólo *end* para terminar los comandos. Un ejemplo:

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

Para eliminar todos los comandos de un punto de interrupción, escribe `commands` y sigue inmediatamente con `end`, es decir, no des órdenes.

Sin el argumento `bnumber`, `commands` se refiere al último punto de interrupción establecido.

Puede utilizar los comandos de punto de interrupción para iniciar el programa de nuevo. Simplemente usa el comando `continue`, o `step`, o cualquier otro comando que reanude la ejecución.

Al especificar cualquier comando que reanude la ejecución (actualmente `continue`, `step`, `next`, `return`, `jump`, `quit` y sus abreviaturas) se termina la lista de comandos (como si ese comando fuera inmediatamente seguido de `end`.) Esto se debe a que cada vez que se reanuda la ejecución (incluso con un simple siguiente o paso), puede encontrarse con otro punto de interrupción, que podría tener su propia lista de comandos, lo que conduce a ambigüedades sobre qué lista ejecutar.

Si utiliza el comando “silent” en la lista de comandos, no se imprime el mensaje habitual sobre la parada en un punto de interrupción. Esto puede ser deseable para los puntos de interrupción que deben imprimir un mensaje específico y luego continuar. Si ninguno de los otros comandos imprime nada, no se ve ninguna señal de que se haya alcanzado el punto de interrupción.

s (tep)

Ejecutar la línea actual, detenerse en la primera ocasión posible (ya sea en una función que se llame o en la siguiente línea de la función actual).

n (ext)

Continúe la ejecución hasta que se alcance la siguiente línea de la función actual o vuelva. (La diferencia entre `next` y `step` es que `step` se detiene dentro de una función llamada, mientras que `next` ejecuta las funciones llamadas a (casi) toda velocidad, deteniéndose sólo en la siguiente línea de la función actual).

unt (il) [lineno]

Sin argumento, continúe la ejecución hasta que se alcance la línea con un número mayor que el actual.

Con un número de línea, continúe la ejecución hasta que se alcance una línea con un número mayor o igual a ese. En ambos casos, también se detiene cuando vuelve la trama actual.

Distinto en la versión 3.2: Permita dar un número de línea explícito.

r (eturn)

Continúe la ejecución hasta que vuelva la función actual.

c (ont (inue))

Continúa la ejecución, sólo se detiene cuando se encuentra un punto de ruptura.

j (ump) lineno

Establezca la siguiente línea que será ejecutada. Sólo disponible en el marco de más bajo. Esto te permite saltar hacia atrás y ejecutar el código de nuevo, o saltar hacia adelante para saltar el código que no quieres ejecutar.

Cabe señalar que no todos los saltos están permitidos – por ejemplo, no es posible saltar en medio de un bucle `for` o fuera de una cláusula `finally`.

l (ist) [first[, last]]

Enumere el código fuente del archivo actual. Sin argumentos, enumere 11 líneas alrededor de la línea actual o continúe la lista anterior. Con `.` como argumento, enumere 11 líneas alrededor de la línea actual. Con un argumento, enumere 11 líneas alrededor de esa línea. Con dos argumentos, enumere el rango dado; si el segundo argumento es menor que el primero, se interpreta como un conteo.

La línea actual en el cuadro actual se indica con `->`. Si se está depurando una excepción, la línea donde la excepción fue originalmente planteada o propagada se indica con `>>`, si difiere de la línea actual.

Nuevo en la versión 3.2: El marcador `>>`.

ll | longlist

Enumere todos los códigos fuente de la función o marco actual. Las líneas interesantes están marcadas como `list`.

Nuevo en la versión 3.2.

a (rgs)

Imprime la lista de argumentos de la función actual.

p *expression*

Evalúa la *expression* en el contexto actual e imprime su valor.

Nota: `print()` también se puede usar, pero no es un comando de depuración — esto ejecuta la función Python `print()`.

pp *expression*

Como el comando *p*, excepto que el valor de la expresión se imprime bastante usando el módulo `pprint`.

whatis *expression*

Imprime el tipo de la *expression*.

source *expression*

Intenta obtener el código fuente del objeto dado y mostrarlo.

Nuevo en la versión 3.2.

display [*expression*]

Muestra el valor de la expresión si ha cambiado, cada vez que se detenga la ejecución en el marco actual.

Sin expresión, enumere todas las expresiones de visualización para el cuadro actual.

Nuevo en la versión 3.2.

undisplay [*expression*]

No muestren más la expresión en el cuadro actual. Sin la expresión, borre todas las expresiones de la pantalla para el marco actual.

Nuevo en la versión 3.2.

interact

Inicie un intérprete interactivo (usando el módulo `code`) cuyo espacio de nombres global contiene todos los nombres (globales y locales) que se encuentran en el ámbito actual.

Nuevo en la versión 3.2.

alias [*name* [*command*]]

Crear un alias llamado *name* que ejecute el *command*. El comando no debe estar entre comillas. Los parámetros reemplazables pueden ser indicados por `%1`, `%2`, y así sucesivamente, mientras que `%*` es reemplazado por todos los parámetros. Si no se da ningún comando, se muestra el alias actual de *name*. Si no se dan argumentos, se muestran todos los alias.

Los alias pueden anidarse y pueden contener cualquier cosa que se pueda teclear legalmente en el prompt de `pdb`. Tenga en cuenta que los comandos internos de `pdb` pueden ser anulados por los alias. Dicho comando se oculta hasta que se elimina el alias. El alias se aplica de forma recursiva a la primera palabra de la línea de comandos; todas las demás palabras de la línea se dejan en paz.

Como ejemplo, aquí hay dos alias útiles (especialmente cuando se colocan en el archivo `.pdbrc`):

```
# 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*

Elimine el alias especificado.

! *statement*

Ejecute (una línea) *statement* en el contexto del marco de la pila actual. El signo de exclamación puede ser omitido a menos que la primera palabra de la declaración se parezca a un comando de depuración. Para establecer una variable global, puede anteponer al comando de asignación una declaración `global` en la misma línea, por ejemplo:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...]

restart [args ...]

Reinicie el programa Python depurado. Si se suministra un argumento, se divide con `shlex` y el resultado se utiliza como el nuevo `sys.argv`. Se conservan el historial, los puntos de interrupción, las acciones y las opciones del depurador. `restart` es un alias de `run`.

q(uit)

Salga del depurador. El programa que se está ejecutando fue abortado.

debug code

Introduce un depurador recursivo que pasa por el argumento del código (que es una expresión o declaración arbitraria que debe ejecutarse en el entorno actual).

retval

Print the return value for the last return of a function.

Notas de pie de página

27.5 Los perfiladores de Python

Código fuente: `Lib/profile.py` y `Lib/pstats.py`

27.5.1 Introducción a los perfiladores

`cProfile` y `profile` proporcionan *deterministic profiling* de los programas de Python. `profile` es un conjunto de estadísticas que describe con qué frecuencia y durante cuánto tiempo se ejecutaron varias partes del programa. Estas estadísticas pueden formatearse en informes a través del módulo `pstats`.

La biblioteca estándar de Python proporciona dos implementaciones diferentes de la misma interfaz de creación de perfiles:

1. `cProfile` se recomienda para la mayoría de los usuarios; Es una extensión C con una sobrecarga razonable que la hace adecuada para perfilar programas de larga duración. Basado en `lsprof`, aportado por Brett Rosen y Ted Czotter.
2. `profile`, un módulo Python puro cuya interfaz es imitada por `cProfile`, pero que agrega una sobrecarga significativa a los programas perfilados. Si está intentando extender el generador de perfiles de alguna manera, la tarea podría ser más fácil con este módulo. Originalmente diseñado y escrito por Jim Roskind.

Nota: Los módulos del generador de perfiles están diseñados para proporcionar un perfil de ejecución para un programa determinado, no para fines de evaluación comparativa (para eso, está `timeit` que permite obtener resultados razonablemente precisos). Esto se aplica particularmente a la evaluación comparativa del código Python contra el código C: los perfiladores introducen gastos generales para el código Python, pero no para las funciones de nivel C, por lo que el código C parecería más rápido que cualquier Python.

27.5.2 Manual instantáneo de usuario

Esta sección se proporciona a los usuarios que «no quieren leer el manual». Proporciona una visión general muy breve y permite a un usuario realizar perfiles rápidamente en una aplicación existente.

Para perfilar una función que toma un solo argumento, puede hacer:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(Use `profile` en lugar de `cProfile` si este último no está disponible en su sistema.)

La acción anterior ejecutaría `re.compile()` e imprime resultados de perfil como los siguientes:

```
197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name
```

ncalls	totttime	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)

La primera línea indica que se monitorearon 197 llamadas. De esas llamadas, 192 fueron *primitive*, lo que significa que la llamada no fue inducida por recursividad. La siguiente línea: `Ordered by: standard name`, indica que la cadena de texto en la columna del extremo derecho se utilizó para ordenar la salida. Los encabezados de columna incluyen:

ncalls por el número de llamadas.

totttime para el tiempo total empleado en la función dada (y excluyendo el tiempo realizado en llamadas a subfunciones)

percall es el cociente de `totttime` dividido por `ncalls`

cumtime es el tiempo acumulado empleado en esta y todas las subfunciones (desde la invocación hasta la salida). Esta cifra es precisa *incluso* para funciones recursivas.

percall es el cociente de `cumtime` dividido por llamadas primitivas

filename:lineno(función) proporciona los datos respectivos de cada función

Cuando hay dos números en la primera columna (por ejemplo, 3/1), significa que la función se repite. El segundo valor es el número de llamadas primitivas y el primero es el número total de llamadas. Tenga en cuenta que cuando la función no se repite, estos dos valores son iguales y solo se imprime la figura.

En lugar de imprimir la salida al final de la ejecución del perfil, puede guardar los resultados en un archivo especificando un nombre de archivo en la función `run()`:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

La clase `pstats.Stats` lee los resultados del perfil desde un archivo y los formatea de varias maneras.

Los archivos `cProfile` y `profile` también se pueden invocar como un script para perfilar otro script. Por ejemplo:

```
python -m cProfile [-o output_file] [-s sort_order] (-m module | myscript.py)
```

- o escribe los resultados del perfil en un archivo en lugar de stdout
- s especifica uno de los valores de clasificación `sort_stats()` para ordenar la salida. Esto solo se aplica cuando
- o no se suministra.
- m especifica que se está perfilando un módulo en lugar de un script.

Nuevo en la versión 3.7: Se agregó la opción `–m` a `cProfile`.

Nuevo en la versión 3.8: Se agregó la opción `–m` a `profile`.

Los módulos `pstats` clase `Stats` tienen una variedad de métodos para manipular e imprimir los datos guardados en un archivo de resultados de perfil:

```
import pstats
from pstats import SortKey
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

El método `strip_dirs()` eliminó la ruta extraña de todos los nombres de módulos. El método `sort_stats()` clasificó todas las entradas de acuerdo con el módulo/línea/nombre de cadena de caracteres estándar que se imprime. El método `print_stats()` imprimió todas las estadísticas. Puede intentar las siguientes llamadas de clasificación:

```
p.sort_stats(SortKey.NAME)
p.print_stats()
```

La primera llamada realmente ordenará la lista por nombre de función, y la segunda llamada imprimirá las estadísticas. Las siguientes son algunas llamadas interesantes para experimentar:

```
p.sort_stats(SortKey.CUMULATIVE).print_stats(10)
```

Esto ordena el perfil por tiempo acumulado en una función y luego solo imprime las diez líneas más significativas. Si desea comprender qué algoritmos están tomando tiempo, la línea anterior es lo que usaría.

Si estuviera buscando ver qué funciones se repetían mucho y toman mucho tiempo, usted haría:

```
p.sort_stats(SortKey.TIME).print_stats(10)
```

para ordenar según el tiempo dedicado a cada función y luego imprimir las estadísticas de las diez funciones principales.

También puedes probar:

```
p.sort_stats(SortKey.FILENAME).print_stats('__init__')
```

Esto ordenará todas las estadísticas por nombre de archivo y luego imprimirá estadísticas solo para los métodos de inicio de clase (ya que están escritos con `__init__` en ellos). Como último ejemplo, puedes probar:

```
p.sort_stats(SortKey.TIME, SortKey.CUMULATIVE).print_stats(.5, 'init')
```

Esta línea clasifica las estadísticas con una clave primaria de tiempo y una clave secundaria de tiempo acumulado, y luego imprime algunas de las estadísticas. Para ser específicos, la lista primero se reduce al 50% (re: `.5`) de su tamaño original, luego solo se mantienen las líneas que contienen `init` y se imprime esa sub-sublista.

Si se preguntó qué funciones denominaban las funciones anteriores, ahora podría (p todavía está ordenada según el último criterio) hacer:

```
p.print_callers(.5, 'init')
```

y obtendría una lista de llamadas para cada una de las funciones enumeradas.

Si desea más funcionalidad, tendrá que leer el manual o adivinar lo que hacen las siguientes funciones:

```
p.print_callees()
p.add('restats')
```

Invocado como un script, el módulo `pstats` es un navegador de estadísticas para leer y examinar volcados de perfil. Tiene una interfaz simple orientada a la línea de comandos (implementada usando `cmd`) y ayuda interactiva.

27.5.3 Referencia del módulo `profile` y `cProfile`

Los módulos `profile` y `cProfile` proporcionan las siguientes funciones:

`profile.run(command, filename=None, sort=-1)`

Esta función toma un único argumento que se puede pasar a la función `exec()` y un nombre de archivo opcional. En todos los casos esta rutina ejecuta:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

y recopila estadísticas de perfiles de la ejecución. Si no hay ningún nombre de archivo, esta función crea automáticamente una instancia de `Stats` e imprime un informe de perfil simple. Si se especifica el valor de clasificación, se pasa a esta instancia `Stats` para controlar cómo se ordenan los resultados.

`profile.runctx(command, globals, locals, filename=None, sort=-1)`

Esta función es similar a `run()`, con argumentos agregados para proporcionar los diccionarios globales y locales para la cadena de caracteres `command`. Esta rutina ejecuta:

```
exec(command, globals, locals)
```

y recopila estadísticas de perfiles como en la función `run()` anterior.

class `profile.Profile(timer=None, timeunit=0.0, subcalls=True, builtins=True)`

Esta clase normalmente solo es usada si se necesita un control más preciso sobre la creación de perfiles que el que proporciona la función `cProfile.run()`.

Se puede suministrar un temporizador personalizado para medir cuánto tiempo tarda el código en ejecutarse mediante el argumento `timer`. Esta debe ser una función que retorna un solo número que representa la hora actual. Si el número es un entero, la `timeunit` especifica un multiplicador que especifica la duración de cada unidad de tiempo. Por ejemplo, si el temporizador retorna tiempos medidos en miles de segundos, la unidad de tiempo sería `.001`.

El uso directo de la clase `Profile` permite formatear los resultados del perfil sin escribir los datos del perfil en un archivo:

```
import cProfile, pstats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

La clase `Profile` también se puede usar como administrador de contexto (solo se admite en el módulo `cProfile`. Ver *Tipos Gestores de Contexto*):

```
import cProfile

with cProfile.Profile() as pr:
    # ... do something ...

pr.print_stats()
```

Distinto en la versión 3.8: Se agregó soporte de administrador de contexto.

enable()Comienza a recopilar datos de perfiles. Solo en *cProfile*.**disable()**Deja de recopilar datos de perfiles. Solo en *cProfile*.**create_stats()**

Deja de recopilar datos de perfiles y registre los resultados internamente como el perfil actual.

print_stats(sort=-1)Crea un objeto *Stats* en función del perfil actual e imprima los resultados en *stdout*.**dump_stats(filename)**Escribe los resultados del perfil actual en *filename*.**run(cmd)**Perfila el cmd a través de *exec()*.**runctx(cmd, globals, locals)**Perfila el cmd a través de *exec()* con el entorno global y local especificado.**runcall(func, *args, **kwargs)**Perfila *func(*args, **kwargs)*

Tenga en cuenta que la creación de perfiles solo funcionará si el comando/función llamado realmente regresa. Si el intérprete se termina (por ejemplo, a través de una llamada a *sys.exit()* durante la ejecución del comando/función llamado) no se imprimirán resultados de generación de perfiles.

27.5.4 La clase Stats

El análisis de los datos del generador de perfiles es realizado utilizando la clase *Stats*.

class *pstats.Stats* (*filenames or profile, stream=sys.stdout)

Este constructor de clase crea una instancia de un «objeto de estadísticas» a partir de un *filename* (o una lista de nombres de archivo) o de una instancia *Profile*. La salida se imprimirá en la secuencia especificada por *stream*.

El archivo seleccionado por el constructor anterior debe haber sido creado por la versión correspondiente de *profile* o *cProfile*. Para ser específicos, *no* hay compatibilidad de archivos garantizada con futuras versiones de este generador de perfiles, y no hay compatibilidad con archivos producidos por otros perfiladores, o el mismo generador de perfiles ejecutado en un sistema operativo diferente. Si se proporcionan varios archivos, se combinarán todas las estadísticas para funciones idénticas, de modo que se pueda considerar una vista general de varios procesos en un solo informe. Si es necesario combinar archivos adicionales con datos en un objeto existente *Stats*, se puede usar el método *add()*.

En lugar de leer los datos del perfil de un archivo, se puede usar un objeto *cProfile.Profile* o *profile.Profile* como fuente de datos del perfil.

los objetos *Stats* tienen los siguientes métodos:

strip_dirs()

Este método para la clase *Stats* elimina toda la información de ruta principal de los nombres de archivo. Es muy útil para reducir el tamaño de la impresión para que quepa en (cerca de) 80 columnas. Este método modifica el objeto y se pierde la información eliminada. Después de realizar una operación de tira, se considera que el objeto tiene sus entradas en un orden «aleatorio», como lo fue justo después de la inicialización y carga del objeto. Si *strip_dirs()* hace que dos nombres de funciones no se puedan distinguir (están en la misma línea del mismo nombre de archivo y tienen el mismo nombre de función), entonces las estadísticas para estas dos entradas se acumulan en un sola entrada.

add(*filenames)

Este método de la clase *Stats* acumula información de perfil adicional en el objeto de perfil actual. Sus argumentos deben referirse a los nombres de archivo creados por la versión correspondiente de *profile.run()* o *cProfile.run()*. Las estadísticas para funciones con nombre idéntico (re: archivo, línea, nombre) se acumulan automáticamente en estadísticas de función única.

dump_stats (*filename*)

Guarda los datos cargados en el objeto `Stats` en un archivo llamado *filename*. El archivo se crea si no existe y se sobrescribe si ya existe. Esto es equivalente al método del mismo nombre en las clases: `class profile.Profile` y `cProfile.Profile`.

sort_stats (**keys*)

Este método modifica el objeto `Stats` ordenándolo de acuerdo con los criterios proporcionados. El argumento puede ser una cadena de caracteres o una enumeración `SortKey` que identifica la base de una clasificación (ejemplo: `'time'`, `'name'`, `SortKey.TIME` o `SortKey.NAME`). El argumento enumeraciones `SortKey` tiene ventaja sobre el argumento de cadena de caracteres en que es más robusto y menos propenso a errores.

Cuando se proporciona más de una llave, se utilizan llaves adicionales como criterios secundarios cuando hay igualdad en todas las llaves seleccionadas antes que ellas. Por ejemplo, `sort_stats(SortKey.NAME, SortKey.FILE)` ordenará todas las entradas de acuerdo con el nombre de su función y resolverá todos los vínculos (nombres de función idénticos) clasificándolos por nombre de archivo.

Para el argumento de cadena de caracteres, se pueden usar abreviaturas para cualquier nombre de llaves, siempre que la abreviatura no sea ambigua.

Los siguientes son la cadena de caracteres válida y `SortKey`:

Arg de cadena de caracteres válido	Arg de enumeración válido	Significado
'calls'	SortKey.CALLS	recuento de llamadas
'cumulative'	SortKey.CUMULATIVE	tiempo acumulado
'cumtime'	N/A	tiempo acumulado
'file'	N/A	nombre de archivo
'filename'	SortKey.FILENAME	nombre de archivo
'module'	N/A	nombre de archivo
'ncalls'	N/A	recuento de llamadas
'pcalls'	SortKey.PCALLS	recuento de llamadas primitivas
'line'	SortKey.LINE	número de línea
'name'	SortKey.NAME	nombre de la función
'nfl'	SortKey.NFL	nombre/archivo/línea
'stdname'	SortKey.STDNAME	nombre estándar
'time'	SortKey.TIME	tiempo interno
'tottime'	N/A	tiempo interno

Tenga en cuenta que todos los tipos de estadísticas están en orden descendente (colocando primero los elementos que requieren más tiempo), donde las búsquedas de nombre, archivo y número de línea están en orden ascendente (alfabético). La sutil distinción entre `SortKey.NFL` y `SortKey.STDNAME` es que el nombre estándar es una especie de nombre tal como está impreso, lo que significa que los números de línea incrustados se comparan de manera extraña. Por ejemplo, las líneas 3, 20 y 40 aparecerían (si los nombres de los archivos fueran los mismos) en el orden de las cadenas 20, 3 y 40. En contraste, `SortKey.NFL` hace una comparación numérica de los números de línea. De hecho, `sort_stats(SortKey.NFL)` es lo mismo que `sort_stats(SortKey.NAME, SortKey.FILENAME, SortKey.LINE)`.

Por razones de compatibilidad con versiones anteriores, se permiten los argumentos numéricos -1, 0, 1, y 2. Se interpretan como `'stdname'`, `'calls'`, `'time'`, y `'cumulative'` respectivamente. Si se usa este formato de estilo antiguo (numérico), solo se usará una clave de clasificación (la tecla numérica) y los argumentos adicionales se ignorarán en silencio.

Nuevo en la versión 3.7: Enumeración `SortKey` añadida.

reverse_order ()

Este método para la clase `Stats` invierte el orden de la lista básica dentro del objeto. Tenga en cuenta que, de forma predeterminada, el orden ascendente vs descendente se selecciona correctamente en función de la llave de clasificación elegida.

print_stats (*restrictions)

Este método para la clase `Stats` imprime un informe como se describe en la definición `profile.run()`.

El orden de la impresión se basa en la última operación `sort_stats()` realizada en el objeto (sujeto a advertencias en `add()` y `strip_dirs()`).

Los argumentos proporcionados (si los hay) se pueden usar para limitar la lista a las entradas significativas. Inicialmente, se considera que la lista es el conjunto completo de funciones perfiladas. Cada restricción es un número entero (para seleccionar un recuento de líneas), o una fracción decimal entre 0.0 y 1.0 inclusive (para seleccionar un porcentaje de líneas), o una cadena que se interpretará como una expresión regular (para que el patrón coincida con el nombre estándar eso está impreso). Si se proporcionan varias restricciones, se aplican secuencialmente. Por ejemplo:

```
print_stats(.1, 'foo:')
```

primero limitaría la impresión al primer 10% de la lista, y luego solo imprimiría las funciones que formaban parte del nombre del archivo `. *foo:.` En contraste, el comando:

```
print_stats('foo:', .1)
```

limitaría la lista a todas las funciones que tengan nombres de archivo: `archivo . *foo:`, y luego procedería a imprimir solo el primer 10% de ellas.

print_callers (*restrictions)

Este método para la clase `Stats` imprime una lista de todas las funciones que llamaron a cada función en la base de datos perfilada. El orden es idéntico al proporcionado por `print_stats()`, y la definición del argumento de restricción también es idéntica. Cada persona que llama se informa en su propia línea. El formato difiere ligeramente según el generador de perfiles que produjo las estadísticas:

- Con `profile`, se muestra un número entre paréntesis después de cada llamada para mostrar cuántas veces se realizó esta llamada específica. Por conveniencia, un segundo número sin paréntesis repite el tiempo acumulado empleado en la función de la derecha.
- Con `cProfile`, cada persona que llama está precedida por tres números: la cantidad de veces que se realizó esta llamada específica y los tiempos totales y acumulativos gastados en la función actual mientras fue invocada por esta persona que llama.

print_callees (*restrictions)

Este método para la clase `Stats` imprime una lista de todas las funciones que fueron llamadas por la función indicada. Aparte de esta inversión de la dirección de las llamadas (re: llamado vs fue llamado por), los argumentos y el orden son idénticos al método `print_callers()`.

27.5.5 ¿Qué es el perfil determinista?

Deterministic profiling está destinada a reflejar el hecho de que se supervisan todos los eventos *function call*, *function return*, y *exception*, y se realizan temporizaciones precisas para los intervalos entre estos eventos (durante los cuales el código del usuario se está ejecutando). Por el contrario, *statistical profiling* (que no se realiza en este módulo) muestrea aleatoriamente el puntero de instrucción efectivo y deduce dónde se está gastando el tiempo. La última técnica tradicionalmente implica menos sobrecarga (ya que el código no necesita ser instrumentado), pero proporciona solo indicaciones relativas de dónde se está gastando el tiempo.

En Python, dado que hay un intérprete activo durante la ejecución, no se requiere la presencia de código instrumentado para realizar un perfil determinista. Python proporciona automáticamente un *hook* (retrollamada opcional) para cada evento. Además, la naturaleza interpretada de Python tiende a agregar tanta sobrecarga a la ejecución, que los perfiles deterministas tienden a agregar solo una pequeña sobrecarga de procesamiento en aplicaciones típicas. El resultado es que el perfil determinista no es tan costoso, pero proporciona estadísticas extensas de tiempo de ejecución sobre la ejecución de un programa Python.

Las estadísticas de recuento de llamadas se pueden usar para identificar errores en el código (recuentos sorprendentes) e identificar posibles puntos de expansión en línea (recuentos altos de llamadas). Las estadísticas de tiempo interno se pueden utilizar para identificar «bucles activos» que deben optimizarse cuidadosamente. Las estadísticas

de tiempo acumulativo deben usarse para identificar errores de alto nivel en la selección de algoritmos. Tenga en cuenta que el manejo inusual de los tiempos acumulativos en este generador de perfiles permite que las estadísticas de implementaciones recursivas de algoritmos se comparen directamente con implementaciones iterativas.

27.5.6 Limitaciones

Una limitación tiene que ver con la precisión de la información de sincronización. Hay un problema fundamental con los perfiladores deterministas que implican precisión. La restricción más obvia es que el «reloj» subyacente solo funciona a una velocidad (típicamente) de aproximadamente 0,001 segundos. Por lo tanto, ninguna medición será más precisa que el reloj subyacente. Si se toman suficientes medidas, entonces el «error» tenderá a promediar. Desafortunadamente, eliminar este primer error induce una segunda fuente de error.

El segundo problema es que «lleva un tiempo» desde que se distribuye un evento hasta que la llamada del generador de perfiles para obtener la hora en realidad *obtiene* el estado del reloj. Del mismo modo, hay un cierto retraso al salir del controlador de eventos del generador de perfiles desde el momento en que se obtuvo el valor del reloj (y luego se retiró), hasta que el código del usuario se está ejecutando nuevamente. Como resultado, las funciones que se llaman muchas veces, o llaman a muchas funciones, generalmente acumularán este error. El error que se acumula de esta manera es típicamente menor que la precisión del reloj (menos de un tic de reloj), pero *puede* acumularse y volverse muy significativo.

El problema es más importante con `profile` que con la sobrecarga inferior `cProfile`. Por esta razón, `profile` proporciona un medio para calibrarse a sí mismo para una plataforma dada para que este error pueda ser eliminado probabilísticamente (en promedio). Después de calibrar el generador de perfiles, será más preciso (en un sentido al menos cuadrado), pero a veces producirá números negativos (cuando el recuento de llamadas es excepcionalmente bajo y los dioses de la probabilidad trabajan en su contra :-)) *No* se alarme por los números negativos en el perfil. Deberían aparecer *solo* si ha calibrado su generador de perfiles, y los resultados son realmente mejores que sin calibración.

27.5.7 Calibración

El generador de perfiles del módulo `profile` resta una constante de cada tiempo de manejo de eventos para compensar la sobrecarga de llamar a la función de tiempo y eliminar los resultados. Por defecto, la constante es 0. El siguiente procedimiento se puede usar para obtener una mejor constante para una plataforma dada (ver [Limitaciones](#)).

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

El método ejecuta el número de llamadas Python dadas por el argumento, directamente y nuevamente bajo el generador de perfiles, midiendo el tiempo para ambas. Luego calcula la sobrecarga oculta por evento del generador de perfiles y la retorna como flotante. Por ejemplo, en un Intel Core i5 de 1.8Ghz con Mac OS X y usando `time.process_time()` de Python como temporizador, el número mágico es aproximadamente 4.04e-6.

El objetivo de este ejercicio es obtener un resultado bastante consistente. Si su computadora es *muy* rápida, o su función de temporizador tiene una resolución pobre, es posible que tenga que pasar 100000, o incluso 1000000, para obtener resultados consistentes.

Cuando tiene una respuesta consistente, hay tres formas de usarla:

```
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
```

(continué en la próxima página)

(proviene de la página anterior)

```
# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

Si tiene una opción, es mejor que elija una constante más pequeña, y luego sus resultados «con menos frecuencia» se mostrarán como negativos en las estadísticas del perfil.

27.5.8 Usando un temporizador personalizado

Si desea cambiar la forma en que se determina el tiempo actual (por ejemplo, para forzar el uso del tiempo del reloj de pared o el tiempo transcurrido del proceso), pase la función de temporización que desee a la clase constructora `Profile`:

```
pr = profile.Profile(your_time_func)
```

El generador de perfiles resultante llamará a `your_time_func`. Dependiendo de si está utilizando `profile.Profile` o `cProfile.Profile`, el valor de retorno de `your_time_func` se interpretará de manera diferente:

`profile.Profile` `your_time_func` debería retornar un solo número o una lista de números cuya suma es la hora actual (como lo que `os.times()` retorna). Si la función retorna un número de tiempo único o la lista de números retornados tiene una longitud de 2, obtendrá una versión especialmente rápida de la rutina de envío.

Tenga en cuenta que debe calibrar la clase de generador de perfiles para la función de temporizador que elija (consulte [Calibración](#)). Para la mayoría de las máquinas, un temporizador que retorna un valor entero solitario proporcionará los mejores resultados en términos de baja sobrecarga durante la creación de perfiles. (`os.times()` es *bastante* malo, ya que retorna una tupla de valores de coma flotante). Si desea sustituir un temporizador mejor de la manera más limpia, obtenga una clase y conecte un método de envío de reemplazo que maneje mejor su llamada de temporizador, junto con la constante de calibración adecuada.

`cProfile.Profile` `your_time_func` debería retornar un solo número. Si retorna enteros, también puede invocar al constructor de la clase con un segundo argumento que especifique la duración real de una unidad de tiempo. Por ejemplo, si `your_integer_time_func` retorna tiempos medidos en miles de segundos, construiría la instancia `Profile` de la siguiente manera:

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

Como la clase `cProfile.Profile` no se puede calibrar, las funciones de temporizador personalizadas deben usarse con cuidado y deben ser lo más rápidas posible. Para obtener los mejores resultados con un temporizador personalizado, puede ser necesario codificarlo en la fuente C del módulo interno `_lsprof`.

Python 3.3 agrega varias funciones nuevas en `time` que se puede usar para realizar mediciones precisas del proceso o el tiempo del reloj de pared (*wall-clock time*). Por ejemplo, vea `time.perf_counter()`.

27.6 `timeit` — Mide el tiempo de ejecución de pequeños fragmentos de código

Código fuente: [Lib/timeit.py](#)

Este módulo proporciona una manera sencilla de cronometrar pequeños bits de código Python. Tiene un *Interfaz de línea de comandos* así como un *callable*. Evita una serie de trampas comunes para medir los tiempos de ejecución. Véase también la introducción de Tim Peters al capítulo «Algorithms» en el libro *Python Cookbook*, publicado por O'Reilly.

27.6.1 Ejemplos básicos

En el ejemplo siguiente se muestra cómo se puede utilizar *Interfaz de línea de comandos* para comparar tres expresiones diferentes:

```
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 5: 30.2 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 5: 27.5 usec per loop
$ python3 -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 5: 23.2 usec per loop
```

Esto se puede lograr desde *Interfaz de Python* con:

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

Un invocable también se puede pasar desde el *Interfaz de Python*:

```
>>> timeit.timeit(lambda: '"-".join(map(str, range(100))), number=10000)
0.19665591977536678
```

Sin embargo, tenga en cuenta que `timeit()` determinará automáticamente el número de repeticiones solo cuando se utiliza la interfaz de línea de comandos. En la sección *Ejemplos* puede encontrar ejemplos más avanzados.

27.6.2 Interfaz de Python

El módulo define tres funciones de conveniencia y una clase pública:

`timeit.timeit(stmt='pass', setup='pass', timer=<default timer>, number=1000000, globals=None)`
 Cree una instancia de *Timer* con la instrucción dada, el código *setup* y la función *timer* y ejecute su método `timeit()` con las ejecuciones *number*. El argumento *globals* opcional especifica un espacio de nombres en el que se ejecutará el código.

Distinto en la versión 3.5: El parámetro opcional *globals* fue añadido.

`timeit.repeat(stmt='pass', setup='pass', timer=<default timer>, repeat=5, number=1000000, globals=None)`

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. The optional *globals* argument specifies a namespace in which to execute the code.

Distinto en la versión 3.5: El parámetro opcional *globals* fue añadido.

Distinto en la versión 3.7: El valor por defecto para *repeat* cambió de 3 a 5.

`timeit.default_timer()`

El temporizador por defecto, que es siempre `time.perf_counter()`.

Distinto en la versión 3.3: `time.perf_counter()` es ahora el temporizador por defecto.

`class timeit.Timer(stmt='pass', setup='pass', timer=<timer function>, globals=None)`

Clase para la velocidad de tiempo de ejecución de pequeños fragmentos de código.

El constructor toma una instrucción que se cronometra, una instrucción adicional utilizada para la instalación, y una función de temporizador. Ambas instrucciones tienen como valor predeterminado 'pass'; la función del temporizador depende de la plataforma (consulte la cadena *doc* del módulo). *stmt* y *setup* también pueden contener varias instrucciones separadas por `;` o líneas nuevas, siempre y cuando no contengan literales de

cadena de varias líneas. La instrucción se ejecutará de forma predeterminada dentro del espacio de nombres `timeit`; este comportamiento se puede controlar pasando un espacio de nombres a `globals`.

Para medir el tiempo de ejecución de la primera instrucción, utilice el método `timeit()`. Los métodos `repeat()` y `autorange()` son métodos de conveniencia para llamar al método `timeit()` varias veces.

El tiempo de ejecución de `setup` se excluye de la ejecución de tiempo total.

Los parámetros `stmt` y `setup` también pueden tomar objetos a los que se puede llamar sin argumentos. Esto embebe llamadas a ellos en una función de temporizador que luego será ejecutada por `timeit()`. Tenga en cuenta que la sobrecarga de tiempo es un poco mayor en este caso debido a las llamadas de función adicionales.

Distinto en la versión 3.5: El parámetro opcional `globals` fue añadido.

timeit (*number=1000000*)

Tiempo *number* ejecuciones de la instrucción principal. Esto ejecuta la instrucción `setup` una vez y, a continuación, retorna el tiempo que se tarda en ejecutar la instrucción principal varias veces, medida en segundos como un float. El argumento es el número de veces que se ejecuta el bucle, por defecto a un millón. La instrucción principal, la instrucción `setup` y la función `timer` que se va a utilizar se pasan al constructor.

Nota: De forma predeterminada, `timeit()` desactiva temporalmente `garbage collection` durante la medición. La ventaja de este enfoque es que hace que los tiempos independientes sean más comparables. La desventaja es que GC puede ser un componente importante del rendimiento de la función que se está midiendo. Si es así, GC se puede volver a habilitar como la primera instrucción en la cadena `setup`. Por ejemplo:

```
timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
```

autorange (*callback=None*)

Determina automáticamente cuántas veces llamar a `timeit()`.

Esta es una función de conveniencia que llama a `timeit()` repetidamente para que el tiempo total \geq 0,2 segundos, retornando el eventual (número de bucles, tiempo tomado para ese número de bucles). Este método llama a `timeit()` con números crecientes de la secuencia 1, 2, 5, 10, 20, 50, ... hasta que el tiempo necesario sea de al menos 0.2 segundos.

Si `callback` se da y no es `None`, se llamará después de cada prueba con dos argumentos: `callback(number, time_taken)`.

Nuevo en la versión 3.6.

repeat (*repeat=5, number=1000000*)

Llama a `timeit()` algunas veces.

Esta es una función de conveniencia que llama a `timeit()` repetidamente, retornando una lista de resultados. El primer argumento especifica cuántas veces se debe llamar a `timeit()`. El segundo argumento especifica el argumento `number` para `timeit()`.

Nota: Es tentador calcular la media y la desviación estándar del vector de resultados e informar de estos. Sin embargo, esto no es muy útil. En un caso típico, el valor más bajo proporciona un límite inferior para la rapidez con la que el equipo puede ejecutar el fragmento de código especificado; valores más altos en el vector de resultado normalmente no son causados por la variabilidad en la velocidad de Python, sino por otros procesos que interfieren con su precisión de sincronización. Así que el `min()` del resultado es probablemente el único número que debería estar interesado en. Después de eso, usted debe mirar todo el vector y aplicar el sentido común en lugar de las estadísticas.

Distinto en la versión 3.7: El valor por defecto para `repeat` cambió de 3 a 5.

print_exc (*file=None*)

Ayudante para imprimir un seguimiento desde el código cronometrado.

Uso típico:

```
t = Timer(...)      # outside the try/except
try:
    t.timeit(...)    # or t.repeat(...)
except Exception:
    t.print_exc()
```

La ventaja sobre el seguimiento estándar es que se mostrarán las líneas de origen de la plantilla compilada. El argumento opcional *file* dirige dónde se envía el seguimiento; por defecto a `sys.stderr`.

27.6.3 Interfaz de línea de comandos

Cuando se llama como un programa desde la línea de comandos, se utiliza el siguiente formulario:

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-h] [statement ...]
```

Cuando las siguientes opciones son entendidas:

- n N, --number=N**
cuantas veces para ejecutar “*statement*”
- r N, --repeat=N**
cuantas veces se va a repetir el temporizador (predeterminado 5)
- s S, --setup=S**
instrucción a ser ejecutada una vez inicialmente (por defecto `pass`)
- p, --process**
mide el tiempo de proceso, no el tiempo total de ejecución, utilizando `time.process_time()` en lugar de `time.perf_counter()`, que es el valor predeterminado
Nuevo en la versión 3.3.
- u, --unit=U**
especifica una unidad de tiempo para la salida del temporizador; puede ser `nsec`, `usec`, `msec` o `sec`
Nuevo en la versión 3.5.
- v, --verbose**
imprime los resultados de tiempo en bruto; repetir para más dígitos de precisión
- h, --help**
imprime un mensaje de uso corto y sale

Se puede dar una instrucción de varias líneas especificando cada línea como un argumento de instrucción independiente; Las líneas con sangría son posibles entrecomillando un argumento y utilizando espacios iniciales. Múltiples opciones `-s` se tratan de forma similar.

Si no se da `-n`, se calcula un número adecuado de bucles intentando aumentar los números de la secuencia 1, 2, 5, 10, 20, 50, ... hasta que el tiempo total sea de al menos 0.2 segundos.

Las mediciones de `default_timer()` pueden verse afectadas por otros programas que se ejecutan en la misma máquina, por lo que lo mejor que se puede hacer cuando es necesario una medición de tiempo precisa es repetir la medición un par de veces y utilizar el mejor tiempo. La opción `-r` es buena para esto; el valor predeterminado de 5 repeticiones es probablemente suficiente en la mayoría de los casos. Puede usar `time.process_time()` para medir el tiempo de CPU.

Nota: Hay una cierta sobrecarga de línea base asociada con la ejecución de una instrucción `pass`. El código aquí no intenta ocultarlo, pero debe ser consciente de ello. La sobrecarga de línea base se puede medir invocando el programa sin argumentos y puede diferir entre versiones de Python.

27.6.4 Ejemplos

Es posible proporcionar una instrucción *setup* que se ejecuta solo una vez al inicio:

```
$ python -m timeit -s 'text = "sample string"; char = "g"' 'char in text'
5000000 loops, best of 5: 0.0877 usec per loop
$ python -m timeit -s 'text = "sample string"; char = "g"' 'text.find(char)'
1000000 loops, best of 5: 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
```

Se puede hacer lo mismo usando la clase *Timer* y sus métodos:

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40183617287970225, 0.37027556854118704, 0.38344867356679524, 0.3712595970846668,
↪ 0.37866875250654886]
```

En los ejemplos siguientes se muestra cómo cronometrar expresiones que contienen varias líneas. Aquí comparamos el coste de usar *hasattr()* frente a *try/except* para probar los atributos de objeto faltantes y presentes:

```
$ python -m timeit 'try:' ' str.__bool__' 'except AttributeError:' ' pass'
20000 loops, best of 5: 15.7 usec per loop
$ python -m timeit 'if hasattr(str, "__bool__"): pass'
50000 loops, best of 5: 4.26 usec per loop

$ python -m timeit 'try:' ' int.__bool__' 'except AttributeError:' ' pass'
200000 loops, best of 5: 1.43 usec per loop
$ python -m timeit 'if hasattr(int, "__bool__"): pass'
100000 loops, best of 5: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__bool__
... 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.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

Para dar al módulo `timeit` acceso a las funciones que defina, puede pasar un parámetro `setup` que contenga una instrucción import:

```
def test():
    """Stupid test function"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

Otra opción es pasar `globals()` al parámetro `globals`, lo que hará que el código se ejecute dentro del espacio de nombres global actual. Esto puede ser más conveniente que especificar individualmente las importaciones:

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```

27.7 trace — Rastrear la ejecución de la declaración de Python

Código fuente: [Lib/trace.py](#)

El módulo `trace` le permite rastrear la ejecución del programa, generar listas de cobertura de declaraciones anotadas, imprimir relaciones entre llamador/destinatario y listar funciones ejecutadas durante la ejecución de un programa. Se puede utilizar en otro programa o desde la línea de comandos.

Ver también:

Coverage.py Una herramienta popular de terceros de cobertura que proporciona salida HTML junto con funciones avanzadas como cobertura de sucursales.

27.7.1 Uso de la línea de comandos

El módulo `trace` se puede invocar desde la línea de comandos. Puede ser tan simple como

```
python -m trace --count -C . somefile.py ...
```

Lo anterior ejecutará `somefile.py` y generará listados anotados de todos los módulos de Python importados durante la ejecución en el directorio actual.

--help

Muestra uso y sale.

--version

Muestra la versión del módulo y sale.

Nuevo en la versión 3.8: Se agregó la opción `--module` que permite ejecutar un módulo ejecutable.

Opciones principales

Se debe especificar al menos una de las siguientes opciones al invocar *trace*. La opción *--listfuncs* es mutuamente excluyente con las opciones *--trace* y *--count*. Cuando se proporciona *--listfuncs*, no se aceptan *--count* ni *--trace*, y viceversa.

- c, --count**
Genera un conjunto de archivos de lista anotados al finalizar el programa que muestra cuántas veces se ejecutó cada instrucción. Vea también *--coverdir*, *--file* y *--no-report* a continuación.
- t, --trace**
Muestra las líneas a medida que se ejecutan.
- l, --listfuncs**
Muestra las funciones ejecutadas al ejecutar el programa.
- r, --report**
Genera una lista anotada de una ejecución del programa anterior que utilizó la opción *--count* y *--file*. Esto no ejecuta ningún código.
- T, --trackcalls**
Muestra las relaciones de llamada expuestas al ejecutar el programa.

Modificadores

- f, --file=<file>**
Nombre de un archivo para acumular recuentos durante varias ejecuciones de seguimiento. Debe usarse con la opción *--count*.
- C, --coverdir=<dir>**
Directorio donde van los archivos del informe. El informe de cobertura para *paquete.modulo* se escribe en el archivo *directorio/paquete/modulo.cobertura*.
- m, --missing**
Al generar listados anotados, marque las líneas que no se ejecutaron con >>>>>.
- s, --summary**
Cuando use *--count* o *--report*, escriba un breve resumen en stdout para cada archivo procesado.
- R, --no-report**
No genera listados anotados. Esto es útil si tiene la intención de realizar varias ejecuciones con *--count*, y luego producir un solo conjunto de listados anotados al final.
- g, --timing**
Prefija cada línea con la hora desde que se inició el programa. Solo se usa durante el rastreo.

Filtros

Estas opciones pueden repetirse varias veces.

- ignore-module=<mod>**
Ignora cada uno de los nombres de módulo dados y sus submódulos (si es un paquete). El argumento puede ser una lista de nombres separados por una coma.
- ignore-dir=<dir>**
Ignora todos los módulos y paquetes en el directorio y subdirectorios nombrados. El argumento puede ser una lista de directorios separados por *os.pathsep*.

27.7.2 Interfaz programática

class `trace.Trace` (*count=1, trace=1, countfuncs=0, countcallers=0, ignoremods=(), ignoredirs=(), infile=None, outfile=None, timing=False*)

Crea un objeto para rastrear la ejecución de una sola declaración o expresión. Todos los parámetros son opcionales. *count* habilita el conteo de números de línea. *trace* habilita el seguimiento de ejecución de línea. *countfuncs* habilita la lista de las funciones llamadas durante la ejecución. *countcallers* habilita el seguimiento de la relación de llamadas. *ignoremods* es una lista de módulos o paquetes para ignorar. *ignoredirs* es una lista de directorios cuyos módulos o paquetes deben ignorarse. *infile* es el nombre del archivo desde el cual leer la información de conteo almacenada. *outfile* es el nombre del archivo en el que se escribe la información de conteo actualizada. *timing* permite mostrar una marca de tiempo relativa al momento en que se inició el seguimiento.

run (*cmd*)

Ejecuta el comando y recopile estadísticas de la ejecución con los parámetros de seguimiento actuales. *cmd* debe ser una cadena o un objeto de código, adecuado para pasar a `exec()`.

runtx (*cmd, globals=None, locals=None*)

Ejecuta el comando y recopile estadísticas de la ejecución con los parámetros de seguimiento actuales, en los entornos globales y locales definidos. Si no está definido, *globals* y *locals* por defecto son diccionarios vacíos.

runfunc (*func, *args, **kwargs*)

Llama a *func* con los argumentos dados bajo el control del objeto `Trace` con los parámetros de rastreo actuales.

results ()

Retorna un objeto `CoverageResults` que contiene los resultados acumulativos de todas las llamadas anteriores a `run`, `runtx` y `runfunc` para la instancia dada `Trace`. No restablece los resultados de seguimiento acumulados.

class `trace.CoverageResults`

Un contenedor para los resultados de la cobertura, creado por `Trace.results()`. No debe ser creado directamente por el usuario.

update (*other*)

Fusiona datos de otro objeto `CoverageResults`.

write_results (*show_missing=True, summary=False, coverdir=None*)

Escribe los resultados de la cobertura. Configure **show_missing** para mostrar las líneas que no tuvieron coincidencias. Configurar *summary* para incluir en la salida el resumen de cobertura por módulo. *coverdir* especifica el directorio en el que se enviarán los archivos de resultado de cobertura. Si es `None`, los resultados de cada archivo fuente se colocan en su directorio.

Un ejemplo simple que demuestra el uso de la interfaz programática:

```
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=".")
```

27.8 tracemalloc— Rastrea la asignación de memoria

Nuevo en la versión 3.4.

Código fuente: [Lib/tracemalloc.py](#)

El módulo `tracemalloc` es una herramienta de depuración para rastrear los espacios de memoria asignados por Python. Este proporciona la siguiente información:

- El rastreo al lugar de origen del objeto asignado
- Las estadísticas en los espacios de memoria asignados por nombre de archivo y por número de línea: tamaño total, número y tamaño promedio de los espacios de memoria asignados
- Calcula las diferencias entre dos informes instantáneos para detectar alguna filtración en la memoria

Para rastrear la mayoría de los espacios de memoria asignados por Python; el módulo debe empezar tan pronto como sea posible configurando la variable del entorno `PYTHONTRACEMALLOC` a 1, o usando la opción del comando de línea `-X tracemalloc`. La función `tracemalloc.start()` puede ser llamada en tiempo de ejecución para empezar a rastrear las asignaciones de memoria de Python.

Por defecto, el rastreo de un espacio de memoria asignado solo guarda el cuadro mas reciente (1 cuadro). Para guardar 25 cuadros desde el `PYTHONTRACEMALLOC` comienzo, configura la variable del entorno a ``25``, o usa `-X tracemalloc=25` en la opción de línea de comando.

27.8.1 Ejemplos

Mostrar los 10 principales

Mostrar los 10 archivos asignando la mayor cantidad de memoria:

```
import tracemalloc

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Ejemplo de la salida del conjunto de pruebas de Python:

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315, ↵
↵average=108 B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

Se puede ver que Python ha cargado 4855 KiB de data (código de bytes y constantes) desde los módulos y que el modulo `collections` asigno 24KiB para crear tipos `namedtuple`.

Mira `Snapshot.statistics()` para más opciones.

Calcula las diferencias

Toma dos capturas instantáneas y muestra las diferencias:

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

Ejemplo de la salida antes y después de probar el conjunto de pruebas de Python:

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369),
↪ average=117 B
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106), ↪
↪ average=119 B
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589),
↪ average=519 B
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526), ↪
↪ average=139 B
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334), ↪
↪ average=86 B
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1), ↪
↪ average=96.0 KiB
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109), ↪
↪ average=784 B
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143 ↪
↪ (+143), average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969 ↪
↪ (+969), average=76 B
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126), ↪
↪ average=546 B
```

Se puede ver que Python cargó 8173 KiB de información del modulo (código de bytes y constantes), y que eso es `4428KiB` más de lo que ha sido cargado antes de los test, cuando la anterior captura de pantalla fue tomada. De manera similar, el modulo `linecache` ha almacenado en caché “940 KiB” del código fuente de Python para formatear los seguimientos, todo desde la captura instantánea.

Si el sistema tiene poca memoria libre, los informes instantáneos pueden ser escritos en el disco usando el método `Snapshot.dump()` para analizar el informe instantáneo offline. Después usa el método `Snapshot.load()` para actualizar el informe.

Consigue el seguimiento del bloque de memoria

Código para configurar el seguimiento del bloque de memoria más grande:

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

Ejemplo de la salida del conjunto de pruebas de Python (rastreo limitado a 25 cuadros):

```
903 memory blocks: 870.1 KiB
File "<frozen importlib._bootstrap>", line 716
File "<frozen importlib._bootstrap>", line 1036
File "<frozen importlib._bootstrap>", line 934
File "<frozen importlib._bootstrap>", line 1068
File "<frozen importlib._bootstrap>", line 619
File "<frozen importlib._bootstrap>", line 1581
File "<frozen importlib._bootstrap>", line 1614
File "/usr/lib/python3.4/doctest.py", line 101
import pdb
File "<frozen importlib._bootstrap>", line 284
File "<frozen importlib._bootstrap>", line 938
File "<frozen importlib._bootstrap>", line 1068
File "<frozen importlib._bootstrap>", line 619
File "<frozen importlib._bootstrap>", line 1581
File "<frozen importlib._bootstrap>", line 1614
File "/usr/lib/python3.4/test/support/__init__.py", line 1728
import doctest
File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
File "/usr/lib/python3.4/test/regtest.py", line 1276
    test_runner()
File "/usr/lib/python3.4/test/regtest.py", line 976
    display_failure=not verbose)
File "/usr/lib/python3.4/test/regtest.py", line 761
    match_tests=ns.match_tests)
File "/usr/lib/python3.4/test/regtest.py", line 1563
    main()
File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

Se puede ver que la mayor parte de la memoria fue asignada en el módulo `importlib` para cargar datos (códigos de bytes y constantes) desde módulos 870.1 KiB. El rastreo esta donde el módulo `importlib` cargó datos más recientemente: en la línea `import pdb` el módulo `doctest`.

Los 10 más bonitos

Codifica para configurar las 10 líneas que asignan gran parte de la memoria con una salida bonita, ignorando los archivos “<frozen importlib._bootstrap>” y <unknown>:

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        print("#%s: %s:%s: %.1f KiB"
              % (index, frame.filename, frame.lineno, stat.size / 1024))
        line = linecache.getline(frame.filename, frame.lineno).strip()
        if line:
            print('    %s' % line)

    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Ejemplo de la salida del conjunto de pruebas de Python:

```
Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
    _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars]
#2: Lib/base64.py:306: 419.8 KiB
    _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars]
#3: collections/__init__.py:368: 293.6 KiB
    exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
    cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
    testMethod()
#6: Lib/linecache.py:127: 95.4 KiB
    lines = fp.readlines()
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
    _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB
```

Mira `Snapshot.statistics()` para más opciones.

27.8.2 API

Funciones

`tracemalloc.clear_traces()`

Limpia los rastros de los bloques de memoria asignados por Python.

Mira también la función `stop()`.

`tracemalloc.get_object_traceback(obj)`

Obtén el rastreo de donde el objeto de Python fue asignado. Retorna una instancia `Traceback` o `None` si el módulo `tracemalloc` no esta rastreando ninguna asignación de memoria o no rastreó la asignación del objeto.

Mira también las funciones `gc.get_referrers()` y `sys.getsizeof()`.

`tracemalloc.get_traceback_limit()`

Obtén el número máximo de cuadros guardados en el seguimiento de un rastro.

El módulo `tracemalloc` debe rastrear las asignaciones de memoria para obtener el límite, de otra manera se inicia una excepción.

El limite es establecido por la función `start()`.

`tracemalloc.get_traced_memory()`

Obtén el tamaño actual y tamaño pico de los bloques de memorias rastreados por el módulo `tracemalloc` como una tupla: (current: int, peak: int).

`tracemalloc.get_tracemalloc_memory()`

Obtén el uso de la memoria en bytes desde el modulo `tracemalloc` usado para guardar rastreos de bloques de memoria. Retorna una clase `int`.

`tracemalloc.is_tracing()`

Si el módulo `tracemalloc` esta rastreando asignaciones de memoria de Python retorna `True` sino retorna `False`.

También mira las funciones `start()` y `stop()`.

`tracemalloc.start(nframe:int=1)`

Empieza a rastrear las asignaciones de memoria de Python: instala *hooks* en las asignaciones de memoria de Python. Los rastreos coleccionados van a estar limitados a *nframe*. Por defecto, un rastro de un bloque de memoria solo guarda el cuadro mas reciente: el limite es 1. *nframe* debe ser mayor o igual a 1.

Guardar mas de 1 cuadro es solo útil para calcular estadísticas agrupadas por seguimiento o para calcular estadísticas acumulativa: mira las funciones `Snapshot.compare_to()` y `Snapshot.statistics()`.

Guardar mas cuadros aumenta la memoria y la sobrecargar de la CPU del modulo `tracemalloc`. Usa la función `get_tracemalloc_memory()` para medir cuanta memoria se usa por el módulo `tracemalloc`.

La variable del entorno `PYTHONTRACEMALLOC` (`PYTHONTRACEMALLOC=NFRAME`) y la opción de comando de linea `-X tracemalloc=NFRAME` se puede usar para empezar a rastrear desde el inicio.

También mira las funciones `stop()`, `is_tracing()` y `get_traceback_limit()`.

`tracemalloc.stop()`

Stop tracing Python memory allocations: uninstall hooks on Python memory allocators. Also clears all previously collected traces of memory blocks allocated by Python.

Llama a la función `take_snapshot()` para tomar una captura instantánea de los rastreos, antes de limpiarlos.

También mira las funciones `start()`, `is_tracing()` y `clear_traces()`.

`tracemalloc.take_snapshot()`

Toma una captura instantánea de los bloques de memoria asignados por Python. Retorna una nueva instancia de la clase `Snapshot`.

La captura instantánea no incluye ningún bloque de memoria asignado antes de que el módulo `tracemalloc` haya empezado a rastrear asignaciones de memoria.

Los seguimientos de los rastros son limitados por la función `get_traceback_limit()`. Usa el parámetro `nframe` de la función `start()` para guardar mas cuadros.

El módulo `tracemalloc` debe empezar a rastrear las asignaciones de memoria para tomar una captura instantánea. Mira la función `start()`.

También mira la función `get_object_traceback()`.

Filtro de dominio

class `tracemalloc.DomainFilter` (*inclusive: bool, domain: int*)

Filtra los rastros de los bloques de memoria por su espacio de dirección (dominio)

Nuevo en la versión 3.6.

inclusive

Si *inclusive* es `True` (incluye), relaciona los bloques de memoria asignados en el espacio de dirección *domain*.

Si *inclusive* es `False` (excluye), relaciona los bloques de memoria no asignados en el espacio de dirección *domain*.

domain

Espacio de dirección de un bloque de memoria (`int`). Propiedad solo-lectura.

Filtro

class `tracemalloc.Filter` (*inclusive: bool, filename_pattern: str, lineno: int=None, all_frames: bool=False, domain: int=None*)

Filtra los rastros de los bloques de memoria.

También mira la función `fnmatch.fnmatch()` para la sintaxis de *filename_pattern*. La extensión `'.pyc'` es remplazada por `'.py'`.

Ejemplos:

- `Filter(True, subprocess.__file__)` solo incluye los rastros de el módulo `subprocess`
- `Filter(False, tracemalloc.__file__)` excluye los rastros de memoria del módulo `tracemalloc`
- `Filter(False, "<unknown>")` excluye los seguimientos vacíos

Distinto en la versión 3.5: La extensión `'.pyo'` ya no se remplaza con `'.py'`.

Distinto en la versión 3.6: Agregado el atributo *domain*.

domain

El espacio de dirección de un bloque de memoria (`int` o `None`).

`tracemalloc` usa el dominio 0 para rastrear las asignaciones de memoria hechas por Python. Las extensiones C pueden usar otros dominios para rastrear otros recursos.

inclusive

Si *inclusive* es `True` (incluye), solo relaciona los bloques de memoria asignados en un archivo con el nombre coincidiendo con el atributo *filename_pattern* en el número de línea del atributo *lineno*.

Si *inclusive* es `False` (excluye), ignora los bloques de memoria asignados en un archivo con el nombre coincidiendo con el atributo *filename_pattern* en el número de línea del atributo *lineno*.

lineno

El número de línea (`int`) del filtro. Si *lineno* es `None`, el filtro se relaciona con cualquier número de línea.

filename_pattern

El patrón del nombre de archivo del filtro (`str`). Propiedad solo-lectura.

all_frames

Si *all_frames* es `True`, todos los cuadros de los rastreos son chequeados. Si *all_frames* es `False`, solo el cuadro mas reciente es chequeado.

El atributo no tiene efecto si el limite de rastreo es 1. Mira la función `get_traceback_limit()` y el atributo `Snapshot.traceback_limit`.

Cuadro

class tracemalloc.**Frame**

Cuadro de un rastreo.

La clase *Traceback* es una secuencia de las instancias de la clase *Frame*.

filename

Nombre de archivo (`str`).

lineno

Número de línea (`int`).

Captura instantánea

class tracemalloc.**Snapshot**

Captura instantánea de los rastros de los bloques de memoria asignados por Python.

La función `take_snapshot()` crea una instancia de captura instantánea.

compare_to (*old_snapshot: Snapshot, key_type: str, cumulative: bool=False*)

Calcula las diferencias con una vieja captura instantánea. Obtiene las estadísticas en una lista ordenada de instancias de la clase *StatisticDiff* agrupadas por *key_type*.

Mira el método `Snapshot.statistics()` para los parámetros *key_type* y *cumulative*.

El resultado esta guardado desde el más grande al más pequeño por los valores absolutos de `StatisticDiff.size_diff()`, `StatisticDiff.size()`, el valor absoluto de `StatisticDiff.count_diff()`, `Statistic.count()` y después por el atributo `StatisticDiff.traceback()`.

dump (*filename*)

Escribe la captura instantánea en un archivo.

Usa el método `load()` para recargar la captura instantánea.

filter_traces (*filters*)

Crea una nueva instancia de clase *Snapshot* con una secuencia de *traces* filtrados, *filters* es una lista de las instancias de *DomainFilter* y *Filter*. Si *filters* es una lista vacía, retorna una nueva instancia de clase *Snapshot* con una copia de los rastreos.

Los filtros `todo` `incluido` se aplican de a uno, si los filtros no incluidos coinciden. Si al menos un filtro exclusivo coincide, se ignora un rastro.

Distinto en la versión 3.6: Las instancias de clase *DomainFilter* ahora también son aceptadas en *filters*.

classmethod **load** (*filename*)

Carga la captura instantánea desde un archivo.

También mira el método `dump()`.

statistics (*key_type: str, cumulative: bool=False*)

Obtiene estadísticas como una lista ordenada, de instancias de *Statistic* agrupadas por *key_type*:

key_type	descripción
'filename'	nombre del archivo
'lineno'	nombre del archivo y número de línea
'traceback'	seguimiento

Si *cumulative* es *True*, acumula el tamaño y cuenta los bloques de memoria de todos los cuadros del seguimiento de un rastro, no solo el cuadro mas reciente. El modo acumulativo solo puede ser usado cuando *key_type* se iguala a 'filename' y 'lineno'.

El resultado se organiza desde el más grande hasta el más pequeño por el atributo *Statistic.size*, *Statistic.count* y después por *Statistic.traceback*.

traceback_limit

El número máximo de cuadros organizados en el rastreo del atributo *traces*: resulta de la función *get_traceback_limit* cuando la captura instantánea fue tomada.

traces

Rastros de todos los bloques de memoria asignados por Python: secuencia de instancias de *Trace*.

La secuencia tiene un orden que no esta definido. Usa el método *Snapshot.statistics()* para obtener una lista ordenada de estadísticas.

Estadística

class tracemalloc.*Statistic*

Estadística de las asignaciones de memoria.

Snapshot.statistics() retorna una lista de instancias de la clase *Statistic*.

También mira la clase *StatisticDiff*.

count

Número de bloques de memoria (*int*).

size

El tamaño total de los bloques de memoria en bytes (*int*).

traceback

Rastrea donde un bloque de memoria fue asignado, la instancia de la clase *Traceback*.

StatisticDiff

class tracemalloc.*StatisticDiff*

La diferencia de estadística en las asignaciones de memoria entre una vieja y una nueva instancia de clase *Snapshot*.

Snapshot.compare_to() retorna una lista de instancias de la clase *StatisticDiff*. Mira también la clase *Statistic*.

count

El número de bloques de memoria en la nueva captura de pantalla (*int*): 0 si los bloques de memoria han sido liberados en la nueva captura instantánea.

count_diff

La diferencia de los números de los bloques de memoria entre las capturas viejas y nuevas (*int*): 0 si el bloque de memoria ha sido asignado en la nueva captura instantánea.

size

El tamaño total de los bloques de memoria en bytes en la nueva captura instantánea (*int*): 0 si el bloque de memoria ha sido liberado en la nueva captura instantánea.

size_diff

La diferencia de el tamaño total de un bloque de memoria en bytes entre las capturas instantáneas viejas y nuevas (`int`): 0 si el bloque de memoria ha sido asignado en la nueva captura instantánea.

traceback

Rastrea donde los bloques de memoria han sido asignados, instancia de la clase `Traceback`.

Rastro

class `tracemalloc.Trace`

Rastro de un bloque de memoria.

El atributo `Snapshot.traces` es una secuencia de las instancias de la clase `Trace`.

Distinto en la versión 3.6: Agregado el atributo `domain`.

domain

Espacio de dirección de un bloque de memoria (`int`). Propiedad solo-lectura.

`tracemalloc` usa el dominio 0 para rastrear las asignaciones de memoria hechas por Python. Las extensiones C pueden usar otros dominios para rastrear otros recursos.

size

Tamaño de un bloque de memoria en bytes (`int`).

traceback

Rastrea donde un bloque de memoria fue asignado, la instancia de la clase `Traceback`.

Seguimiento

class `tracemalloc.Traceback`

La secuencia de las instancias de la clase `Frame` organizadas desde el cuadro mas antiguo al más reciente.

Un seguimiento contiene por lo menos 1 cuadro. Si el módulo `tracemalloc` falla al traer un cuadro, se usa el nombre del archivo "`<unknown>`" en el número de línea 0.

Cuando se toma una captura instantánea, los seguimientos de los rastros son limitados a `get_traceback_limit()` cuadros. Mira la función `take_snapshot()`.

El atributo `Trace.traceback` es una instancia de la clase `Traceback`.

Distinto en la versión 3.7: Los cuadros están organizados desde el mas antiguo hasta el más reciente, en vez de el más reciente al más antiguo.

format (`limit=None`, `most_recent_first=False`)

Formatea el seguimiento como una lista de líneas con nuevas líneas. Usa el módulo `linecache` para obtener líneas del código fuente. Si se establece `limit`, si `limit` es positivo: formatea el cuadro mas reciente. Si no, formatea los `abs(limit)` cuadros más antiguos. Si `most_recent_first` es `True`, el orden de los cuadros formateados es invertido, retornando primero el cuadro más reciente en vez del último.

Similar a la función `traceback.format_tb()`, excepto por el método `format()` que no incluye nuevas líneas.

Ejemplo:

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

Salida:

```
Traceback (most recent call first):  
  File "test.py", line 9  
    obj = Object()  
  File "test.py", line 12  
    tb = tracemalloc.get_object_traceback(f())
```

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.

28.1 `distutils` — Creación e instalación de módulos Python

El paquete `distutils` proporciona soporte para crear e instalar módulos adicionales en una instalación de Python. Los nuevos módulos pueden ser 100% Python puro, o pueden ser módulos de extensión escritos en C, o pueden ser colecciones de paquetes de Python que incluyen módulos programados en Python y C.

La mayoría de los usuarios de Python *no* querrán utilizar este módulo directamente, sino que usarán las herramientas cross-version mantenidas por la Python Packaging Authority. En particular, `setuptools` es una alternativa mejorada a `distutils` que proporciona:

- soporte para declarar dependencias del proyecto
- mecanismos adicionales para configurar cuáles archivos incluir en lanzamientos de código fuente (incluyendo plugins para la integración con sistemas de control de versiones)
- la capacidad de declarar «puntos de entrada» del proyecto, los cuales pueden ser utilizados como base para los sistemas de plugins de aplicaciones
- la capacidad de generar, automáticamente, ejecutables de línea de comandos de Windows en el momento de la instalación, en lugar de tener que compilarlos previamente
- comportamiento consistente en todas las versiones de Python soportadas

El instalador `pip` recomendado ejecuta todos los scripts `setup.py` con `setuptools`, incluso si el propio script sólo importa `distutils`. Consulte la [Python Packaging User Guide](#) para más información.

Para beneficio de los autores de herramientas de empaquetado y los usuarios que buscan una comprensión más profunda de los detalles del sistema actual de empaquetado y distribución, la documentación de usuario heredada basada en `distutils` y la referencia de la API permanecen disponibles:

- `install-index`
- `distutils-index`

28.2 `ensurepip` — Ejecutando el instalador `pip`

Nuevo en la versión 3.4.

El paquete `ensurepip` proporciona soporte para ejecutar el instalador `pip` en una instalación de Python existente o en un entorno virtual. Este enfoque de arranque refleja el hecho de que `pip` es un proyecto independiente con su propio ciclo de lanzamiento, y la última versión estable disponible se incluye con el mantenimiento y las versiones de características del intérprete de referencia CPython.

En la mayoría de los casos, los usuarios finales de Python no deberían tener que invocar este módulo directamente (como `pip` deben arrancarse de forma predeterminada), pero puede ser necesario si se omitió la instalación de `pip` al instalar Python (o al crear un entorno virtual) o después de desinstalar explícitamente `pip`.

Nota: Este módulo *no* accede a Internet. Todos los componentes necesarios para ejecutar `pip` se incluyen como partes internas del paquete.

Ver también:

installing-index La guía del usuario final para instalar paquetes python

PEP 453: Arranque explícito de `pip` en instalaciones de Python La justificación original y la especificación de este módulo.

28.2.1 Interfaz de línea de comandos

La interfaz de línea de comandos se invoca mediante el modificador `-m` del intérprete.

La invocación más simple posible es:

```
python -m ensurepip
```

Esta invocación instalará `pip` si aún no está instalado, pero de lo contrario no hace nada. Para asegurarse de que la versión instalada de `pip` es al menos tan reciente como la incluida con `ensurepip`, pase la opción `--upgrade`:

```
python -m ensurepip --upgrade
```

De forma predeterminada, `pip` se instala en el entorno virtual actual (si uno está activo) o en los paquetes de sitio del sistema (si no hay ningún entorno virtual activo). La ubicación de instalación se puede controlar a través de dos opciones de línea de comandos adicionales:

- `--root <dir>`: Instala `pip` en relación con el directorio raíz dado en lugar de la raíz del entorno virtual activo actualmente (si existe) o la raíz predeterminada para la instalación actual de Python.
- `--user`: Instala `pip` en el directorio de paquetes de sitio de usuario en lugar de globalmente para la instalación actual de Python (esta opción no está permitida dentro de un entorno virtual activo).

De forma predeterminada, se instalarán los scripts `pipX` y `pipX.Y` (donde `X.Y` representa la versión de Python utilizada para invocar `ensurepip`). Los scripts instalados se pueden controlar a través de dos opciones de línea de comandos adicionales:

- `--altinstall`: si se solicita una instalación alternativa, *no* se instalará el script `pipX`.
- `--default-pip`: si se solicita una instalación de «`pip` predeterminado», se instalará el script `pip` además de los dos scripts regulares.

Proporcionar ambas opciones de selección de script desencadenará una excepción.

28.2.2 API del módulo

`ensurepip` expone dos funciones para su uso programático:

`ensurepip.version()`

Retorna una cadena de caracteres que especifica la versión incluida de pip que se instalará al ejecutarlo en un entorno.

`ensurepip.bootstrap(root=None, upgrade=False, user=False, altinstall=False, default_pip=False, verbosity=0)`

Ejecuta pip en el entorno actual o designado.

root especifica un directorio raíz alternativo para instalar en relación con. Si *root* es `None`, la instalación utiliza la ubicación de instalación predeterminada para el entorno actual.

upgrade indica si se debe actualizar o no una instalación existente de una versión anterior de pip a la versión incluida.

user indica si se debe utilizar el esquema de usuario en lugar de instalar globalmente.

De forma predeterminada, se instalarán los scripts `pipX` y `pipX.Y` (donde `X.Y` representa la versión actual de Python).

Si se establece *altinstall*, no se instalará `pipX`.

Si se establece *default_pip*, se instalará pip además de los dos scripts normales.

Establecer tanto *altinstall* como *default_pip* desencadenará `ValueError`.

verbosity controla el nivel de salida a `sys.stdout` de la operación de ejecución.

Genera un evento `auditing.ensurepip.bootstrap` con el argumento *root*.

Nota: El proceso de ejecución tiene efectos secundarios tanto en `sys.path` como `os.environ`. Invocar la interfaz de línea de comandos en un subprocesso en su lugar permite evitar estos efectos secundarios.

Nota: El proceso de ejecución puede instalar módulos adicionales requeridos por pip, pero otro software no debe asumir que esas dependencias siempre estarán presentes de forma predeterminada (ya que las dependencias se pueden eliminar en una versión futura de pip).

28.3 venv — Creación de entornos virtuales

Nuevo en la versión 3.3.

Código fuente: [Lib/venv/](#)

El módulo `venv` proporciona soporte para crear «entornos virtuales» ligeros con sus propios directorios de ubicación, aislados opcionalmente de los directorios de ubicación del sistema. Cada entorno virtual tiene su propio binario Python (que coincide con la versión del binario que se utilizó para crear este entorno) y puede tener su propio conjunto independiente de paquetes Python instalados en sus directorios de ubicación.

Ver [PEP 405](#) para más información sobre los entornos virtuales de Python.

Ver también:

[Python Packaging User Guide: Creating and using virtual environments](#)

28.3.1 Creación de entornos virtuales

La creación de *entornos virtuales* se hace ejecutando el comando `venv`:

```
python3 -m venv /path/to/new/virtual/environment
```

Al ejecutar este comando se crea el directorio de destino (creando cualquier directorio padre que todavía no exista) y coloca un archivo `pyenv.config` en él con una clave `home` apuntando a la instalación Python desde la que el comando se lanzó (un nombre común para el directorio destino es `.venv`). También crea un subdirectorio `bin` (o `Scripts` en Windows) conteniendo una copia/enlace simbólico del binario/s Python (según lo apropiado para la plataforma o los argumentos usados en el momento de la creación del entorno). También crea un directorio (inicialmente vacío) `lib/pythonX.Y/site-packages` (en Windows, este es `Lib\site-packages`). Si se especifica un directorio existente, será reutilizado.

Obsoleto desde la versión 3.6: `pyenv` fue la herramienta recomendada para la creación de entornos virtuales en Python 3.3 y 3.4, y es *obsoleta para Python 3.6*.

Distinto en la versión 3.5: Ahora se recomienda el uso de `venv` para crear entornos virtuales.

En Windows, invoca el comando `venv` de la siguiente manera:

```
c:\>c:\Python35\python -m venv c:\path\to\myenv
```

Como alternativa, si configuraste las variables `PATH` y `PATHEXT` para tu instalación de Python:

```
c:\>python -m venv c:\path\to\myenv
```

El comando, si se ejecuta con `-h`, mostrará las opciones disponibles:

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
           [--upgrade] [--without-pip] [--prompt PROMPT]
           ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target directories.

positional arguments:
  ENV_DIR                A directory to create the environment in.

optional arguments:
  -h, --help            show this help message and exit
  --system-site-packages
                        Give the virtual environment access to the system
                        site-packages dir.
  --symlinks            Try to use symlinks rather than copies, when symlinks
                        are not the default for the platform.
  --copies              Try to use copies rather than symlinks, even when
                        symlinks are the default for the platform.
  --clear              Delete the contents of the environment directory if it
                        already exists, before environment creation.
  --upgrade            Upgrade the environment directory to use this version
                        of Python, assuming Python has been upgraded in-place.
  --without-pip        Skips installing or upgrading pip in the virtual
                        environment (pip is bootstrapped by default)
  --prompt PROMPT      Provides an alternative prompt prefix for this
                        environment.

Once an environment has been created, you may wish to activate it, e.g. by
sourcing an activate script in its bin directory.
```

Distinto en la versión 3.4: Instala pip por defecto, añadidas las opciones `--without-pip` y `--copies`

Distinto en la versión 3.4: En versiones anteriores, si el directorio objetivo ya existía, se producía un error, a menos que se incluyera la opción `--clear` or `--upgrade`.

Nota: Aunque los enlaces simbólicos están soportados en Windows, no se recomiendan. Cabe destacar que al hacer doble clic en `python.exe` en el Explorador de Archivos, el enlace simbólico se resolverá con rapidez e ignorará el entorno virtual.

Nota: En Microsoft Windows, puede ser necesario habilitar el *script* `Activate.ps1` estableciendo la política de ejecución para el usuario. Puedes hacer esto ejecutando el siguiente comando de PowerShell:

```
PS C:> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

See [About Execution Policies](#) for more information.

El archivo creado `pyvenv.cfg` también incluye la clave `include-system-site-packages`, fijada a `true` si `venv` se ejecuta con la opción `--system-site-packages`, `false` en caso contrario.

A menos que se use la opción `--without-pip`, se invocará *ensurepip* para arrancar `pip` en el entorno virtual.

Se pueden dar múltiples rutas a `venv`, en cuyo caso se creará un entorno virtual idéntico, de acuerdo a las opciones dadas, en cada ruta proporcionada.

Una vez que se ha creado un entorno virtual, puede ser «activado» usando un *script* en el directorio binario del entorno virtual. La ejecución del *script* es específica de la plataforma (`<venv>` debe ser reemplazado por la ruta del directorio que contiene el entorno virtual):

Plataforma	Consola	Comando para activar el entorno virtual
POSIX	bash/zsh	<code>\$ source <venv>/bin/activate</code>
	fish	<code>\$. <venv>/bin/activate.fish</code>
	csh/tcsh	<code>\$ source <venv>/bin/activate.csh</code>
Windows	PowerShell Core	<code>\$ <venv>/bin/Activate.ps1</code>
	cmd.exe	<code>C:\> <venv>\Scripts\activate.bat</code>
	PowerShell	<code>PS C:\> <venv>\Scripts\Activate.ps1</code>

When a virtual environment is active, the `VIRTUAL_ENV` environment variable is set to the path of the virtual environment. This can be used to check if one is running inside a virtual environment.

No es *necesario* activar un entorno específicamente; la activación sólo prepara el directorio binario del entorno virtual para tu ruta, de modo que «python» invoque el intérprete de Python del entorno virtual y puedas ejecutar los *scripts* instalados sin tener que usar su ruta completa. Sin embargo, todos los *scripts* instalados en un entorno virtual deberían ser ejecutables sin activarlo, y ejecutarse con el Python del entorno virtual automáticamente.

Puedes desactivar un entorno virtual tecleando «*deactivate*» en tu consola. El mecanismo exacto es específico de la plataforma y es un detalle de implementación interna (típicamente se usará un *script* o una función de la consola).

Nuevo en la versión 3.4: *fish* y *csh* *scripts* de activación.

Nuevo en la versión 3.8: *scripts* de activación PowerShell instalados bajo POSIX para el soporte del PowerShell Core.

Nota: Un entorno virtual es un entorno Python en el que el intérprete Python, las bibliotecas y los *scripts* instalados en él están aislados de los instalados en otros entornos virtuales, y (por defecto) cualquier biblioteca instalada en un «sistema» Python, es decir, uno que esté instalado como parte de tu sistema operativo.

Un entorno virtual es un árbol de directorios que contiene archivos ejecutables de Python y otros archivos que indican que es un entorno virtual.

Las herramientas de instalación habituales como *setuptools* y *pip* funcionan como se espera con entornos virtuales. En otras palabras, cuando un entorno virtual está activo, instalan los paquetes Python en el entorno virtual sin necesidad de que se les diga explícitamente que lo hagan.

Cuando un entorno virtual está activo (es decir, el intérprete de Python del entorno virtual se está ejecutando), los atributos `sys.prefix` y `sys.exec_prefix` apuntan al directorio base del entorno virtual, mientras que `sys.base_prefix` y `sys.base_exec_prefix` apuntan a la instalación Python del entorno no virtual que se utilizó para crear el entorno virtual. Si un entorno virtual no está activo, entonces `sys.prefix` es lo mismo que `sys.base_prefix` y `sys.exec_prefix` es lo mismo que `sys.base_exec_prefix` (todos ellos apuntan a una instalación Python de entorno no virtual).

Cuando un entorno virtual está activo, cualquier opción que cambie la ruta de instalación será ignorada de todos los archivos de configuración de `distutils` para evitar que los proyectos se instalen inadvertidamente fuera del entorno virtual.

Cuando se trabaja en una consola, las/os usuarias/os pueden hacer que un entorno virtual se active ejecutando un *script* `activate` en el directorio de ejecutables del entorno virtual (el nombre preciso del archivo y el comando para utilizarlo dependen del entorno de consola), que envía previamente el directorio de ejecutables del entorno virtual a la variable de entorno `PATH` para la consola en ejecución. En otras circunstancias no debería ser necesario activar un entorno virtual; los *scripts* instalados en entornos virtuales tienen una línea «*shebang*» que apunta al intérprete Python del entorno virtual. Esto significa que el *script* se ejecutará con ese intérprete sin importar el valor de `PATH`. En Windows, el procesamiento de la línea «*shebang*» está soportado si tienes instalado el Python *Launcher* para Windows (este fue añadido a Python en 3.3 - ver [PEP 397](#) para más detalles). Por lo tanto, al hacer doble clic en un *script* instalado en una ventana del Explorador de Windows se debería ejecutar el *script* con el intérprete correcto sin necesidad de que haya ninguna referencia a su entorno virtual en `PATH`.

28.3.2 API

El método de alto nivel descrito anteriormente utiliza una sencilla API que proporciona mecanismos para que los creadores de entornos virtuales de terceras/os puedan personalizar la creación de entornos según sus necesidades, la clase `EnvBuilder`.

```
class venv.EnvBuilder (system_site_packages=False, clear=False, symlinks=False, upgrade=False,
                      with_pip=False, prompt=None)
```

La clase `EnvBuilder` acepta los siguientes argumentos de palabras clave en la instanciación:

- `system_site_packages` – un valor booleano que indica que los *site-packages* del sistema Python deben estar disponibles para el entorno (por defecto es `False`).
- `clear` – un valor booleano que, si es verdadero, borrará el contenido de cualquier directorio de destino existente, antes de crear el entorno.
- `symlinks` – un valor booleano que indica si se debe intentar crear un enlace simbólico del binario de Python en lugar de copiarlo.
- `upgrade` – un valor booleano que, si es verdadero, actualizará un entorno existente con el Python en ejecución – para ser usado cuando ese Python haya sido actualizado in situ (por defecto es `False`).
- `with_pip` – un valor booleano que, si es verdadero, asegura que `pip` está instalado en el entorno virtual. Esto usa `ensurepip` con la opción `--default-pip`.
- `prompt` – una cadena que se usará después de que se active el entorno virtual (por defecto es `None` lo que significa que se usará el nombre de directorio del entorno).

Distinto en la versión 3.4: Añadido el parámetro `with_pip`

Nuevo en la versión 3.6: Añadido el parámetro `prompt`

Las/Los creadoras/es de herramientas de entorno virtual de terceros/as serán libres de usar la clase `EnvBuilder` proporcionada como clase base.

El *env-builder* retornado es un objeto que tiene un método, `create`:

```
create (env_dir)
```

Crear un entorno virtual especificando el directorio de destino (de forma absoluta o relativa al directorio actual) que ha de contener el entorno virtual. El método `create` creará el entorno en el directorio especificado, o lanzará la correspondiente excepción.

El método `create` de la clase `EnvBuilder` ilustra los enlaces disponibles para la personalización de la subclase:

```
def create(self, env_dir):
    """
    Create a virtualized Python environment in a directory.
    env_dir is the target directory to create an environment in.
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)
```

Cada uno de los métodos `ensure_directories()`, `create_configuration()`, `setup_python()`, `setup_scripts()` y `post_setup()` pueden ser anulados.

ensure_directories (*env_dir*)

Crea el directorio del entorno y todos los directorios necesarios, y retorna un objeto de contexto. Esto es sólo un soporte para atributos (como rutas), para ser usado por los otros métodos. Se permite que los directorios ya existan, siempre y cuando se haya especificado `clear` o `upgrade` para permitir operar en un directorio de entorno existente.

create_configuration (*context*)

Crea el archivo de configuración `pyvenv.cfg` en el entorno.

setup_python (*context*)

Crea una copia o enlace simbólico al ejecutable Python en el entorno. En los sistemas POSIX, si se usó un ejecutable específico `python3.x`, se crearán enlaces simbólicos a `python` y `python3` apuntando a ese ejecutable, a menos que ya existan archivos con esos nombres.

setup_scripts (*context*)

Instala los *scripts* de activación apropiados para la plataforma en el entorno virtual.

post_setup (*context*)

Un método de marcador de posición que puede ser anulado en implementaciones de terceros/as para previo instalar paquetes en el entorno virtual o realizar otros pasos posteriores a la creación.

Distinto en la versión 3.7.2: Windows ahora usa *scripts* de redireccionamiento para `python[w].exe` en lugar de copiar los propios binarios. Para 3.7.2 solamente `setup_python()` no hace nada a menos que se ejecute desde una compilación en el árbol de directorios fuente.

Distinto en la versión 3.7.3: Windows copia los *scripts* de redireccionamiento como parte de `setup_python()` en lugar de `setup_scripts()`. Este no era el caso para 3.7.2. Cuando se usan enlaces simbólicos, los ejecutables originales se enlazan.

Además, `EnvBuilder` proporciona este método de utilidad que puede ser llamado desde `setup_scripts()` o `post_setup()` en subclases para ayudar a instalar *scripts* personalizados en el entorno virtual.

install_scripts (*context*, *path*)

path es la ruta a un directorio que debería contener los subdirectorios «*common*», «*posix*», «*nt*», cada uno de los cuales contiene *scripts* destinados al directorio *bin* del entorno. El contenido de «*common*» y el directorio correspondiente a `os.name` se copian después de algún reemplazo de texto de los marcadores de posición:

- `__VENV_DIR__` se sustituye por la ruta absoluta del directorio del entorno.
- `__VENV_NAME__` se sustituye por el nombre del entorno (parte final de la ruta del directorio del entorno).
- `__VENV_PROMPT__` se sustituye por el *prompt* (el nombre del entorno entre paréntesis y con un espacio posterior)
- `__VENV_BIN_NAME__` se sustituye con el nombre del directorio *bin* (ya sea `bin` o `Scripts`).

- `__VENV_PYTHON__` se sustituye con la ruta absoluta del ejecutable del entorno.

Se permite la existencia de los directorios (para cuando se está actualizando un entorno existente).

También hay una función de conveniencia a nivel de módulo:

```
venv.create(env_dir, system_site_packages=False, clear=False, symlinks=False, with_pip=False,
            prompt=None)
```

Crea un `EnvBuilder` con los argumentos de la palabra clave dada, y llama a su método `create()` con el argumento `env_dir`.

Nuevo en la versión 3.3.

Distinto en la versión 3.4: Añadido el parámetro `with_pip`

Distinto en la versión 3.6: Añadido el parámetro `prompt`

28.3.3 Un ejemplo de la extensión de `EnvBuilder`

El siguiente *script* muestra como extender `EnvBuilder` implementando una subclase que instala `setuptools` y `pip` en un entorno virtual creado:

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv

class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    This builder installs setuptools and pip so that you can pip or
    easy_install other packages into the created virtual environment.

    :param nodist: If true, setuptools and pip are not installed into the
        created virtual environment.
    :param nopip: If true, pip is not installed into the created
        virtual environment.
    :param progress: If setuptools or pip are installed, the progress of the
        installation can be monitored by passing a progress
        callable. If specified, it is called with two
        arguments: a string indicating some progress, and a
        context indicating where the string is coming from.
        The context argument can have one of three values:
        'main', indicating that it is called from virtualize()
        itself, and 'stdout' and 'stderr', which are obtained
        by reading lines from the output streams of a subprocess
        which is used to install the app.

        If a callable is not specified, default progress
        information is output to sys.stderr.
    """

    def __init__(self, *args, **kwargs):
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
        self.verbose = kwargs.pop('verbose', False)
        super().__init__(*args, **kwargs)

    def post_setup(self, context):
```

(continué en la próxima página)

(proviene de la página anterior)

```

"""
Set up any packages which need to be pre-installed into the
virtual environment being created.

:param context: The information for the virtual environment
                 creation request being processed.
"""
os.environ['VIRTUAL_ENV'] = context.env_dir
if not self.nodist:
    self.install_setuptools(context)
# Can't install pip without setuptools
if not self.nopip and not self.nodist:
    self.install_pip(context)

def reader(self, stream, context):
    """
    Read lines from a subprocess' output stream and either pass to a progress
    callable (if specified) or write progress information to sys.stderr.
    """
    progress = self.progress
    while True:
        s = stream.readline()
        if not s:
            break
        if progress is not None:
            progress(s, context)
        else:
            if not self.verbose:
                sys.stderr.write('.')
            else:
                sys.stderr.write(s.decode('utf-8'))
            sys.stderr.flush()
    stream.close()

def install_script(self, context, name, url):
    _, _, path, _, _, _ = urlparse(url)
    fn = os.path.split(path)[-1]
    binpath = context.bin_path
    distpath = os.path.join(binpath, fn)
    # Download script into the virtual environment's binaries folder
    urlretrieve(url, distpath)
    progress = self.progress
    if self.verbose:
        term = '\n'
    else:
        term = ''
    if progress is not None:
        progress('Installing %s ...%s' % (name, term), 'main')
    else:
        sys.stderr.write('Installing %s ...%s' % (name, term))
        sys.stderr.flush()
    # Install in the virtual environment
    args = [context.env_exe, fn]
    p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
    t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
    t1.start()
    t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
    t2.start()
    p.wait()
    t1.join()
    t2.join()

```

(continué en la próxima página)

(proviene de la página anterior)

```

    if progress is not None:
        progress('done.', 'main')
    else:
        sys.stderr.write('done.\n')
    # Clean up - no longer needed
    os.unlink(distpath)

def install_setuptools(self, context):
    """
    Install setuptools in the virtual environment.

    :param context: The information for the virtual environment
                    creation request being processed.
    """
    url = 'https://bitbucket.org/pypa/setuptools/downloads/ez_setup.py'
    self.install_script(context, 'setuptools', url)
    # clear up the setuptools archive which gets downloaded
    pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
    files = filter(pred, os.listdir(context.bin_path))
    for f in files:
        f = os.path.join(context.bin_path, f)
        os.unlink(f)

def install_pip(self, context):
    """
    Install pip in the virtual environment.

    :param context: The information for the virtual environment
                    creation request being processed.
    """
    url = 'https://bootstrap.pypa.io/get-pip.py'
    self.install_script(context, 'pip', url)

def main(args=None):
    compatible = True
    if sys.version_info < (3, 3):
        compatible = False
    elif not hasattr(sys, 'base_prefix'):
        compatible = False
    if not compatible:
        raise ValueError('This script is only for use with '
                          'Python 3.3 or later')
    else:
        import argparse

        parser = argparse.ArgumentParser(prog=__name__,
                                         description='Creates virtual Python '
                                                         'environments in one or '
                                                         'more target '
                                                         'directories.')
        parser.add_argument('dirs', metavar='ENV_DIR', nargs='+',
                            help='A directory in which to create the '
                                  'virtual environment.')
        parser.add_argument('--no-setuptools', default=False,
                            action='store_true', dest='nodist',
                            help="Don't install setuptools or pip in the "
                                  "virtual environment.")
        parser.add_argument('--no-pip', default=False,
                            action='store_true', dest='nopip',
                            help="Don't install pip in the virtual "
                                  "environment.")

```

(continué en la próxima página)

(proviene de la página anterior)

```

parser.add_argument('--system-site-packages', default=False,
                    action='store_true', dest='system_site',
                    help='Give the virtual environment access to the '
                        'system site-packages dir.')

if os.name == 'nt':
    use_symlinks = False
else:
    use_symlinks = True
parser.add_argument('--symlinks', default=use_symlinks,
                    action='store_true', dest='symlinks',
                    help='Try to use symlinks rather than copies, '
                        'when symlinks are not the default for '
                        'the platform.')
parser.add_argument('--clear', default=False, action='store_true',
                    dest='clear', help='Delete the contents of the '
                        'virtual environment '
                        'directory if it already '
                        'exists, before virtual '
                        'environment creation.')
parser.add_argument('--upgrade', default=False, action='store_true',
                    dest='upgrade', help='Upgrade the virtual '
                        'environment directory to '
                        'use this version of '
                        'Python, assuming Python '
                        'has been upgraded '
                        'in-place.')
parser.add_argument('--verbose', default=False, action='store_true',
                    dest='verbose', help='Display the output '
                        'from the scripts which '
                        'install setuptools and pip.')

options = parser.parse_args(args)
if options.upgrade and options.clear:
    raise ValueError('you cannot supply --upgrade and --clear together.')
builder = ExtendedEnvBuilder(system_site_packages=options.system_site,
                             clear=options.clear,
                             symlinks=options.symlinks,
                             upgrade=options.upgrade,
                             nodist=options.nodist,
                             nopip=options.nopip,
                             verbose=options.verbose)

for d in options.dirs:
    builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)

```

Este *script* está también disponible para su descarga [online](#).

28.4 zipapp — Gestiona archivadores zip ejecutables de Python

Nuevo en la versión 3.5.

Código fuente: [Lib/zipapp.py](#)

Este módulo provee herramientas para administrar la creación de archivos zip que contengan código Python, los que pueden ser ejecutados directamente por el intérprete de Python. El módulo provee tanto una *Interfaz de línea de comando* y una *API de Python*.

28.4.1 Ejemplo básico

El siguiente ejemplo muestra cómo la *Interfaz de línea de comando* puede utilizarse para crear un archivador ejecutable de un directorio que contenga código en Python. Al ponerse en funcionamiento, el archivador ejecutará la función `main` del módulo `myapp` en el archivador.

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

28.4.2 Interfaz de línea de comando

En la ejecución como programa desde la línea de comandos, se utiliza el siguiente formato:

```
$ python -m zipapp source [options]
```

Si *source* es un directorio, se creará un archivador zip para los contenidos de *source*. Si *source* es un archivo, debería ser un archivador, y se copiará al archivador de destino (o los contenidos de su línea *shebang* se mostrarán si se especifica la opción `-info`).

Se aceptan las siguientes opciones:

-o <output>, **--output**=<output>

Escribe la salida a un archivo llamado *output*. Si esta opción no está especificada, el nombre del archivo de salida será el mismo que la entrada *source*, con la extensión `.pyz` agregada. Si se provee de un nombre de archivo explícito, se usa tal como está (por lo que una extensión `.pyz` debería incluirse si esto se requiere).

Un nombre de archivo de salida debe especificarse si *source* es un archivador (y en ese caso, *output* no debería ser igual que *source*).

-p <interpreter>, **--python**=<interpreter>

Agrega una línea `#!` al archivador especificando *interpreter* como comando a ejecutar. También en POSIX, convierte al archivador en ejecutable. La opción por defecto es no escribir una línea `#!`, y no hacer que el archivo sea ejecutable.

-m <mainfn>, **--main**=<mainfn>

Escribe un archivo `__main__.py` en el archivador que ejecuta *mainfn*. El argumento *mainfn* debería tener la forma «pkg.mod:fn», donde «pkg.mod» es un paquete/módulo en el archivador, y «fn» es un invocable (*callable*) en ese módulo. El archivo `__main__.py` ejecutará ese invocable.

`--main` no puede especificarse al copiar un archivador.

-c, **--compress**

Comprime los archivos con el método *deflate*, lo que reduce el tamaño del archivo de salida. Por defecto, los archivos se guardan sin comprimir en el archivador.

`--compress` no surte efecto al copiar un archivador.

Nuevo en la versión 3.7.

--info

Muestra el intérprete incrustado en el archivador, para diagnósticos. En este caso cualquier otra opción se ignora, y SOURCE debe ser un archivador, no un directorio.

-h, --help

Muestra un breve mensaje sobre el modo de uso, y sale.

28.4.3 API de Python

El módulo define dos funciones convenientes:

`zipapp.create_archive(source, target=None, interpreter=None, main=None, filter=None, compressed=False)`

Crea un archivador de aplicación a partir de *source*. Este *source* (origen), puede ser cualquiera de los siguientes:

- El nombre de un directorio, o un *path-like object* que se refiera a un directorio, en cuyo caso un nuevo archivador de aplicación se creará a partir del contenido de dicho directorio.
- El nombre de un archivador de aplicación existente o un *path-like object* que refiera a dicho archivo, en cuyo caso el archivo se copiará al destino (modificándolo para reflejar el valor dado por el argumento *interpreter*. El nombre de archivo debería incluir la extensión `.pyz`, si se requiere.
- Un objeto archivo abierto para lectura en modo bytes. El contenido del archivo debería ser un archivador de aplicación. Se infiere que el objeto archivo está posicionado al comienzo del archivador.

El argumento *target* determina dónde quedará escrito el archivador resultante:

- Si es el nombre de un archivo, o un *path-like object*, el archivador será escrito a ese archivo.
- Si es un objeto archivo abierto, el archivador se escribirá en ese objeto archivo, el cual debe estar abierto para escritura en modo bytes.
- Si el destino (*target*) se omite (o es `None`), el origen (*source*) debe ser un directorio, y el destino será un archivo con el mismo nombre que el origen, con la extensión `.pyz` añadida.

El argumento *interpreter* especifica el nombre del intérprete Python con el que el archivador será ejecutado. Se escribe como una línea «*shebang*» al comienzo del archivador. En POSIX, el Sistema Operativo será quien lo interprete, y en Windows será gestionado por el lanzador Python. Omitir el *interpreter* tendrá como consecuencia que no se escribirá ninguna línea *shebang*. Si se especifica un intérprete y el destino (*target*) es un nombre de archivo, el bit de ejecución del archivo destino será activado.

El argumento *main* especifica el nombre de un invocable (*callable*) que se utilizará como programa principal para el archivador. Solamente se puede especificar si el origen (*source*) es un directorio que no contiene un archivo `__main__.py`. El argumento *main* debería tener la forma «`pkg.module:callable`», y el archivador será ejecutado importando «`pkg.module`» y ejecutando el *callable* sin argumentos. Es un error omitir *main* si el origen es un directorio que no contiene un archivo `__main__.py`, ya que esto resultaría en un archivador no ejecutable.

El argumento opcional *filter* especifica una función *callback* que se pasa como objeto Path, para representar el *path* (ruta) al archivo que se está añadiendo (ruta relativa al directorio origen, *source*). Debería retornar `True` si el archivo será añadido.

El argumento opcional *compressed* determina si los archivos están comprimidos. Si se define como `True`, los archivos en el archivador serán comprimidos con el método `deflate`.

Si se especifica un objeto archivo para *source* o **target**, es responsabilidad de quien invoca cerrarlo luego de invocar a `create_archive`.

Al copiar un archivador existente, los objetos archivo provistos, solamente necesitan los métodos `read` y `readline`, o bien `write`. Al crear un archivador a partir de un directorio, si el destino (*target*) es un objeto archivo, éste se pasará a la clase `zipfile.ZipFile`, y debe proveer los métodos que esa clase necesita.

Nuevo en la versión 3.7: Añadidos los argumentos *filter* y *compressed*.

`zipapp.get_interpreter (archive)`

Retorna el intérprete especificado en la línea `#!` al comienzo del archivador. Si no hay línea `#!`, retorna *None*. El argumento *archive* (archivador), puede ser un nombre de archivo o un objeto tipo archivo abierto para lectura en modo bytes. Se supone que está al principio del archivador.

28.4.4 Ejemplos

Empaqueta un directorio en un archivador, y lo ejecuta.

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

Lo mismo puede lograrse utilizando la función `create_archive()`:

```
>>> import zipapp
>>> zipapp.create_archive('myapp', 'myapp.pyz')
```

Para que la aplicación sea ejecutable directamente en POSIX, especifica un intérprete.

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

Para reemplazar la línea *shebang* en un archivador existente, cree un archivador modificado, utilizando la función `create_archive()`:

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz', '/usr/bin/python3')
```

Para actualizar el archivo en el lugar, reemplaza en memoria utilizando un objeto `BytesIO`, y luego sobrescribe el origen (*source*). Nótese que hay un riesgo al sobrescribir un archivo en el lugar, ya que un error resultará en la pérdida del archivo original. Este código no ofrece protección contra este tipo de errores, sino que el código de producción debería hacerlo. Además, este método solamente funcionará si el archivador cabe en la memoria:

```
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp, '/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
>>>     f.write(temp.getvalue())
```

28.4.5 Especificar el intérprete

Nótese que si se especifica el intérprete y luego se distribuye el archivador de aplicación, es necesario asegurarse de que el intérprete utilizado es portable. El lanzador Python para Windows soporta las formas más comunes de líneas `#!` POSIX, pero hay otras cuestiones a considerar:

- Si se utiliza `#!/usr/bin/env python` (u otras formas del comando «python», como `#!/usr/bin/python`), es necesario considerar que los usuarios quizá tengan tanto Python2 como Python3 como versión por defecto, y el código debe escribirse bajo ambas versiones.
- Si se utiliza una versión específica, por ejemplo `#!/usr/bin/env python3`, la aplicación no funcionará para los usuarios que no tengan esa versión. (Esta puede ser la opción deseada si no se hecho el código compatible con Python 2).
- No hay manera de decir «python X.Y o posterior», así que se debe ser cuidadoso al utilizar una versión exacta, tal como `#!/usr/bin/env python3.4`, ya que será necesario cambiar la línea *shebang* para usuarios de Python 3.5, por ejemplo.

Normalmente, se debería utilizar «/usr/bin/env python2» o «/usr/bin/env python3», según si el código está escrito para Python 2 ó 3.

28.4.6 Creando aplicaciones independientes con zipapp

Utilizando el módulo `zipapp`, es posible crear programas Python auto-contenidos, que pueden ser distribuidos a usuarios finales que solo necesitarán una versión adecuada de Python instalada en sus sistemas. La clave es empaquetar todas las dependencias de la aplicación dentro del archivador, junto al código de la misma.

Los pasos para crear un archivador independiente son los siguientes:

1. Crea tu aplicación en un directorio normalmente, tal que tengas una directorio `myapp` que contenga un archivo `__main__.py`, y cualquier código extra de la aplicación.
2. Instala todas las dependencias de la aplicación en el directorio `myapp`, usando `pip`:

```
$ python -m pip install -r requirements.txt --target myapp
```

(se supone que tienes los requisitos de tu proyecto en un archivo `requirements.txt`, de lo contrario, puedes listar manualmente las dependencias en la línea de comandos de `pip`).

3. Opcionalmente, borra los directorios `.dist-info` creados por `pip` en el directorio `myapp`. Éstos tienen metadatos para que `pip` gestione los paquetes, y como ya no se utilizará `pip`, no hace falta que permanezcan (aunque no causará ningún problema si los dejas).
4. Empaqueta la aplicación utilizando:

```
$ python -m zipapp -p "interpreter" myapp
```

Esto producirá un ejecutable independiente, que puede ser ejecutado en cualquier máquina que disponga del intérprete apropiado. Véase *Especificar el intérprete* para más detalles. Puede enviarse a los usuarios como un solo archivo.

En Unix, el archivo `myapp.pyz` será ejecutable tal como está. Puede ser renombrado, para quitar la extensión `.pyz` si se prefiere un nombre de comando «simple». En Windows, el archivo `myapp.pyz[w]` es ejecutable, ya que el intérprete Python registra las extensiones `.pyz` y `pyzw` al ser instalado.

Hacer un ejecutable para Windows

En Windows, registrar la extensión `.pyz` es opcional, y además hay ciertos sitios que no reconocen las extensiones registradas de manera «transparente» (el ejemplo más simple es que `subprocess.run(['myapp'])` no va a encontrar la aplicación, es necesario especificar explícitamente la extensión).

Por lo tanto, en Windows, suele ser preferible crear un ejecutable a partir del `zipapp`. Esto es relativamente fácil, aunque requiere un compilador de C. La estrategia básica se basa en que los archivos `zip` pueden tener datos arbitrarios antepuestos, y los archivos `exe` de Windows pueden tener datos arbitrarios agregados. Entonces, si se crea un lanzador adecuado mudando el archivo `.pyz` al final del mismo, se obtiene un solo archivo ejecutable que corre la aplicación.

Un lanzador adecuado puede ser así de simple:

```
#define Py_LIMITED_API 1
#include "Python.h"

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#ifdef WINDOWS
int WINAPI wWinMain(
    HINSTANCE hInstance,      /* handle to current instance */
    HINSTANCE hPrevInstance,  /* handle to previous instance */
    LPWSTR lpCmdLine,         /* pointer to command line */
    int nCmdShow              /* show state of window */
)
```

(continué en la próxima página)

(proviene de la página anterior)

```

)
#else
int wmain()
#endif
{
    wchar_t **myargv = __alloca((__argc + 1) * sizeof(wchar_t*));
    myargv[0] = __wargv[0];
    memcpy(myargv + 1, __wargv, __argc * sizeof(wchar_t *));
    return Py_Main(__argc+1, myargv);
}

```

Si se define el símbolo de preprocesador `WINDOWS`, se generará una GUI (interfaz gráfica de usuario) ejecutable, y sin este símbolo, un ejecutable de consola.

Para compilar el ejecutable, se puede usar únicamente la línea de comando estándar `MSVC`, o se puede aprovechar la ventaja de que `distutils` sabe cómo compilar código fuente Python:

```

>>> from distutils.ccompiler import new_compiler
>>> import distutils.sysconfig
>>> import sys
>>> import os
>>> from pathlib import Path

>>> def compile(src):
>>>     src = Path(src)
>>>     cc = new_compiler()
>>>     exe = src.stem
>>>     cc.add_include_dir(distutils.sysconfig.get_python_inc())
>>>     cc.add_library_dir(os.path.join(sys.base_exec_prefix, 'libs'))
>>>     # First the CLI executable
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe)
>>>     # Now the GUI executable
>>>     cc.define_macro('WINDOWS')
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe + 'w')

>>> if __name__ == "__main__":
>>>     compile("zastub.c")

```

El lanzador resultante utiliza «Limited ABI», así que correrá sin cambios con cualquier versión de Python 3.x. Todo lo que necesita es que Python (`python3.dll`) esté en el `PATH` del usuario.

Para una distribución completamente independiente, se puede distribuir el lanzador con la aplicación incluida, empaquetada con la distribución «*embedded*» de Python. Va a funcionar en cualquier PC con la arquitectura adecuada (32 o 64 bits).

Advertencias

Hay algunas limitaciones para empaquetar la aplicación en un solo archivo. En la mayoría, si no en todos los casos, se pueden abordar sin que haga falta ningún cambio importante en la aplicación.

1. Si la aplicación depende de un paquete que incluye una extensión C, ese paquete no puede ser ejecutado desde un archivo zip (esta es una limitación del sistema operativo, dado que el código ejecutable debe estar presente en el sistema de archivos para que el *loader* del SO lo pueda cargar). En este caso, se puede excluir la dependencia del archivo zip, y requerir que los usuarios la tengan instalada, o bien distribuirla conjuntamente con el archivo zip, y agregar código al `__main__.py` para que incluya el directorio que contiene el módulo descomprimido en `sys.path`. En este caso, será necesario asegurarse de distribuir los binarios adecuados para la/s arquitectura/s a las que esté destinada la aplicación (y potencialmente elegir la versión correcta para agregar a `sys.path` en tiempo de ejecución, basándose en la máquina del usuario).

2. Al distribuir ejecutables Windows tal como se describe más arriba, hay que asegurarse de que los usuarios tienen `python3.dll` en su PATH (lo cual no es una opción por defecto en el instalador), o bien empaquetar la aplicación con la distribución *embedded*.
3. El lanzador que se sugiere más arriba, utiliza la API de incrustación de Python (*Python embedding API*). Esto significa que `sys.executable` será la aplicación, y *no* el intérprete Python convencional. El código y sus dependencias deben estar preparados para esta posibilidad. Por ejemplo, si la aplicación utiliza el módulo `multiprocessing`, necesitará invocar a `multiprocessing.set_executable()` para permitir que el módulo sepa dónde encontrar el intérprete Python estándar.

28.4.7 El formato de archivado Zip de aplicaciones Python

Python puede ejecutar archivadores zip que contengan un archivo `__main__.py` desde la versión 2.6. Para que sea ejecutada por Python, basta con que una aplicación de archivador sea un archivo zip estándar que contenga un archivo `__main__.py`, el cual será ejecutado como punto de entrada para la aplicación. Como es usual para cualquier script Python, el elemento padre del script (en este caso, el archivo zip), será ubicado en `sys.path`, por lo que pueden importarse otros módulos desde el archivo zip.

El formato de archivo zip, permite que se antepongan al archivo datos arbitrarios. El formato de aplicación zip utiliza esta capacidad para anteponer una línea «*shebang*» POSIX estándar al archivo (`#!/ruta/al/intérprete`).

Formalmente, el Formato de archivado Zip de aplicaciones Python es:

1. Una línea *shebang* opcional, conteniendo los caracteres `"b"#!"` seguidos por un nombre de intérprete, y luego un carácter de nueva línea (`b'\n'`). El nombre del intérprete puede ser cualquiera que sea aceptable para el procesamiento de *shebang* del Sistema Operativo, o el lanzador Python en Windows. El intérprete debería estar codificado en UTF-8 en Windows, y en `sys.getfilesystemencoding()` en POSIX.
2. Los datos estándares de archivo zip, tal como se generan en el módulo `zipfile`. El contenido del archivo zip *debe* incluir un archivo llamado `__main__.py` (que debe estar en la «raíz» del archivo zip, es decir, no puede estar en un subdirectorío). El los datos del archivo zip pueden estar comprimidos o no.

Si un archivador de aplicación tiene una línea de *shebang*, puede tener el bit de ejecución activado en los sistemas POSIX, para permitir que sea ejecutado directamente.

No se requiere que las herramientas de este módulo sean las que se utilicen para crear archivadores de aplicación. El módulo es útil, pero cualquier archivo que esté en el formato descripto anteriormente es aceptable para Python, no importa cómo haya sido creado.

Servicios en tiempo de ejecución de Python

Los módulos descritos en este capítulo proporcionan una amplia gama de servicios relacionados con el intérprete de Python y su interacción con su entorno. Esta es una descripción general:

29.1 `sys` — Parámetros y funciones específicos del sistema

Este módulo provee acceso a algunas variables usadas o mantenidas por el intérprete y a funciones que interactúan fuertemente con el intérprete. Siempre está disponible.

`sys.abiflags`

En los sistemas POSIX donde Python se construyó con el script estándar `configure`, este contiene los indicadores ABI según lo especificado por [PEP 3149](#).

Distinto en la versión 3.8: Los flags por defecto se convirtieron en una cadena de caracteres vacía (el flag `m` para `pymalloc` ha sido eliminado).

Nuevo en la versión 3.2.

`sys.addaudithook(hook)`

Append the callable *hook* to the list of active auditing hooks for the current (sub)interpreter.

When an auditing event is raised through the `sys.audit()` function, each hook will be called in the order it was added with the event name and the tuple of arguments. Native hooks added by `PySys_AddAuditHook()` are called first, followed by hooks added in the current (sub)interpreter. Hooks can then log the event, raise an exception to abort the operation, or terminate the process entirely.

Lanza un *auditing event* `sys.addaudithook` sin argumentos.

See the [audit events table](#) for all events raised by CPython, and [PEP 578](#) for the original design discussion.

Nuevo en la versión 3.8.

Distinto en la versión 3.8.1: Las excepciones derivadas de `Exception` pero no `RuntimeError` ya no se suprimen.

CPython implementation detail: Cuando el rastreo está habilitado (ver `settrace()`), los ganchos de Python solo se rastrean si el invocable tiene un miembro `__cantrace__` que se establece en un valor verdadero. De lo contrario, las funciones de seguimiento omitirán el enlace.

sys.argv

La lista de argumentos de la línea de comandos pasados a un script de Python. `argv[0]` es el nombre del script (depende del sistema operativo si se trata de una ruta completa o no). Si el comando se ejecutó usando la opción `-c` de la línea de comandos del intérprete, `argv[0]` se establece en la cadena de caracteres `'-c'`. Si no se pasó ningún nombre de secuencia de comandos al intérprete de Python, `argv[0]` es la cadena de caracteres vacía.

Para recorrer la entrada estándar, o la lista de archivos dada en la línea de comando, vea el módulo `fileinput`.

Nota: En Unix, los argumentos de la línea de comandos se pasan por bytes desde el sistema operativo. Python los decodifica con la codificación del sistema de archivos y el controlador de errores «surrogateescape». Cuando necesite bytes originales, puede obtenerlos mediante `[os.fsencode (arg) for arg in sys.argv]`.

sys.audit (event, *args)

Raise an auditing event and trigger any active auditing hooks. *event* is a string identifying the event, and *args* may contain optional arguments with more information about the event. The number and types of arguments for a given event are considered a public and stable API and should not be modified between releases.

For example, one auditing event is named `os.chdir`. This event has one argument called *path* that will contain the requested new working directory.

`sys.audit()` will call the existing auditing hooks, passing the event name and arguments, and will re-raise the first exception from any hook. In general, if an exception is raised, it should not be handled and the process should be terminated as quickly as possible. This allows hook implementations to decide how to respond to particular events: they can merely log the event or abort the operation by raising an exception.

Los ganchos se agregan usando las funciones `sys.addaudithook()` o `PySys_AddAuditHook()`.

El equivalente nativo de esta función es `PySys_Audit()`. Se prefiere usar la función nativa cuando sea posible.

Consulte la [tabla de eventos de auditoría](#) para todos los eventos lanzados por CPython.

Nuevo en la versión 3.8.

sys.base_exec_prefix

Se establece durante el inicio de Python, antes de que se ejecute `site.py`, en el mismo valor que `exec_prefix`. Si no se ejecuta en un [entorno virtual](#), los valores permanecerán iguales; Si `site.py` encuentra que se está utilizando un entorno virtual, los valores de `prefix` y `exec_prefix` se cambiarán para apuntar al entorno virtual, mientras que `base_prefix` y `base_exec_prefix` seguirá apuntando a la instalación base de Python (aquella desde la que se creó el entorno virtual).

Nuevo en la versión 3.3.

sys.base_prefix

Se establece durante el inicio de Python, antes de que se ejecute `site.py`, al mismo valor que `prefix`. Si no se ejecuta en un [entorno virtual](#), los valores permanecerán iguales; si `site.py` encuentra que se está utilizando un entorno virtual, los valores de `prefix` y `exec_prefix` se cambiarán para apuntar al entorno virtual, mientras que `base_prefix` y `base_exec_prefix` seguirá apuntando a la instalación base de Python (aquella desde la que se creó el entorno virtual).

Nuevo en la versión 3.3.

sys.byteorder

Un indicador del orden de bytes nativo. Esto tendrá el valor `'big'` en las plataformas big-endian (el byte más significativo primero) y `'little'` en las plataformas little-endian (el byte menos significativo primero).

sys.builtin_module_names

Una tupla de cadenas de caracteres que proporciona los nombres de todos los módulos que se compilan en este intérprete de Python. (Esta información no está disponible de ninguna otra manera — `modules.keys()` solo enumera los módulos importados.)

`sys.call_tracing(func, args)`

Llama a `func(*args)`, mientras el rastreo está habilitado. El estado de seguimiento se guarda y se restaura posteriormente. Esto está destinado a ser llamado desde un depurador desde un punto de control, para depurar recursivamente algún otro código.

`sys.copyright`

Una cadena de caracteres que contiene los derechos de autor pertenecientes al intérprete de Python.

`sys._clear_type_cache()`

Borra la caché de tipo interno. La caché de tipos se utiliza para acelerar las búsquedas de métodos y atributos. Utilice la función *solo* para eliminar referencias innecesarias durante la depuración de fugas de referencia.

Esta función debe usarse solo para fines internos y especializados.

`sys._current_frames()`

Retorna un diccionario que asigna el identificador de cada subproceso al marco de pila superior actualmente activo en ese subproceso en el momento en que se llama a la función. Tenga en cuenta que las funciones en el módulo *traceback* pueden construir la pila de llamadas dado tal marco.

Esto es más útil para depurar *deadlock*: esta función no requiere la cooperación de los subprocesos con *deadlock*, y las pilas de llamadas de dichos subprocesos se congelan mientras permanezcan en *deadlock*. El marco retornado para un subproceso no en *deadlock* puede no tener relación con la actividad actual de ese subproceso en el momento en que el código de llamada examina el marco.

Esta función debe usarse solo para fines internos y especializados.

Lanza un *auditing event* `sys._current_frames` sin argumentos.

`sys.breakpointhook()`

Esta función de gancho es llamada por built-in *breakpoint()*. De forma predeterminada, lo coloca en el depurador *pdb*, pero se puede configurar en cualquier otra función para que pueda elegir qué depurador se utiliza.

La firma de esta función depende de lo que llame. Por ejemplo, el enlace predeterminado (por ejemplo, `pdb.set_trace()`) no espera argumentos, pero puede vincularlo a una función que espera argumentos adicionales (posicional o palabra clave). La función incorporada *breakpoint()* pasa sus **args* y ***kws* directamente. Lo que sea que retorne *breakpointhooks()* se retorna desde *breakpoint()*.

La implementación predeterminada consulta primero la variable de entorno `PYTHONBREAKPOINT`. Si se establece en "0", esta función vuelve inmediatamente; es decir, no es una operación. Si la variable de entorno no se establece, o se establece en la cadena vacía, se llama a `pdb.set_trace()`. De lo contrario, esta variable debería nombrar una función para ejecutar, utilizando la nomenclatura de importación con puntos de Python, por ejemplo `package.subpackage.module.function`. En este caso, se importaría `package.subpackage.module` y el módulo resultante debe tener un invocable llamado *function()*. Esto se ejecuta, pasando **args* y ***kws*, y lo que sea que *function()* retorna, *sys.breakpointhook()* retorna a la función *breakpoint()*.

Tenga en cuenta que si algo sale mal al importar el invocable nombrado por `PYTHONBREAKPOINT`, se informa un *RuntimeWarning* y se ignora el punto de interrupción (*breakpoint*).

También tenga en cuenta que si *sys.breakpointhook()* se anula mediante programación, `PYTHONBREAKPOINT` no se consulta.

Nuevo en la versión 3.7.

`sys._debugmallocstats()`

Imprime información de bajo nivel para stderr sobre el estado del asignador de memoria de CPython.

Si Python está configurado `-with-pydebug`, también realiza algunas comprobaciones de consistencia internas costosas.

Nuevo en la versión 3.3.

CPython implementation detail: Esta función es específica de CPython. El formato exacto de salida no se define aquí y puede cambiar.

sys.dllhandle

Número entero que especifica el identificador de la DLL de Python.

Disponibilidad: Windows.

sys.displayhook (*value*)

Si *value* no es None, esta función imprime `repr(value)` en `sys.stdout` y guarda *value* en `builtins._`. Si `repr(value)` no se puede codificar en `sys.stdout.encoding` con el controlador de errores `sys.stdout.errors` (que probablemente sea 'strict'), codifíquelo para `sys.stdout.encoding` con el controlador de errores 'backslashreplace'.

Se llama a `sys.displayhook` como resultado de evaluar un *expression* ingresado en una sesión interactiva de Python. La visualización de estos valores se puede personalizar asignando otra función de un argumento a `sys.displayhook`.

Pseudo-código:

```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

Distinto en la versión 3.2: Usa el manejador de error 'backslashreplace' en `UnicodeEncodeError`.

sys.dont_write_bytecode

Si esto es cierto, Python no intentará escribir archivos `.pyc` en la importación de módulos fuente. Este valor se establece inicialmente en True o False según la opción `-B` de la línea de comando y la variable de entorno `PYTHONDONTWRITEBYTECODE`, pero puede configurarlo usted mismo para controlar el código de bytes generación de archivos.

sys.pycache_prefix

Si esto está configurado (no None), Python escribirá archivos de bytecode-cache `.pyc` en (y los leerá desde) un árbol de directorios paralelo enraizado en este directorio, en lugar de `__pycache__` directorios en el árbol de código fuente. Cualquier directorio `__pycache__` en el árbol de código fuente será ignorado y los nuevos archivos `.pyc` se escribirán dentro del prefijo `pycache`. Por lo tanto, si usa `compileall` como paso previo a la compilación, debe asegurarse de ejecutarlo con el mismo prefijo de `pycache` (si lo hay) que usará en tiempo de ejecución.

Una ruta relativa se interpreta en relación con el directorio de directorio actual.

Este valor se establece inicialmente en función del valor de la opción de línea de comandos `-X pycache_prefix=PATH` o la variable de entorno `PYTHONPYCACHEPREFIX` (la línea de comandos tiene prioridad). Si no se establece ninguno, es None.

Nuevo en la versión 3.8.

sys.excepthook (*type, value, traceback*)

Esta función imprime un rastreo y una excepción dados a `sys.stderr`.

Cuando se genera una excepción y no se detecta, el intérprete llama a `sys.excepthook` con tres argumentos, la clase de excepción, la instancia de excepción y un objeto de rastreo. En una sesión interactiva, esto

sucede justo antes de que se retorne el control al indicador; en un programa de Python esto sucede justo antes de que el programa salga. El manejo de tales excepciones de nivel superior se puede personalizar asignando otra función de tres argumentos a `sys.excepthook`.

Lanza un *auditing event* `sys.excepthook` con argumentos `hook`, `type`, `value`, `traceback`.

Ver también:

La función `sys.unraisablehook()` maneja las excepciones que no se pueden evaluar y la función `threading.excepthook()` maneja la excepción lanzada por `threading.Thread.run()`.

```
sys.__breakpointhook__
sys.__displayhook__
sys.__excepthook__
sys.__unraisablehook__
```

Estos objetos contienen los valores originales de `breakpointhook`, `displayhook`, `excepthook` y `unraisablehook` al inicio del programa. Se guardan para que `breakpointhook`, `displayhook` y `excepthook`, `unraisablehook` se puedan restaurar en caso de que sean reemplazados por objetos rotos o alternativos.

Nuevo en la versión 3.7: `__breakpointhook__`

Nuevo en la versión 3.8: `__unraisablehook__`

```
sys.exc_info()
```

Esta función retorna una tupla de tres valores que proporcionan información sobre la excepción que se está manejando actualmente. La información retornada es específica tanto del hilo actual como del marco de pila actual. Si el marco de pila actual no está manejando una excepción, la información se toma del marco de pila que llama, o de quien la llama, y así sucesivamente hasta que se encuentra un marco de pila que está manejando una excepción. Aquí, «manejar una excepción» se define como «ejecutar una cláusula `except`». Para cualquier marco de pila, solo se puede acceder a la información sobre la excepción que se maneja actualmente.

Si no se maneja ninguna excepción en ninguna parte de la pila, se retorna una tupla que contiene tres valores `None`. De lo contrario, los valores retornados son `(type, value, traceback)`. Su significado es: `type` obtiene el tipo de excepción que se está manejando (una subclase de `BaseException`); `value` obtiene la instancia de excepción (una instancia del tipo de excepción); `traceback` obtiene un objeto `traceback` que encapsula la pila de llamadas en el punto donde ocurrió originalmente la excepción.

```
sys.exec_prefix
```

Una cadena de caracteres que proporciona el prefijo de directorio específico del sitio donde están instalados los archivos Python dependientes de la plataforma; de forma predeterminada, también es `'/usr/local'`. Esto se puede configurar en el momento de la compilación con el argumento `--exec-prefix` del script **configure**. Específicamente, todos los archivos de configuración (por ejemplo, el archivo de encabezado `pyconfig.h`) se instalan en el directorio `exec_prefix/lib/pythonXY/config`, y los módulos de la biblioteca compartida se instalan en `exec_prefix/lib/pythonXY/lib-dynload`, donde `XY` es el número de versión de Python, por ejemplo, “3.2”.

Nota: Si un *virtual environment* está en efecto, este valor se cambiará en `site.py` para apuntar al entorno virtual. El valor para la instalación de Python seguirá estando disponible a través de `base_exec_prefix`.

```
sys.executable
```

Una cadena de caracteres que proporciona la ruta absoluta del binario ejecutable para el intérprete de Python, en sistemas donde esto tiene sentido. Si Python no puede recuperar la ruta real a su ejecutable, `sys.executable` será una cadena de caracteres vacía o `None`.

```
sys.exit([arg])
```

Salir de Python. Esto se implementa lanzando la excepción `SystemExit`, por lo que las acciones de limpieza especificadas por las cláusulas finalmente de las declaraciones `try` son respetadas y es posible interceptar el intento de salida en un nivel externo.

El argumento opcional `arg` puede ser un número entero que dé el estado de salida (por defecto es cero) u otro tipo de objeto. Si es un número entero, cero se considera «terminación exitosa» y cualquier valor distinto de cero se considera «terminación anormal» por los shells y similares. La mayoría de los sistemas requieren que esté en

el rango 0-127 y, de lo contrario, producen resultados indefinidos. Algunos sistemas tienen una convención para asignar significados específicos a códigos de salida específicos, pero estos generalmente están subdesarrollados; Los programas Unix generalmente usan 2 para errores de sintaxis de línea de comandos y 1 para todos los demás tipos de errores. Si se pasa otro tipo de objeto, `None` equivale a pasar cero, y cualquier otro objeto se imprime en `stderr` y da como resultado un código de salida de 1. En particular, `sys.exit("algún mensaje de error")` es una forma rápida de salir de un programa cuando ocurre un error.

Dado que `exit()` finalmente «solo» genera una excepción, solo saldrá del proceso cuando sea llamado desde el hilo principal, y la excepción no es interceptada.

Distinto en la versión 3.6: Si se produce un error en la limpieza después de que el intérprete de Python haya detectado `SystemExit` (como un error al vaciar los datos almacenados en el búfer en los flujos estándar), el estado de salida cambia a 120.

`sys.flags`

El *named tuple* `flags` expone el estado de los indicadores de la línea de comando. Los atributos son de solo lectura.

atributo	flag
<code>debug</code>	<code>-d</code>
<code>inspect</code>	<code>-i</code>
<code>interactive</code>	<code>-i</code>
<code>isolated</code>	<code>-I</code>
<code>optimize</code>	<code>-O o -OO</code>
<code>dont_write_bytecode</code>	<code>-B</code>
<code>no_user_site</code>	<code>-s</code>
<code>no_site</code>	<code>-S</code>
<code>ignore_environment</code>	<code>-E</code>
<code>verbose</code>	<code>-v</code>
<code>bytes_warning</code>	<code>-b</code>
<code>quiet</code>	<code>-q</code>
<code>hash_randomization</code>	<code>-R</code>
<code>dev_mode</code>	<code>-X dev</code>
<code>utf8_mode</code>	<code>-X utf8</code>
<code>int_max_str_digits</code>	<code>-X int_max_str_digits</code> (<i>integer string conversion length limitation</i>)

Distinto en la versión 3.2: Agregado el atributo `quiet` para el nuevo flag `-q`.

Nuevo en la versión 3.2.3: El atributo `hash_randomization`.

Distinto en la versión 3.3: Eliminado el atributo obsoleto `division_warning`.

Distinto en la versión 3.4: Agregado el atributo `isolated` para el flag `-I isolated`.

Distinto en la versión 3.7: Se agregó el atributo `dev_mode` para el nuevo flag `-X dev` y el atributo `utf8_mode` para la nueva `-X utf8`.

Distinto en la versión 3.8.14: Added the `int_max_str_digits` attribute.

`sys.float_info`

Un *named tuple* que contiene información sobre el tipo de flotante. Contiene información de bajo nivel sobre la precisión y la representación interna. Los valores corresponden a las diversas constantes de coma flotante definidas en el archivo de encabezado estándar `float.h` para el lenguaje de programación “C”; consulte la sección 5.2.4.2.2 de la norma *ISO/IEC C* de 1999 [C99], “Características de los tipos flotantes”, para obtener más detalles.

atributo	macro float.h	explicación
<code>epsilon</code>	<code>DBL_EPSILON</code>	diferencia entre 1.0 y el menor valor mayor que 1.0 que es representable como flotante
<code>dig</code>	<code>DBL_DIG</code>	número máximo de dígitos decimales que se pueden representar fielmente en un flotante; véase a continuación
<code>mant_dig</code>	<code>DBL_MANT_DIG</code>	precisión de flotantes: el número de dígitos base-radix en el significando de un flotante
<code>max</code>	<code>DBL_MAX</code>	máximo punto flotante positivo representable
<code>max_exp</code>	<code>DBL_MAX_EXP</code>	entero máximo e tal que $\text{radix}^{**}(e-1)$ es un flotante finito representable
<code>max_10_exp</code>	<code>DBL_MAX_10_EXP</code>	entero máximo e tal que $10^{**}e$ está en el rango de flotantes finitos representables
<code>min</code>	<code>DBL_MIN</code>	flotante <i>normalizado</i> mínimo representable positivo
<code>min_exp</code>	<code>DBL_MIN_EXP</code>	entero mínimo e tal que $\text{radix}^{**}(e-1)$ es un flotante normalizado
<code>min_10_exp</code>	<code>DBL_MIN_10_EXP</code>	entero mínimo e tal que $10^{**}e$ es un valor flotante normalizado
<code>radix</code>	<code>FLT_RADIX</code>	base de representación de exponente
<code>rounds</code>	<code>FLT_ROUNDS</code>	constante entera que representa el modo de redondeo utilizado para operaciones aritméticas. Esto refleja el valor de la macro <code>FLT_ROUNDS</code> del sistema en el momento de inicio del intérprete. Consulte la sección 5.2.4.2.2 del estándar C99 para obtener una explicación de los posibles valores y sus significados.

El atributo `sys.float_info.dig` necesita más explicación. Si `s` es cualquier cadena que represente un número decimal con como máximo `sys.float_info.dig` dígitos significativos, entonces la conversión de `s` a un flotante y viceversa recuperará una cadena que representa el mismo decimal valor:

```
>>> 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'
```

Pero para cadenas con más de `sys.float_info.dig` dígitos significativos, esto no siempre es cierto:

```
>>> s = '9876543211234567'    # 16 significant digits is too many!
>>> format(float(s), '.16g')   # conversion changes value
'9876543211234568'
```

`sys.float_repr_style`

Una cadena que indica cómo se comporta la función `repr()` para los flotantes. Si la cadena tiene el valor `'short'`, entonces para un flotante finito `x`, `repr(x)` tiene como objetivo producir una cadena corta con la propiedad de que `float(repr(x)) == x`. Este es el comportamiento habitual en Python 3.1 y posteriores. De lo contrario, `float_repr_style` tiene el valor `'legacy'` y `repr(x)` se comporta de la misma manera que en las versiones de Python anteriores a la 3.1.

Nuevo en la versión 3.1.

`sys.getallocatedblocks()`

Retorna el número de bloques de memoria asignados actualmente por el intérprete, independientemente de su tamaño. Esta función es principalmente útil para rastrear y depurar pérdidas de memoria. Debido a los cachés internos del intérprete, el resultado puede variar de una llamada a otra; puede que tenga que llamar a `_clear_type_cache()` y `gc.collect()` para obtener resultados más predecibles.

Si una construcción o implementación de Python no puede calcular razonablemente esta información, `getallocatedblocks()` puede retornar 0 en su lugar.

Nuevo en la versión 3.4.

`sys.getandroidapilevel()`

Retorna la versión de la API de tiempo de compilación de Android como un número entero.

Disponibilidad: Android.

Nuevo en la versión 3.7.

`sys.getcheckinterval()`

Retorna el «intervalo de verificación» del intérprete; ver `setcheckinterval()`.

Obsoleto desde la versión 3.2: Use `getswitchinterval()` en su lugar.

`sys.getdefaultencoding()`

Retorna el nombre de la codificación de cadena predeterminada actual utilizada por la implementación de Unicode.

`sys.getdlopenflags()`

Retorna el valor actual de las flags que se utilizan para llamadas `dlopen()`. Los nombres simbólicos para los valores de las flags se pueden encontrar en el módulo `os` (constantes `RTLD_XXX`, por ejemplo `os.RTLD_LAZY`).

Disponibilidad: Unix.

`sys.getfilesystemencoding()`

Retorna el nombre de la codificación utilizada para convertir entre nombres de archivo Unicode y nombres de archivo en bytes. Para una mejor compatibilidad, se debe utilizar `str` para los nombres de archivo en todos los casos, aunque también se admite la representación de nombres de archivo como bytes. Las funciones que aceptan o retornan nombres de archivo deben admitir `str` o bytes y se deben convertir internamente a la representación preferida del sistema.

Esta codificación es siempre compatible con ASCII.

`os.fsencode()` y `os.fsdecode()` deben usarse para garantizar que se utilizan la codificación correcta y el modo de errores.

- En el modo UTF-8, la codificación es `utf-8` en cualquier plataforma.
- En macOS, la codificación es `'utf-8'`.
- En Unix, la codificación es la codificación local.
- En Windows, la codificación puede ser `'utf-8'` o `'mbcs'`, según la configuración del usuario.
- En Android, la codificación es `'utf-8'`.
- En VxWorks, la codificación es `'utf-8'`.

Distinto en la versión 3.2: el resultado de `getfilesystemencoding()` ya no puede ser `None`.

Distinto en la versión 3.6: Ya no se garantiza que Windows retorne `'mbcs'`. Consulte [PEP 529](#) y `_enablelegacywindowsfsencoding()` para obtener más información.

Distinto en la versión 3.7: Retorna “utf-8” en el modo UTF-8.

`sys.getfilesystemencodeerrors()`

Retorna el nombre del modo de error utilizado para convertir entre nombres de archivo Unicode y nombres de archivo en bytes. El nombre de la codificación se retorna desde `getfilesystemencoding()`.

`os.fsencode()` y `os.fsdecode()` deben usarse para garantizar que se utilizan la codificación correcta y el modo de errores.

Nuevo en la versión 3.6.

`sys.get_int_max_str_digits()`

Returns the current value for the *integer string conversion length limitation*. See also `set_int_max_str_digits()`.

Nuevo en la versión 3.8.14.

`sys.getrefcount(object)`

Retorna el recuento de referencias del *object*. El recuento retornado es generalmente uno más alto de lo que cabría esperar, porque incluye la referencia (temporal) como argumento para `getrefcount()`.

`sys.getrecursionlimit()`

Retorna el valor actual del límite de recursividad, la profundidad máxima de la pila de intérpretes de Python. Este límite evita que la recursividad infinita cause un desbordamiento de la pila C y bloquee Python. Se puede configurar mediante `setrecursionlimit()`.

`sys.getsizeof(object[, default])`

Retorna el tamaño de un objeto en bytes. El objeto puede ser cualquier tipo de objeto. Todos los objetos integrados retornarán resultados correctos, pero esto no tiene por qué ser cierto para las extensiones de terceros, ya que es una implementación específica.

Solo se tiene en cuenta el consumo de memoria atribuido directamente al objeto, no el consumo de memoria de los objetos a los que se refiere.

Si se proporciona, se retornará *predeterminado* si el objeto no proporciona los medios para recuperar el tamaño. De lo contrario, se generará un `TypeError`.

`getsizeof()` llama al método `__sizeof__` del objeto y agrega una sobrecarga adicional del recolector de basura si el objeto es administrado por el recolector de basura.

Consulte [receta de sizeof recursivo](#) para ver un ejemplo del uso de `getsizeof()` de forma recursiva para encontrar el tamaño de los contenedores y todo su contenido.

`sys.getswitchinterval()`

Retorna el «intervalo de cambio de hilo» del intérprete; ver `setswitchinterval()`.

Nuevo en la versión 3.2.

`sys._getframe([depth])`

Retorna un objeto de marco de la pila de llamadas. Si se proporciona un entero opcional *depth*, retorna el objeto de marco que muchas llamadas debajo de la parte superior de la pila. Si eso es más profundo que la pila de llamadas, se lanza `ValueError`. El valor predeterminado de *depth* es cero, lo que retorna el marco en la parte superior de la pila de llamadas.

Lanza un *auditing event* `sys._getframe` sin argumentos.

CPython implementation detail: Esta función debe utilizarse únicamente para fines internos y especializados. No se garantiza que exista en todas las implementaciones de Python.

`sys.getprofile()`

Obtiene la función de generador de perfiles establecida por `setprofile()`.

`sys.gettrace()`

Obtiene la función de seguimiento (*trace*) establecida por `settrace()`.

CPython implementation detail: La función `gettrace()` está pensada solo para implementar depuradores, perfiladores, herramientas de cobertura y similares. Su comportamiento es parte de la plataforma de implementación, en lugar de parte de la definición del lenguaje y, por lo tanto, es posible que no esté disponible en todas las implementaciones de Python.

`sys.getwindowsversion()`

Retorna una tupla con nombre que describe la versión de Windows que se está ejecutando actualmente. Los elementos nombrados son *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, *product_type* y *platform_version*. *service_pack* contiene una cadena de caracteres, *platform_version* una tupla de 3 y todos los demás valores son números enteros. También se puede acceder a los componentes por su nombre, por lo que `sys.getwindowsversion()[0]` es equivalente a `sys.getwindowsversion().major`. Para compatibilidad con versiones anteriores, solo los primeros 5 elementos se pueden recuperar mediante la indexación.

platform será 2 (`VER_PLATFORM_WIN32_NT`).

product_type puede ser uno de los siguientes valores:

Constante	Significado
1 (VER_NT_WORKSTATION)	El sistema es una estación de trabajo.
2 (VER_NT_DOMAIN_CONTROLLER)	El sistema es un controlador de dominio.
3 (VER_NT_SERVER)	El sistema es un servidor, pero no un controlador de dominio.

Esta función envuelve la función Win32 `GetVersionEx()`; consulte la documentación de Microsoft en `OSVERSIONINFOEX()` para obtener más información sobre estos campos.

platform_version returns the major version, minor version and build number of the current operating system, rather than the version that is being emulated for the process. It is intended for use in logging rather than for feature detection.

Nota: *platform_version* derives the version from kernel32.dll which can be of a different version than the OS version. Please use *platform* module for achieving accurate OS version.

Disponibilidad: Windows.

Distinto en la versión 3.2: Cambiada a una tupla con nombre y agregado *service_pack_minor*, *service_pack_major*, *suite_mask*, y *product_type*.

Distinto en la versión 3.6: Agregado *platform_version*

`sys.get_asyncgen_hooks()`

Retorna un objeto *asyncgen_hooks*, que es similar a *namedtuple* de la forma (*firstiter*, *finalizer*), donde se espera que *firstiter* y *finalizer* sean `None` o funciones que toman un *asynchronous generator iterator* como argumento, y se utilizan para programar la finalización de un generador asíncrono mediante un bucle de eventos.

Nuevo en la versión 3.6: Ver [PEP 525](#) para más detalles.

Nota: Esta función se ha añadido de forma provisional (consulte [PEP 411](#) para obtener más detalles).

`sys.get_coroutine_origin_tracking_depth()`

Obtiene la profundidad de seguimiento del origen de la co-rutina actual, según lo establecido por *set_coroutine_origin_tracking_depth()*.

Nuevo en la versión 3.7.

Nota: Esta función se ha añadido de forma provisional (consulte [PEP 411](#) para obtener más detalles). Úsela sólo para fines de depuración.

`sys.hash_info`

A *named tuple* dando parámetros de la implementación de hash numérico. Para obtener más detalles sobre el hash de tipos numéricos, consulte *Calculo del hash de tipos numéricos*.

atributo	explicación
<code>width</code>	ancho en bits usado para valores hash
<code>modulus</code>	primer módulo P utilizado para el esquema de hash numérico
<code>inf</code>	valor hash retornado para un infinito positivo
<code>nan</code>	valor hash retornado para un nan
<code>imag</code>	multiplicador utilizado para la parte imaginaria de un número complejo
<code>algorithm</code>	nombre del algoritmo para el hash de str, bytes y memoryview
<code>hash_bits</code>	tamaño de salida interno del algoritmo hash
<code>seed_bits</code>	tamaño de la clave semilla del algoritmo hash

Nuevo en la versión 3.2.

Distinto en la versión 3.4: Agregado *algoritmo*, *hash_bits* y *seed_bits*

sys.hexversion

El número de versión codificado como un solo entero. Se garantiza que esto aumentará con cada versión, incluido el soporte adecuado para versiones que no son de producción. Por ejemplo, para probar que el intérprete de Python es al menos la versión 1.5.2, use:

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

Esto se llama `hexversion` ya que solo parece realmente significativo cuando se ve como el resultado de pasarlo a la función incorporada `hex()`. El *named tuple* `sys.version_info` puede usarse para una codificación más amigable para los humanos de la misma información.

Se pueden encontrar más detalles de `hexversion` en `apiabiversion`.

sys.implementation

Un objeto que contiene información sobre la implementación del intérprete de Python en ejecución. Los siguientes atributos deben existir en todas las implementaciones de Python.

`name` es el identificador de la implementación, por ejemplo `'cpython'`. La cadena de caracteres real está definida por la implementación de Python, pero se garantiza que estará en minúsculas.

`version` es una tupla con nombre, en el mismo formato que `sys.version_info`. Representa la versión de la implementación de Python. Esto tiene un significado distinto de la versión específica del lenguaje de Python al que se ajusta el intérprete que se está ejecutando actualmente, que representa `sys.version_info`. Por ejemplo, para PyPy 1.8 `sys.implementation.version` podría ser `sys.version_info(1, 8, 0, 'final', 0)`, mientras que `sys.version_info` sería `sys.version_info(2, 7, 2, 'final', 0)`. Para CPython tienen el mismo valor, ya que es la implementación de referencia.

`hexversion` es la versión de implementación en formato hexadecimal, como `sys.hexversion`.

`cache_tag` es la etiqueta utilizada por la maquinaria de importación en los nombres de archivo de los módulos almacenados en caché. Por convención, sería una combinación del nombre y la versión de la implementación, como `'cpython-33'`. Sin embargo, una implementación de Python puede usar algún otro valor si corresponde. Si `cache_tag` está configurado como `None`, indica que el almacenamiento en caché del módulo debe estar deshabilitado.

`sys.implementation` puede contener atributos adicionales específicos de la implementación de Python. Estos atributos no estándar deben comenzar con un guion bajo y no se describen aquí. Independientemente de su contenido, `sys.implementation` no cambiará durante la ejecución del intérprete, ni entre las versiones de implementación. (Sin embargo, puede cambiar entre las versiones del lenguaje Python). Consulte [PEP 421](#) para obtener más información.

Nuevo en la versión 3.3.

Nota: La adición de nuevos atributos obligatorios debe pasar por el proceso normal de PEP. Consulte [PEP 421](#) para obtener más información.

sys.int_info

Un *named tuple* que contiene información sobre la representación interna de Python de los enteros. Los atributos son de solo lectura.

Atributo	Explicación
<code>bits_per_digit</code>	número de bits retenidos en cada dígito. Los enteros de Python se almacenan internamente en la base <code>2**int_info.bits_per_digit</code>
<code>sizeof_digit</code>	tamaño en bytes del tipo C utilizado para representar un dígito
<code>default_max_str_digits</code>	default value for <code>sys.get_int_max_str_digits()</code> when it is not otherwise explicitly configured.
<code>str_digits_check_threshold</code>	minimum non-zero value for <code>sys.set_int_max_str_digits()</code> , <code>PYTHONINTMAXSTRDIGITS</code> , or <code>-X int_max_str_digits</code> .

Nuevo en la versión 3.1.

Distinto en la versión 3.8.14: Added `default_max_str_digits` and `str_digits_check_threshold`.

`sys.__interactivehook__`

Cuando existe este atributo, su valor se llama automáticamente (sin argumentos) cuando se lanza el intérprete en modo interactivo. Esto se hace después de leer el archivo `PYTHONSTARTUP`, para que pueda establecer este enlace allí. El módulo *site establece este*.

Lanza un *auditing event* `cpython.run_interactivehook` con el argumento `hook`.

Nuevo en la versión 3.4.

`sys.intern(string)`

Ingresa *string* en la tabla de cadenas de caracteres «internadas» y retorna la cadena de caracteres interna, que es *string* en sí misma o una copia. Internar cadenas de caracteres es útil para obtener un poco de rendimiento en la búsqueda de diccionario: si las claves en un diccionario están internadas y la clave de búsqueda está interna, las comparaciones de claves (después del hash) se pueden realizar mediante una comparación de punteros en lugar de una comparación de cadenas. Normalmente, los nombres utilizados en los programas de Python se internan automáticamente, y los diccionarios utilizados para contener los atributos de módulo, clase o instancia tienen claves internas.

Las cadenas de caracteres internas no son inmortales; debe mantener una referencia al valor de retorno de `intern()` para beneficiarse de él.

`sys.is_finalizing()`

Retorna *True* si el intérprete de Python es *shutting down*, *False* en caso contrario.

Nuevo en la versión 3.5.

`sys.last_type`

`sys.last_value`

`sys.last_traceback`

Estas tres variables no siempre están definidas; se establecen cuando no se maneja una excepción y el intérprete imprime un mensaje de error y un seguimiento de la pila. Su uso previsto es permitir que un usuario interactivo importe un módulo depurador y realice una depuración post-mortem sin tener que volver a ejecutar el comando que provocó el error. (El uso típico es `import pdb; pdb.pm()` para ingresar al depurador post-mortem; consulte módulo *pdb* para obtener más información).

El significado de las variables es el mismo que el de los valores retornados de `exc_info()` arriba.

`sys.maxsize`

Un entero que da el valor máximo que puede tomar una variable de tipo `Py_ssize_t`. Suele ser `2**31 - 1` en una plataforma de 32 bits y `2**63 - 1` en una plataforma de 64 bits.

`sys.maxunicode`

Un número entero que da el valor del punto de código Unicode más grande, es decir, `1114111` (`0x10FFFF` en hexadecimal).

Distinto en la versión 3.3: Antes [PEP 393](#), `sys.maxunicode` solía ser `0xFFFF` o `0x10FFFF`, dependiendo de la opción de configuración que especificaba si los caracteres Unicode se almacenaban como UCS-2 o UCS-4.

`sys.meta_path`

Una lista de objetos *meta path finder* que tienen sus métodos `find_spec()` llamados para ver si uno de los objetos puede encontrar el módulo que se va a importar. Se llama al método `find_spec()` con al menos el nombre absoluto del módulo que se está importando. Si el módulo que se va a importar está contenido en un paquete, el atributo del paquete principal `__path__` se pasa como segundo argumento. El método retorna *module spec*, o `None` si no se puede encontrar el módulo.

Ver también:

`importlib.abc.MetaPathFinder` La clase base abstracta que define la interfaz de los objetos del buscador en *meta_path*.

`importlib.machinery.ModuleSpec` La clase concreta que `find_spec()` debería retornar instancias de.

Distinto en la versión 3.4: *Especificaciones del módulo* se introdujeron en Python 3.4, por [PEP 451](#). Las versiones anteriores de Python buscaban un método llamado `find_module()`. Esto todavía se llama como una alternativa si una entrada *meta_path* no tiene un método `find_spec()`.

`sys.modules`

Este es un diccionario que asigna los nombres de los módulos a los módulos que ya se han cargado. Esto se puede manipular para forzar la recarga de módulos y otros trucos. Sin embargo, reemplazar el diccionario no necesariamente funcionará como se esperaba y eliminar elementos esenciales del diccionario puede hacer que Python falle.

`sys.path`

Una lista de cadenas de caracteres que especifica la ruta de búsqueda de módulos. Inicializado desde la variable de entorno `PYTHONPATH`, más un valor predeterminado que depende de la instalación.

Como se inicializó al iniciar el programa, el primer elemento de esta lista, `path[0]`, es el directorio que contiene el script que se utilizó para invocar al intérprete de Python. Si el directorio de la secuencia de comandos no está disponible (por ejemplo, si el intérprete se invoca de forma interactiva o si la secuencia de comandos se lee desde la entrada estándar), `path[0]` es la cadena de caracteres vacía, que dirige a Python a buscar módulos en el directorio actual primero. Observe que el directorio del script se inserta *antes* de las entradas insertadas como resultado de `PYTHONPATH`.

Un programa es libre de modificar esta lista para sus propios fines. Solo se deben agregar cadenas de caracteres y bytes a `sys.path`; todos los demás tipos de datos se ignoran durante la importación.

Ver también:

Módulo *site* Esto describe cómo usar archivos `.pth` para extender `sys.path`.

`sys.path_hooks`

Una lista de invocables que toman un argumento de ruta para intentar crear un *finder* para la ruta. Si se puede crear un buscador, el invocable debe retornar; de lo contrario, lanza `ImportError`.

Especificado originalmente en [PEP 302](#).

`sys.path_importer_cache`

Un diccionario que actúa como caché para objetos *finder*. Las claves son rutas que se han pasado a `sys.path_hooks` y los valores son los buscadores que se encuentran. Si una ruta es una ruta de sistema de archivos válida pero no se encuentra ningún buscador en `sys.path_hooks`, entonces se almacena `None`.

Especificado originalmente en [PEP 302](#).

Distinto en la versión 3.3: `None` se almacena en lugar de `imp.NullImporter` cuando no se encuentra ningún buscador.

`sys.platform`

Esta cadena de caracteres contiene un identificador de plataforma que se puede usar para agregar componentes específicos de la plataforma a `sys.path`, por ejemplo.

Para los sistemas Unix, excepto en Linux y AIX, este es el nombre del sistema operativo en minúsculas como lo retorna `uname -s` con la primera parte de la versión retornada por `uname -r` adjunta, por ejemplo `'sunos5'` o `'freebsd8'`, *en el momento en que se construyó Python*. A menos que desee probar una versión específica del sistema, se recomienda utilizar el siguiente idioma:

```
if sys.platform.startswith('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
elif sys.platform.startswith('aix'):
    # AIX-specific code here...
```

Para otros sistemas, los valores son:

Sistema	valor platform
AIX	'aix'
Linux	'linux'
Windows	'win32'
Windows/Cygwin	'cygwin'
macOS	'darwin'

Distinto en la versión 3.3: En Linux, `sys.platform` ya no contiene la versión principal. Siempre es `'linux'`, en lugar de `'linux2'` o `'linux3'`. Dado que las versiones anteriores de Python incluyen el número de versión, se recomienda utilizar siempre el idioma `startswith` presentado anteriormente.

Distinto en la versión 3.8: En AIX, `sys.platform` ya no contiene la versión principal. Siempre es `'aix'`, en lugar de `'aix5'` o `'aix7'`. Dado que las versiones anteriores de Python incluyen el número de versión, se recomienda utilizar siempre el idioma `startswith` presentado anteriormente.

Ver también:

`os.name` tiene una granularidad más gruesa. `os.uname()` proporciona información de versión dependiente del sistema.

El módulo `platform` proporciona comprobaciones detalladas de la identidad del sistema.

`sys.prefix`

Una cadena de caracteres que proporciona el prefijo de directorio específico del sitio donde se instalan los archivos Python independientes de la plataforma; por defecto, esta es la cadena de caracteres `'/usr/local'`. Esto se puede configurar en el momento de la compilación con el argumento `--prefix` del script **configure**. La colección principal de módulos de la biblioteca de Python se instala en el directorio `prefijo/lib/pythonXY` mientras que los archivos de encabezado independientes de la plataforma (todos excepto `pyconfig.h`) se almacenan en `prefijo/include/pythonXY`, donde `XY` es el número de versión de Python, por ejemplo, `3.2`.

Nota: Si un *virtual environment* está en efecto, este valor se cambiará en `site.py` para apuntar al entorno virtual. El valor para la instalación de Python seguirá estando disponible a través de `base_prefix`.

`sys.ps1`

`sys.ps2`

Cadenas de caracteres que especifican el indicador principal y secundario del intérprete. Estos solo se definen si el intérprete está en modo interactivo. Sus valores iniciales en este caso son `'>>>'` y `'...'`. Si se asigna un objeto que no es una cadena a cualquiera de las variables, su `str()` se vuelve a evaluar cada vez que el intérprete se prepara para leer un nuevo comando interactivo; esto se puede utilizar para implementar un mensaje dinámico.

`sys.setcheckinterval(interval)`

Configure el «intervalo de verificación» del intérprete. Este valor entero determina la frecuencia con la que el intérprete comprueba cosas periódicas como cambios de hilo y manejadores de señales. El valor predeterminado es `100`, lo que significa que la verificación se realiza cada 100 instrucciones virtuales de Python.

Establecerlo en un valor mayor puede aumentar el rendimiento de los programas que utilizan subprocesos. Establecerlo en un valor ≤ 0 verifica cada instrucción virtual, maximizando la capacidad de respuesta y la sobrecarga.

Obsoleto desde la versión 3.2: Esta función ya no tiene ningún efecto, ya que se ha reescrito la lógica interna para el cambio de hilo y las tareas asíncronas. Utilice `setswitchinterval()` en su lugar.

`sys.setdlopenflags(n)`

Establece los indicadores utilizados por el intérprete para llamadas `dlopen()`, como cuando el intérprete carga módulos de extensión. Entre otras cosas, esto permitirá una resolución diferida de símbolos al importar un módulo, si se llama como `sys.setdlopenflags(0)`. Para compartir símbolos entre módulos de extensión, llame como `sys.setdlopenflags(os.RTLD_GLOBAL)`. Los nombres simbólicos para los valores de las banderas se pueden encontrar en el módulo `os` (constantes `RTLD_XXX`, por ejemplo `os.RTLD_LAZY`).

Disponibilidad: Unix.

`sys.set_int_max_str_digits(n)`

Set the *integer string conversion length limitation* used by this interpreter. See also `get_int_max_str_digits()`.

Nuevo en la versión 3.8.14.

`sys.setprofile(profilefunc)`

Configura la función de perfil del sistema, que le permite implementar un generador de perfiles de código fuente de Python en Python. Consulte el capítulo *Los perfiladores de Python* para obtener más información sobre el generador de perfiles de Python. La función de perfil del sistema se llama de manera similar a la función de seguimiento del sistema (ver `settrace()`), pero se llama con diferentes eventos, por ejemplo, no se llama para cada línea de código ejecutada (solo al llamar y regresar, pero el evento de retorno se informa incluso cuando se ha establecido una excepción). La función es específica de subprocesos, pero el generador de perfiles no tiene forma de conocer los cambios de contexto entre subprocesos, por lo que no tiene sentido usar esto en presencia de varios subprocesos. Además, su valor de retorno no se usa, por lo que simplemente puede retornar `None`. Un error en la función del perfil provocará su desarmado.

Las funciones de perfil deben tener tres argumentos: *frame*, *event* y *arg*. *frame* es el marco de pila actual. *event* es una cadena de caracteres: 'call', 'return', 'c_call', 'c_return', o 'c_exception'. *arg* depende del tipo de evento.

Lanza un *auditing event* `sys.setprofile` sin argumentos.

Los eventos tienen el siguiente significado:

'call' Se llama a una función (o se ingresa algún otro bloque de código). Se llama a la función de perfil; *arg* es `None`.

'return' Una función (u otro bloque de código) está a punto de regresar. Se llama a la función de perfil; *arg* es el valor que se retornará, o `None` si el evento es causado por una excepción.

'c_call' Una función C está a punto de ser llamada. Esta puede ser una función de extensión o una incorporada. *arg* es el objeto de función de C.

'c_return' Ha vuelto una función C. *arg* es el objeto de función de C.

'c_exception' Una función C ha generado una excepción. *arg* es el objeto de función de C.

`sys.setrecursionlimit(limit)`

Establece la profundidad máxima de la pila de intérpretes de Python en *limit*. Este límite evita que la recursividad infinita cause un desbordamiento de la pila C y bloquee Python.

El límite más alto posible depende de la plataforma. Un usuario puede necesitar establecer un límite más alto cuando tiene un programa que requiere una recursividad profunda y una plataforma que admite un límite más alto. Esto debe hacerse con cuidado, ya que un límite demasiado alto puede provocar un accidente.

Si el nuevo límite es demasiado bajo en la profundidad de recursividad actual, se genera una excepción `RecursionError`.

Distinto en la versión 3.5.1: A la excepción `RecursionError` ahora se lanza si el nuevo límite es demasiado bajo en la profundidad de recursión actual.

`sys.setswitchinterval(interval)`

Establece el intervalo de cambio de hilo del intérprete (en segundos). Este valor de punto flotante determina la duración ideal de los «segmentos de tiempo» asignados a los subprocesos de Python que se ejecutan simultáneamente. Tenga en cuenta que el valor real puede ser mayor, especialmente si se utilizan funciones o métodos internos de larga duración. Además, qué hilo se programa al final del intervalo es decisión del sistema operativo. El intérprete no tiene su propio programador.

Nuevo en la versión 3.2.

`sys.settrace(tracefunc)`

Configura la función de seguimiento del sistema, que le permite implementar un depurador de código fuente de Python en Python. La función es específica del hilo; para que un depurador admita múltiples subprocesos, debe registrar una función de seguimiento usando `settrace()` para cada subproceso que se depura o use `threading.settrace()`.

Las funciones de seguimiento deben tener tres argumentos: *frame*, *event* y *arg*. *frame* es el marco de pila actual. *event* es una cadena de caracteres: 'call', 'line', 'return', 'exception' or 'opcode'. *arg* depende del tipo de evento.

La función de rastreo se invoca (con *event* establecido en 'call') cada vez que se ingresa un nuevo ámbito local; debe retornar una referencia a una función de rastreo local que se usará para el nuevo alcance, o `None` si no se debe rastrear el alcance.

La función de seguimiento local debe retornar una referencia a sí misma (o a otra función para realizar un seguimiento adicional en ese ámbito), o `None` para desactivar el seguimiento en ese ámbito.

Si se produce algún error en la función de seguimiento, se desarmará, al igual que se llama a `settrace(None)`.

Los eventos tienen el siguiente significado:

'**call**' Se llama a una función (o se ingresa algún otro bloque de código). Se llama a la función de rastreo global; *arg* es `None`; el valor de retorno especifica la función de rastreo local.

'**line**' El intérprete está a punto de ejecutar una nueva línea de código o volver a ejecutar la condición de un bucle. Se llama a la función de rastreo local; *arg* es `None`; el valor de retorno especifica la nueva función de rastreo local. Consulte `Objects/lnotab_notes.txt` para obtener una explicación detallada de cómo funciona. Los eventos por línea se pueden deshabilitar para un marco estableciendo `f_trace_lines` to `False` en ese marco.

'**return**' Una función (u otro bloque de código) está a punto de regresar. Se llama a la función de rastreo local; *arg* es el valor que se retornará, o `None` si el evento es causado por una excepción. Se ignora el valor de retorno de la función de seguimiento.

'**exception**' Ha ocurrido una excepción. Se llama a la función de rastreo local; *arg* es una tupla (`exception`, `value`, `traceback`); el valor de retorno especifica la nueva función de rastreo local.

'**opcode**' El intérprete está a punto de ejecutar un nuevo código de operación (ver *dis* para detalles del código de operación). Se llama a la función de rastreo local; *arg* es `None`; el valor de retorno especifica la nueva función de rastreo local. Los eventos por código de operación no se emiten de forma predeterminada: deben solicitarse explícitamente configurando `f_trace_opcodes` en `True` en el marco.

Tenga en cuenta que como una excepción se propaga a lo largo de la cadena de llamadas de funciones, se lanza un evento de 'excepción' en cada nivel.

Para un uso más detallado, es posible establecer una función de seguimiento asignando `frame.f_trace = tracefunc` explícitamente, en lugar de confiar en que se establezca indirectamente a través del valor de retorno de una función de seguimiento ya instalada. Esto también es necesario para activar la función de seguimiento en el marco actual, lo cual `settrace()` no funciona. Tenga en cuenta que para que esto funcione, se debe haber instalado una función de rastreo global con `settrace()` para habilitar la maquinaria de rastreo en tiempo de ejecución, pero no necesita ser la misma función de rastreo (por ejemplo, podría ser

una función de rastreo de sobrecarga baja que simplemente retorna `None` para deshabilitarse inmediatamente en cada cuadro).

Para obtener más información sobre los objetos de código y marco, consulte `types`.

Lanza un *auditing event* `sys.settrace` sin argumentos.

CPython implementation detail: La función `settrace()` está pensada únicamente para implementar depuradores, perfiladores, herramientas de cobertura y similares. Su comportamiento es parte de la plataforma de implementación, en lugar de parte de la definición del lenguaje y, por lo tanto, es posible que no esté disponible en todas las implementaciones de Python.

Distinto en la versión 3.7: Se agregó el tipo de evento `'opcode'`; atributos `f_trace_lines` y `f_trace_opcodes` agregados a los marcos (*frames*)

`sys.set_asyncgen_hooks` (*firstiter*, *finalizer*)

Acepta dos argumentos de palabras clave opcionales que son invocables que aceptan un *asynchronous generator iterator* como argumento. Se llamará al *firstiter* invocable cuando se repita un generador asincrónico por primera vez. Se llamará al *finalizer* cuando un generador asincrónico esté a punto de ser recolectado como basura.

Lanza un *auditing event* `sys.set_asyncgen_hooks_firstiter` sin argumentos.

Lanza un *auditing event* `sys.set_asyncgen_hooks_finalizer` sin argumentos.

Se lanzan dos eventos de auditoría porque la API subyacente consta de dos llamadas, cada una de las cuales debe generar su propio evento.

Nuevo en la versión 3.6: Ver [PEP 525](#) para más detalles, y para un ejemplo de referencia de un método *finalizer* ver la implementación de `asyncio.Loop.shutdown_asyncgens` en [Lib/asyncio/base_events.py](#)

Nota: Esta función se ha añadido de forma provisional (consulte [PEP 411](#) para obtener más detalles).

`sys.set_coroutine_origin_tracking_depth` (*depth*)

Permite habilitar o deshabilitar el seguimiento de origen de rutina. Cuando está habilitado, el atributo `cr_origin` en los objetos de corrutina contendrá una tupla de (nombre de archivo, número de línea, nombre de función) tuplas que describen el rastreo donde se creó el objeto de corrutina, con la llamada más reciente primero. Cuando esté deshabilitado, `cr_origin` será `None`.

Para habilitarlo, pase un valor de *depth* mayor que cero; esto establece el número de fotogramas cuya información será capturada. Para deshabilitar, pase *depth* a cero.

Esta configuración es específica de cada hilo.

Nuevo en la versión 3.7.

Nota: Esta función se ha añadido de forma provisional (consulte [PEP 411](#) para obtener más detalles). Úsela sólo para fines de depuración.

`sys._enablelegacywindowsfsencoding` ()

Cambia la codificación predeterminada del sistema de archivos y el modo de errores a “mbcs” y “replace” respectivamente, para mantener la coherencia con las versiones de Python anteriores a la 3.6.

Esto es equivalente a definir la variable de entorno `PYTHONLEGACYWINDOWSFSENCODING` antes de iniciar Python.

Disponibilidad: Windows.

Nuevo en la versión 3.6: Consulte [PEP 529](#) para obtener más detalles.

`sys.stdin`

`sys.stdout`

`sys.stderr`

Objetos de archivo utilizado por el intérprete para entradas, salidas y errores estándar:

- `stdin` se usa para todas las entradas interactivas (incluidas las llamadas a `input()`);

- `stdout` se usa para la salida de `print()` y `expression` y para las solicitudes de `input()`;
- Las propias indicaciones del intérprete y sus mensajes de error van a `stderr`.

Estos flujos son regulares *archivos de texto* como los retornados por la función `open()`. Sus parámetros se eligen de la siguiente manera:

- La codificación de caracteres depende de la plataforma. Las plataformas que no son de Windows utilizan la codificación local (consulte `locale.getpreferredencoding()`).

En Windows, se usa UTF-8 para el dispositivo de consola. Los dispositivos que no son caracteres, como archivos de disco y tuberías, utilizan la codificación de la configuración regional del sistema (es decir, la página de códigos ANSI). Los dispositivos de caracteres que no son de consola, como NUL (es decir, donde `isatty()` retorna `True`) utilizan el valor de las páginas de códigos de entrada y salida de la consola al inicio, respectivamente para `stdin` y `stdout/stderr`. Este valor predeterminado es la codificación de la configuración regional del sistema si el proceso no se adjunta inicialmente a una consola.

El comportamiento especial de la consola se puede anular configurando la variable de entorno `PYTHONLEGACYWINDOWSSTDIO` antes de iniciar Python. En ese caso, las páginas de códigos de la consola se utilizan como para cualquier otro dispositivo de caracteres.

En todas las plataformas, puede anular la codificación de caracteres configurando la variable de entorno `PYTHONIOENCODING` antes de iniciar Python o usando la nueva `-X utf8` opción de línea de comando y `PYTHONUTF8` variable de entorno. Sin embargo, para la consola de Windows, esto solo se aplica cuando `PYTHONLEGACYWINDOWSSTDIO` también está configurado.

- Cuando son interactivas, las secuencias `stdout` y `stderr` tienen búfer de línea. De lo contrario, se almacenan en búfer en bloques como archivos de texto normales. Puede anular este valor con la opción de línea de comandos `-u`.

Nota: Para escribir o leer datos binarios desde/hacia los flujos estándar, use el objeto binario subyacente *buffer*. Por ejemplo, para escribir bytes en `stdout`, use `sys.stdout.buffer.write(b'abc')`.

Sin embargo, si está escribiendo una biblioteca (y no controla en qué contexto se ejecutará su código), tenga en cuenta que las transmisiones estándar pueden reemplazarse con objetos similares a archivos como *io.StringIO* que no admiten el atributo `buffer`.

```
sys.__stdin__
sys.__stdout__
sys.__stderr__
```

Estos objetos contienen los valores originales de `stdin`, `stderr` y `stdout` al inicio del programa. Se utilizan durante la finalización y podrían ser útiles para imprimir en el flujo estándar real sin importar si el objeto `sys.std*` ha sido redirigido.

También se puede utilizar para restaurar los archivos reales a objetos de archivo de trabajo conocidos en caso de que se hayan sobrescrito con un objeto roto. Sin embargo, la forma preferida de hacer esto es guardar explícitamente la secuencia anterior antes de reemplazarla y restaurar el objeto guardado.

Nota: En algunas condiciones, `stdin`, `stdout` y `stderr`, así como los valores originales `__stdin__`, `__stdout__` y `__stderr__` pueden ser `None`. Suele ser el caso de las aplicaciones GUI de Windows que no están conectadas a una consola y las aplicaciones Python que comienzan con **pythonw**.

```
sys.thread_info
```

Un *named tuple* que contiene información sobre la implementación del hilo.

Atributo	Explicación
<code>name</code>	Nombre de la implementación de hilos (<i>thread implementation</i>): <ul style="list-style-type: none"> • <code>'nt'</code>: hilos de Windows • <code>'pthread'</code>: hilos de POSIX • <code>'solaris'</code>: hilos de Solaris
<code>lock</code>	Nombre de la implementación de bloqueo (<i>lock implementation</i>): <ul style="list-style-type: none"> • <code>'semaphore'</code>: un bloqueo que utiliza un semáforo • <code>'mutex+cond'</code>: un bloqueo que utiliza un mutex y una variable de condición • <code>None</code> si esta información es desconocida
<code>version</code>	Nombre y versión de la biblioteca de subprocessos. Es una cadena de caracteres, o <code>None</code> si se desconoce esta información.

Nuevo en la versión 3.3.

`sys.tracebacklimit`

Cuando esta variable se establece en un valor entero, determina el número máximo de niveles de información de rastreo que se imprime cuando ocurre una excepción no controlada. El valor predeterminado es 1000. Cuando se establece en 0 o menos, toda la información de rastreo se suprime y solo se imprimen el tipo y el valor de excepción.

`sys.unraisablehook` (*unraisable*, /)

Maneja una excepción que no se puede lanzar.

Se llama cuando se ha producido una excepción pero no hay forma de que Python la maneje. Por ejemplo, cuando un destructor lanza una excepción o durante la recolección de basura (`gc.collect()`).

El argumento *unraisable* (no lanzable) tiene los siguientes atributos:

- *exc_type*: Tipo de excepción.
- *exc_value*: Valor de excepción, puede ser `None`.
- *exc_traceback*: Rastreo de excepción, puede ser `None`.
- *err_msg*: Mensaje de error, puede ser `None`.
- *objeto*: Objeto que causa la excepción, puede ser `None`.

Los formatos de gancho predeterminados *err_msg* y *object* como: `f'{err_msg}:{object!R}'`; use el mensaje de error «Excepción ignorada en» si *err_msg* es `None`.

`sys.unraisablehook()` se puede anular para controlar cómo se manejan las excepciones que no se pueden evaluar.

Almacenar *exc_value* usando un gancho personalizado puede crear un ciclo de referencia. Debe borrarse explícitamente para romper el ciclo de referencia cuando la excepción ya no sea necesaria.

Almacenar *object* usando un gancho personalizado puede resucitarlo si se establece en un objeto que se está finalizando. Evite almacenar *object* después de que se complete el gancho personalizado para evitar resucitar objetos.

Véase también `excepthook()` que maneja excepciones no capturadas.

Lanza un evento de auditoría `sys.unraisablehook` con argumentos *hook*, *unraisable* cuando ocurre una excepción que no se puede manejar. El objeto *no lanzable* es el mismo que se pasará al gancho. Si no se ha colocado ningún gancho, *hook* puede ser `None`.

Nuevo en la versión 3.8.

`sys.version`

Una cadena de caracteres que contiene el número de versión del intérprete de Python más información adicional sobre el número de compilación y el compilador utilizado. Esta cadena se muestra cuando se inicia el intérprete

interactivo. No extraiga información de la versión de él, en su lugar, use `version_info` y las funciones proporcionadas por el módulo `platform`.

`sys.api_version`

La versión de API C para este intérprete. Los programadores pueden encontrar esto útil al depurar conflictos de versiones entre Python y módulos de extensión.

`sys.version_info`

Una tupla que contiene los cinco componentes del número de versión: *major*, *minor*, *micro*, *releaselevel*, y *serial*. Todos los valores excepto *releaselevel* son números enteros; el nivel de lanzamiento es 'alpha', 'beta', 'candidate', o 'final'. El valor de `version_info` correspondiente a la versión 2.0 de Python es (2, 0, 0, 'final', 0). También se puede acceder a los componentes por su nombre, por lo que `sys.version_info[0]` es equivalente a `sys.version_info.major` y así sucesivamente.

Distinto en la versión 3.1: Se agregaron atributos de componentes con nombre.

`sys.warnoptions`

Este es un detalle de implementación del marco de advertencias; No modifique este valor. Consulte el módulo `warnings` para obtener más información sobre el marco de advertencias.

`sys.winver`

El número de versión utilizado para formar claves de registro en plataformas Windows. Esto se almacena como recurso de cadena 1000 en la DLL de Python. El valor son normalmente los primeros tres caracteres de `version`. Se proporciona en el módulo `sys` con fines informativos; la modificación de este valor no tiene ningún efecto sobre las claves de registro que utiliza Python.

Disponibilidad: Windows.

`sys._xoptions`

Un diccionario de las diversas flags específicas de la implementación pasa a través de la opción de línea de comandos `-X`. Los nombres de las opciones se asignan a sus valores, si se dan explícitamente, o a `True`. Ejemplo:

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

CPython implementation detail: Esta es una forma específica de CPython de acceder a las opciones que se pasan a través de `-X`. Otras implementaciones pueden exportarlos a través de otros medios, o no exportarlos.

Nuevo en la versión 3.2.

Citas

29.2 `sysconfig` — Proporciona acceso a la información de configuración de Python

Nuevo en la versión 3.2.

Código fuente: `Lib/sysconfig.py`

El módulo `sysconfig` proporcionado acceso a la información de configuración de Python, como la lista de rutas de instalación y las variables de configuración relevantes para la plataforma actual.

29.2.1 Variables de configuración

Una distribución de Python contiene un `Makefile` y un archivo de cabecera `pyconfig.h` que son necesarios para construir tanto el propio binario de Python como las extensiones C de terceros compiladas usando *distutils*.

sysconfig coloca todas las variables que se encuentran en estos archivos un diccionario al que se puede acceder usando `get_config_vars()` o `get_config_var()`.

Tenga en cuenta que en Windows, es un conjunto mucho más pequeño.

`sysconfig.get_config_vars(*args)`

Sin argumentos, retorna un diccionario de todas las variables de configuración relevantes para la plataforma actual.

Con argumentos, retorna una lista de valores que resultan de buscar cada argumento en el diccionario de variables de configuración.

Por cada argumento, si no se encuentra el valor, retorna `None`.

`sysconfig.get_config_var(name)`

Retorna el valor de un solo nombre de variable (*name*). Equivalente a `get_config_vars().get(name)`.

Si no se encuentra *name*, retorna `None`.

Ejemplos de uso:

```
>>> 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++']
```

29.2.2 Rutas de instalación

Python usa un esquema de instalación que difiere según la plataforma y en las opciones de instalación. Estos esquemas son almacenados en *sysconfig* con identificadores únicos basados en el valor retornado por *os.name*.

Cada nuevo componente que se instala usando *distutils* o un sistema basado en *Distutils* seguirá el mismo esquema para copiar su archivo en los lugares correctos.

Python actualmente admite siete esquemas:

- *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*: esquema para plataformas NT como Windows.
- *nt_user*: esquema para plataformas NT, cuando se usa la opción *user*

Cada esquema está compuesto por una serie de rutas y cada ruta tiene un identificador único. Python actualmente usa ocho rutas:

- *stdlib*: directorio que contiene los archivos de la biblioteca estándar de Python que no son específicos de la plataforma.
- *platstdlib*: directorio que contiene los archivos de la biblioteca estándar de Python que son específicos de la plataforma.

- *platlib*: directorio para archivos específicos del sitio, específicos de la plataforma.
- *purelib*: directorio para archivos específicos del sitio, no específicos de la plataforma.
- *include*: directorio para archivos de encabezado no específicos de plataforma.
- *platinclude*: directorio para archivos de encabezado específicos de la plataforma.
- *scripts*: directorio para archivos de script.
- *data*: directorio para archivos de datos.

sysconfig proporciona algunas funciones para determinar estas rutas.

`sysconfig.get_scheme_names()`

Retorna una tupla que contiene todos los esquemas admitidos actualmente en *sysconfig*.

`sysconfig.get_path_names()`

Retorna una tupla que contiene todos los nombres de rutas admitidos actualmente en *sysconfig*.

`sysconfig.get_path(name[, scheme[, vars[, expand]]])`

Retorna una ruta de instalación correspondiente a la ruta *name*, del esquema de instalación denominado *scheme*.

name tiene que ser un valor de la lista retornado por `get_path_names()`.

sysconfig almacena las rutas de instalación correspondientes a cada nombre de ruta, para cada plataforma, con variables que se expandirán. Por ejemplo, la ruta *stdlib* para el esquema *nt* es: {base}/Lib.

`get_path()` utilizará las variables retornadas por `get_config_vars()` para expandir la ruta. Todas las variables tienen valores predeterminados para cada plataforma, por lo que se puede llamar a esta función y obtener el valor predeterminado.

Si se proporciona el esquema (*scheme*), debe ser un valor de la lista retornada por `get_scheme_names()`. De lo contrario, se utiliza el esquema predeterminado para la plataforma actual.

Si se proporciona *vars*, debe ser un diccionario de variables que actualizará el retorno del diccionario mediante `get_config_vars()`.

Si *expand* se establece en `False`, la ruta no se expandirá usando las variables.

Si no se encuentra *name*, retorna `None`.

`sysconfig.get_paths([scheme[, vars[, expand]]])`

Retorna un diccionario que contiene todas las rutas de instalación correspondientes a un esquema de instalación. Consulte `get_path()` para obtener más información.

Si no se proporciona el esquema (*scheme*), utilizará el esquema predeterminado para la plataforma actual.

Si se proporciona *vars*, debe ser un diccionario de variables que actualizará el diccionario utilizado para expandir las rutas.

Si *expand* se establece en falso, las rutas no se expandirán.

Si *scheme* no es un esquema existente, `get_paths()` lanzará un `KeyError`.

29.2.3 Otras funciones

`sysconfig.get_python_version()`

Retorna el número de versión versión MAJOR.MINOR de Python como una cadena. Similar a '%d.%d' % sys.version_info[:2].

`sysconfig.get_platform()`

Retorna una cadena que identifica la plataforma actual.

Se utiliza principalmente para distinguir los directorios de compilación específicos de la plataforma y las distribuciones compiladas específicas de la plataforma. Por lo general se incluye el nombre y la versión del sistema operativo y la arquitectura (como lo proporciona "os.uname()"), aunque la información exacta incluida depende del sistema operativo: por ejemplo, en Linux, la versión del kernel no es particularmente importante.

Ejemplo de valores retornados:

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u

Windows retornará uno de:

- win-amd64 (Windows de 64 bits en AMD64, también conocido como x86_64, Intel64 y EM64T)
- win32 (todos los demás - específicamente se retorna `sys.platform`)

Mac OS X puede retornar:

- macosx-10.6-ppc
- macosx-10.4-ppc64
- macosx-10.3-i386
- macosx-10.4-fat

Para otras plataformas que no son POSIX, actualmente solo retorna `sys.platform`.

`sysconfig.is_python_build()`

Retorna True si el intérprete de Python en ejecución se compiló a partir de la fuente y se está ejecutando desde su ubicación compilada, y no desde una ubicación resultante de por ejemplo ejecutando `make install` o instalando a través de un instalador binario.

`sysconfig.parse_config_h(fp[, vars])`

Analiza un archivo de estilo `config.h`.

`fp` es un objeto similar a un archivo que apunta al archivo similar a `config.h`.

Se retorna un diccionario que contiene pares de nombre/valor. Si se pasa un diccionario opcional como un segundo argumento, se utiliza en lugar de un nuevo diccionario y se actualiza con los valores leídos en el archivo.

`sysconfig.get_config_h_filename()`

Retorna la ruta de `pyconfig.h`.

`sysconfig.get_makefile_filename()`

Retorna la ruta de `Makefile`.

29.2.4 Usando `sysconfig` como un script

Puedes usar `sysconfig` como un script con la opción `-m` de Python:

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
    data = "/usr/local"
    include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
    platinclude = "."
    platlib = "/usr/local/lib/python3.2/site-packages"
    platstdlib = "/usr/local/lib/python3.2"
    purelib = "/usr/local/lib/python3.2/site-packages"
    scripts = "/usr/local/bin"
    stdlib = "/usr/local/lib/python3.2"

Variables:
    AC_APPLE_UNIVERSAL_BUILD = "0"
```

(continué en la próxima página)

(proviene de la página anterior)

```
AIX_GENUINE_CPLUSPLUS = "0"
AR = "ar"
ARFLAGS = "rc"
...
```

Esta llamada imprimirá en la salida estándar la información retornada por `get_platform()`, `get_python_version()`, `get_path()` y `get_config_vars()`.

29.3 builtins — Objetos incorporados

Este módulo proporciona acceso directo a todos los identificadores “incorporados” de Python. Por ejemplo, `builtins.open` es el nombre completo de la función incorporada `open()`. Vea *Funciones Built-in* y *Constantes incorporadas* para documentación.

Este módulo normalmente no es accedido explícitamente por la mayoría de las aplicaciones, pero puede ser útil en módulos que provean objetos con el mismo nombre que un valor incorporado, pero que sea necesario también el incorporado con ese nombre. Por ejemplo, en un módulo que quiera implementar una función `open()` que envuelva la integrada `open()`, este módulo puede ser usado directamente:

```
import builtins

def open(path):
    f = builtins.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()

    # ...
```

Como un detalle de implementación, la mayoría de los módulos tienen el nombre `__builtins__` disponible como parte de sus globales. El valor de `__builtins__` es normalmente o este módulo o el valor del atributo `__dict__` de este módulo. Como este es un detalle de implementación, puede que no sea usado por implementaciones alternativas de Python.

29.4 __main__ — Entorno de script del nivel superior

`'__main__'` es el nombre del ámbito en el que se ejecuta el código de nivel superior. El atributo `__name__` de un módulo se establece igual a `'__main__'` cuando se lee desde una entrada estándar, un script o un prompt interactivo.

Un módulo puede descubrir si se está ejecutando o no en el ámbito principal al verificar su propio `__name__`, lo cual permite un idioma común para ejecutar código condicionalmente en un módulo cuando este se ejecuta como un script o con `python -m` pero no cuando este es importado:


```
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Para un paquete, se puede lograr el mismo efecto incluyendo un módulo `__main__.py`, cuyo contenido se ejecutara cuando el módulo se ejecute con `-m`.

29.5 warnings — Control de advertencias

Código fuente: [Lib/warnings.py](#)

Los mensajes de advertencia suelen emitirse en situaciones en las que es útil alertar al usuario de alguna condición en un programa, cuando esa condición (normalmente) no justifica que se haga una excepción y se termine el programa. Por ejemplo, se puede emitir una advertencia cuando un programa utiliza un módulo obsoleto.

Los programadores de Python emiten advertencias llamando a la función `warn()` definida en este módulo. (Los programadores de C usan `PyErr_WarnEx()`; ver [exceptionhandling](#) para más detalles)

Los mensajes de advertencia se escriben normalmente en `sys.stderr`, pero su disposición puede cambiarse de manera flexible, desde ignorar todas las advertencias hasta convertirlas en excepciones. La disposición de las advertencias puede variar en función de la *warning category*, el texto del mensaje de advertencia y la ubicación de la fuente donde se emite. Las repeticiones de una advertencia particular para la misma ubicación de la fuente son típicamente suprimidas.

El control de las advertencias consta de dos etapas: en primer lugar, cada vez que se emite una advertencia, se determina si se debe emitir un mensaje o no; en segundo lugar, si se debe emitir un mensaje, se le da formato y se imprime utilizando un gancho establecido por el usuario.

La determinación de si se debe emitir un mensaje de advertencia está controlada por el *warning filter*, que es una secuencia de reglas y acciones que coinciden. Se pueden añadir reglas al filtro llamando a `filterwarnings()` y restablecer su estado por defecto llamando a `resetwarnings()`.

La impresión de los mensajes de advertencia se realiza llamando a `showwarning()`, que puede ser anulado; la implementación por defecto de esta función da formato al mensaje llamando a `formatwarning()`, que también está disponible para su uso en implementaciones personalizadas.

Ver también:

`logging.captureWarnings()` permite manejar todas las advertencias con la infraestructura de registro estándar.

29.5.1 Categorías de advertencia

Hay una serie de excepciones incorporadas que representan categorías de advertencia. Esta categorización es útil para poder filtrar grupos de advertencias.

Aunque técnicamente se trata de *built-in exceptions*, están documentadas aquí, porque pertenecen al mecanismo de advertencias.

El código de usuario puede definir categorías de advertencia adicionales mediante la subclasificación de una de las categorías de advertencia estándar. Una categoría de advertencia siempre debe ser una subclase de la clase `Warning` class.

Actualmente están definidas las siguientes clases de categorías de advertencias:

Clase	Descripción
<i>Warning</i>	Esta es la clase principal para todas las clases que pertenecen a la categoría de advertencia. Es una subclase de <i>Exception</i> .
<i>UserWarning</i>	La categoría por defecto para <i>warn()</i> .
<i>DeprecationWarning</i>	Categoría principal para advertencias sobre características obsoletas cuando esas advertencias están destinadas a otros desarrolladores de Python (ignoradas por defecto, a menos que sean activadas por el código en <code>__main__</code>).
<i>SyntaxWarning</i>	Categoría principal para las advertencias sobre características sintácticas dudosas.
<i>RuntimeWarning</i>	Categoría principal para las advertencias sobre características dudosas de tiempo de ejecución.
<i>FutureWarning</i>	Categoría principal para las advertencias sobre características obsoletas cuando esas advertencias están destinadas a los usuarios finales de aplicaciones escritas en Python.
<i>PendingDeprecationWarning</i>	Categoría principal para las advertencias sobre las características que serán desaprobadas en el futuro (ignoradas por defecto).
<i>ImportWarning</i>	Categoría principal para las advertencias que se activan durante el proceso de importación de un módulo (se ignoran por defecto).
<i>UnicodeWarning</i>	Categoría principal para las advertencias relacionadas con Unicode.
<i>BytesWarning</i>	Categoría principal para las advertencias relacionadas con <i>bytes</i> y <i>bytearray</i> .
<i>ResourceWarning</i>	Categoría base para las advertencias relacionadas con el uso de los recursos.

Distinto en la versión 3.7: Anteriormente *DeprecationWarning* y *FutureWarning* se distinguían en función de si una característica se eliminaba por completo o se cambiaba su comportamiento. Ahora se distinguen en función de su público objetivo y de la forma en que son manejados por los filtros de advertencia predeterminados.

29.5.2 El filtro de advertencias

El filtro de advertencias controla si las advertencias se ignoran, se muestran o se convierten en errores (planteando una excepción).

Conceptualmente, el filtro de advertencias mantiene una lista ordenada de especificaciones de filtros; cualquier advertencia específica se compara con cada especificación de filtro de la lista por turno hasta que se encuentra una coincidencia; el filtro determina la disposición de la coincidencia. Cada entrada es una tupla de la forma (*action*, *message*, *category*, *module*, *lineno*), donde:

- action* es una de las siguientes cadenas:

Valor	Disposición
"default"	imprime la primera ocurrencia de advertencias coincidentes para cada lugar (módulo + número de línea) donde se emite la advertencia
"error"	convertir las advertencias de coincidencia en excepciones
"ignore"	nunca imprime advertencias que coincidan
"always"	Siempre imprime advertencias que coincidan
"module"	imprime la primera ocurrencia de advertencias coincidentes para cada módulo en el que se emite la advertencia (independientemente del número de línea)
"once"	imprime sólo la primera aparición de advertencias coincidentes, independientemente de la ubicación

- message* es una cadena que contiene una expresión regular que el inicio del mensaje de advertencia debe coincidir. La expresión está compilada para que siempre sea insensible a las mayúsculas y minúsculas.
- category* es una clase (una subclase de *Warning*) de la cual la categoría de advertencia debe ser una subclase para poder coincidir.

- *module* es una cadena que contiene una expresión regular que el nombre del módulo debe coincidir. La expresión se compila para que distinga entre mayúsculas y minúsculas.
- *lineno* es un número entero que el número de línea donde ocurrió la advertencia debe coincidir, o 0 para coincidir con todos los números de línea.

Como la clase `Warning` se deriva de la clase `Excepción` incorporada, para convertir una advertencia en un error simplemente elevamos la `category` (`message`).

Si se informa de una advertencia y no coincide con ningún filtro registrado, se aplica la acción «por defecto» (de ahí su nombre).

Descripción de los filtros de advertencia

El filtro de advertencias se inicializa con `-W options` pasado a la línea de comandos del intérprete de Python y la variable de entorno `PYTHONWARNINGS`. El intérprete guarda los argumentos de todas las entradas suministradas sin interpretación en `sys.warnoptions`; el módulo `warnings` los analiza cuando se importa por primera vez (las opciones no válidas se ignoran, después de imprimir un mensaje a `sys.stderr`).

Los filtros de advertencia individuales se especifican como una secuencia de campos separados por dos puntos:

```
action:message:category:module:line
```

El significado de cada uno de estos campos es como se describe en *El filtro de advertencias*. Cuando se enumeran varios filtros en una sola línea (como en `PYTHONWARNINGS`), los filtros individuales se separan por comas y los filtros que se enumeran más tarde tienen prioridad sobre los que se enumeran antes de ellos (ya que se aplican de izquierda a derecha, y los filtros aplicados más recientemente tienen prioridad sobre los anteriores).

Los filtros de advertencia comúnmente utilizados se aplican a todas las advertencias, a las advertencias de una categoría particular o a las advertencias planteadas por módulos o paquetes particulares. Algunos ejemplos:

```
default                # Show all warnings (even those ignored by default)
ignore                 # Ignore all warnings
error                  # Convert all warnings to errors
error::ResourceWarning # Treat ResourceWarning messages as errors
default::DeprecationWarning # Show DeprecationWarning messages
ignore,default::mymodule # Only report warnings triggered by "mymodule"
error::mymodule[.*]     # Convert warnings to errors in "mymodule"
                        # and any subpackages of "mymodule"
```

Filtro de advertencia predeterminado

Por defecto, Python instala varios filtros de advertencia, que pueden ser anulados por la opción de línea de comandos `-W`, la variable de entorno `PYTHONWARNINGS` y llamadas a `filterwarnings()`.

En versiones regulares, el filtro de advertencia por defecto tiene las siguientes entradas (en orden de precedencia):

```
default::DeprecationWarning:__main__
ignore::DeprecationWarning
ignore::PendingDeprecationWarning
ignore::ImportWarning
ignore::ResourceWarning
```

En versiones de depuración (*debug builds*), la lista de filtros de advertencia por defecto está vacía.

Distinto en la versión 3.2: `DeprecationWarning` es ahora ignorado por defecto además de `PendingDeprecationWarning`.

Distinto en la versión 3.7: `DeprecationWarning` se muestra de nuevo por defecto cuando se activa directamente por código en `__main__`.

Distinto en la versión 3.7: `BytesWarning` ya no aparece en la lista de filtros por defecto y en su lugar se configura a través de `sys.warnoptions` cuando `-b` se especifica dos veces.

Anulando el filtro por defecto

Los desarrolladores de aplicaciones escritas en Python pueden desear ocultar *todas* las advertencias de nivel de Python a sus usuarios por defecto, y sólo mostrarlas cuando se realizan pruebas o se trabaja de otra manera en la aplicación. El atributo `sys.warnoptions` utilizado para pasar las configuraciones de los filtros al intérprete puede utilizarse como marcador para indicar si las advertencias deben ser desactivadas o no:

```
import sys

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

Se aconseja a los desarrolladores de pruebas de código Python que se aseguren de que *todas* las advertencias se muestren por defecto para el código que se está probando, usando un código como:

```
import sys

if not sys.warnoptions:
    import os, warnings
    warnings.simplefilter("default") # Change the filter in this process
    os.environ["PYTHONWARNINGS"] = "default" # Also affect subprocesses
```

Por último, se aconseja a los desarrolladores de shells interactivos que ejecuten el código de usuario en un espacio de nombres distinto de `__main__` que se aseguren de que los mensajes `DeprecationWarning` se hagan visibles por defecto, utilizando un código como el siguiente (donde `user_ns` es el módulo utilizado para ejecutar el código introducido interactivamente):

```
import warnings
warnings.filterwarnings("default", category=DeprecationWarning,
                        module=user_ns.get("__name__"))
```

29.5.3 Eliminación temporal de las advertencias

Si está utilizando un código que sabe que va a provocar una advertencia, como una función obsoleta, pero no quiere ver la advertencia (incluso cuando las advertencias se han configurado explícitamente a través de la línea de comandos), entonces es posible suprimir la advertencia utilizando el gestor de contexto `catch_warnings`:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

Mientras que dentro del gestor de contexto todas las advertencias serán simplemente ignoradas. Esto le permite utilizar el código conocido como obsoleto sin tener que ver la advertencia, mientras que no suprime la advertencia para otro código que podría no ser consciente de su uso de código obsoleto. Nota: esto sólo puede garantizarse en una aplicación de un solo hilo. Si dos o más hilos utilizan el gestor de contexto `catch_warnings` al mismo tiempo, el comportamiento es indefinido.

29.5.4 Advertencias de prueba

Para probar las advertencias planteadas por el código, use el administrador de contexto `catch_warnings`. Con él puedes mutar temporalmente el filtro de advertencias para facilitar tus pruebas. Por ejemplo, haz lo siguiente para capturar todas las advertencias levantadas para comprobar:

```
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 isinstance(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

También se puede hacer que todas las advertencias sean excepciones usando `error` en lugar de `always`. Una cosa que hay que tener en cuenta es que si una advertencia ya se ha planteado debido a una regla de once o `default`, entonces no importa qué filtros estén establecidos, la advertencia no se verá de nuevo a menos que el registro de advertencias relacionadas con la advertencia se haya borrado.

Una vez que se cierra el gestor de contexto, el filtro de advertencias se restablece al estado en que se encontraba al entrar en el contexto. Esto evita que las pruebas cambien el filtro de advertencias de forma inesperada entre una prueba y otra, y que se produzcan resultados indeterminados en las pruebas. La función `showwarning()` del módulo también se restaura a su valor original. Nota: esto sólo puede garantizarse en una aplicación de un solo hilo. Si dos o más hilos utilizan el gestor de contexto `catch_warnings` al mismo tiempo, el comportamiento es indefinido.

Cuando se prueban múltiples operaciones que plantean el mismo tipo de advertencia, es importante probarlas de manera que se confirme que cada operación plantea una nueva advertencia (por ejemplo, establecer advertencias que se planteen como excepciones y comprobar que las operaciones plantean excepciones, comprobar que la longitud de la lista de advertencias siga aumentando después de cada operación, o bien suprimir las entradas anteriores de la lista de advertencias antes de cada nueva operación).

29.5.5 Actualización del código para las nuevas versiones de las dependencias

Las categorías de advertencia que interesan principalmente a los desarrolladores de Python (más que a los usuarios finales de aplicaciones escritas en Python) se ignoran por defecto.

En particular, esta lista de «ignorados por defecto» incluye `DeprecationWarning` (para cada módulo excepto `__main__`), lo que significa que los desarrolladores deben asegurarse de probar su código con advertencias típicamente ignoradas hechas visibles para recibir notificaciones oportunas de futuros cambios del API a última hora (ya sea en la biblioteca estándar o en paquetes de terceros).

En el caso ideal, el código tendrá un conjunto de pruebas adecuado, y el corredor de pruebas se encargará de habilitar implícitamente todas las advertencias cuando se ejecuten las pruebas (el corredor de pruebas proporcionado por el módulo `unittest` hace esto).

En casos menos ideales, las aplicaciones pueden ser revisadas por el uso de interfaces desaprobadas pasando `-Wd` al intérprete de Python (esto es la abreviatura de `-W default`) o ajustando `PYTHONWARNINGS=default` en el entorno. Esto permite el manejo por defecto de todas las advertencias, incluyendo aquellas que son ignoradas por defecto. Para cambiar la acción que se lleva a cabo para las advertencias encontradas puede cambiar el argumento que se pasa a `-W` (por ejemplo `-W error`). Consulte el indicador `-W` para obtener más detalles sobre lo que es posible.

29.5.6 Funciones Disponibles

`warnings.warn` (*message*, *category=None*, *stacklevel=1*, *source=None*)

Emite una advertencia, o tal vez la ignora o lanza una excepción. El argumento *category*, si se da, debe ser un *warning category class*; por defecto es *UserWarning*. Alternativamente, *message* puede ser una instancia *Warning*, en cuyo caso *category* será ignorada y se usará *message.__class__*. En este caso, el texto del mensaje será `str(message)`. Esta función hace una excepción si la advertencia particular emitida es convertida en un error por el *warnings filter*. El argumento *stacklevel* puede ser usado por funciones de envoltura escritas en Python, como esta:

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

Esto hace que la advertencia se refiera al invocador de `deprecation()`, en lugar de a la fuente de `deprecation()` en sí (ya que esta última perdería el propósito del mensaje de advertencia).

source, si se suministra, es el objeto destruido que emitió un *ResourceWarning*.

Distinto en la versión 3.6: Añadido parámetro *source*.

`warnings.warn_explicit` (*message*, *category*, *filename*, *lineno*, *module=None*, *registry=None*, *module_globals=None*, *source=None*)

Se trata de una interfaz de bajo nivel para la funcionalidad de `warn()`, pasando explícitamente el mensaje, la categoría, el nombre de archivo y el número de línea, y opcionalmente el nombre del módulo y el registro (que debería ser el diccionario `__warningregistry__` del módulo). El nombre del módulo por defecto es el nombre de archivo con `.py` desmembrado; si no se pasa el registro, la advertencia nunca se suprime. *message* debe ser una cadena y *category* una subclase de *Warning* o *message* puede ser una instancia *Warning*, en cuyo caso *category* será ignorada.

module_globals, si se suministra, debe ser el espacio de nombres global en uso por el código para el que se emite la advertencia. (Este argumento se utiliza para apoyar la visualización de la fuente de los módulos que se encuentran en los archivos zip o en otras fuentes de importación que no son del sistema de archivos).

source, si se suministra, es el objeto destruido que emitió un *ResourceWarning*.

Distinto en la versión 3.6: Añade el parámetro *source*.

`warnings.showwarning` (*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)

Escriba una advertencia en un archivo. La implementación por defecto llama `formatwarning(message, category, filename, lineno, line)` y escribe la cadena resultante a *file*, que por defecto es `sys.stderr`. Puede reemplazar esta función con cualquier interlocutor asignando a `warnings.showwarning`. *line* es una línea de código fuente que se incluirá en el mensaje de advertencia; si no se proporciona *line*, `showwarning()` intentará leer la línea especificada por *nombre de archivo* y *lineno*.

`warnings.formatwarning` (*message*, *category*, *filename*, *lineno*, *line=None*)

Formatea una advertencia de la manera estándar. Esto retorna una cadena que puede contener nuevas líneas incrustadas y termina en una nueva línea. *line* es una línea de código fuente a incluir en el mensaje de advertencia; si *line* no se suministra, `formatwarning()` intentará leer la línea especificada por el nombre de fichero *filename* y *lineno*.

`warnings.filterwarnings` (*action*, *message=""*, *category=Warning*, *module=""*, *lineno=0*, *append=False*)

Inserta una entrada en la lista de *warnings filter specifications*. La entrada se inserta por defecto en el frente; si *append* es verdadero, se inserta al final. Esto comprueba los tipos de los argumentos, compila las expresiones regulares *message* y *module*, y las inserta como una tupla en la lista de filtros de aviso. Las entradas más cercanas al principio de la lista anulan las entradas posteriores de la lista, si ambas coinciden con una advertencia en particular. Los argumentos omitidos predeterminan un valor que coincide con todo.

`warnings.simplefilter` (*action*, *category=Warning*, *lineno=0*, *append=False*)

Inserte una simple entrada en la lista de *warnings filter specifications*. El significado de los parámetros de la función es el mismo que el de `filterwarnings()`, pero no se necesitan expresiones regulares ya que el filtro insertado siempre coincide con cualquier mensaje de cualquier módulo siempre que la categoría y el número de línea coincidan.

`warnings.resetwarnings()`

Reajusta el filtro de advertencias. Esto descarta el efecto de todas las llamadas previas a `filterwarnings()`, incluyendo la de las opciones de línea de comandos de `-W` y las llamadas a `simplefilter()`.

29.5.7 Gestores de Contexto disponibles

class `warnings.catch_warnings` (*, `record=False`, `module=None`)

Un gestor de contexto que copia y, al salir, restaura el filtro de advertencias y la función `showwarning()`. Si el argumento `record` es `False` (por defecto) el gestor de contextos retorna `None` al entrar. Si `record` es `True`, se retorna una lista que se va poblando progresivamente de objetos como se ve por una función personalizada de `showwarning()` (que también suprime la salida a `sys.stdout`). Cada objeto de la lista tiene atributos con los mismos nombres que los argumentos de `showwarning()`.

El argumento `module` toma un módulo que será usado en lugar del módulo retornado cuando se importa `warnings` cuyo filtro será protegido. Este argumento existe principalmente para probar el propio módulo `warnings`.

Nota: El gestor `catch_warnings` funciona reemplazando y luego restaurando la función `showwarning()` del módulo y la lista interna de especificaciones del filtro. Esto significa que el gestor de contexto está modificando el estado global y por lo tanto no es seguro para los hilos.

29.6 dataclasses — Clases de datos

Código fuente: `Lib/dataclasses.py`

Este módulo provee un decorador y funciones para añadir *métodos especiales* automáticamente, como `__init__()` y `__repr__()` por ejemplo, a clases definidas por el usuario. Fue originalmente descrito en [PEP 557](#).

Las variables miembro a utilizar en estos métodos generados son definidas teniendo en cuenta anotaciones de tipo [PEP 526](#). Por ejemplo, en este código:

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

Añadirá, además de otros métodos, un método `__init__()` con la siguiente estructura:

```
def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0):
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

Es importante observar que este método es añadido a la clase automáticamente; está implícito en la definición de `InventoryItem` implementada arriba.

Nuevo en la versión 3.7.

29.6.1 Decoradores, clases y funciones del módulo

`@dataclasses.dataclass` (*, `init=True`, `repr=True`, `eq=True`, `order=False`, `unsafe_hash=False`, `frozen=False`)

Esta función es un *decorator* utilizado para añadir a las clases *los métodos especiales* generados, como se describe a continuación.

El decorador `dataclass()` examina la clase para encontrar `fields`. Un `field` (“campo”) se define como una variable de clase que tiene una *anotación de variable*. A excepción de los dos casos descritos debajo, nada en `dataclass()` examina el tipo especificado en la anotación de variable.

El orden de los campos en los métodos generados es el mismo en el que se encuentran en la definición de la clase.

El decorador `dataclass()` añade varios métodos «dunder» (abreviación de “double underline”) a la clase, descritos a continuación. Si alguno de los métodos añadidos ya existe en la definición de la clase, el comportamiento dependerá del parámetro, como se documenta abajo. El decorador retorna la misma clase con la que es llamado, no crea una nueva.

Si `dataclass()` es llamado como un simple decorador sin parámetros, actúa con los valores por defecto documentados aquí. Específicamente, los siguientes tres usos de `dataclass()` son equivalentes:

```
@dataclass
class C:
    ...

@dataclass()
class C:
    ...

@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False,
↪frozen=False)
class C:
    ...
```

Los parámetros de `dataclass()` son:

- `init`: Si es verdadero (valor por defecto), el método `__init__()` será generado.
Si la clase ya define `__init__()`, este parámetro es ignorado.
- `repr`: Si es verdadero (valor por defecto), el método `__repr__()` es generado. La cadena de representación generada tendrá el nombre de la clase junto al nombre y la representación de cada uno de sus campos, en el mismo orden en el que están definidos en la clase. Es posible indicar que ciertos campos no sean incluidos en la representación. Por ejemplo: `InventoryItem(name='widget', unit_price=3.0, quantity_on_hand=10)`.
Si la clase ya define `__repr__()`, este parámetro es ignorado.
- `eq`: Si es verdadero (por defecto), el método `__eq__()` es generado. Este método compara entre instancias de la clase representando cada una de ellas mediante una tupla, siendo los elementos de la misma los campos de la clase ubicados en el mismo orden en el que fueron definidos (dos tuplas son iguales si, y sólo si, sus campos son iguales).
Si la clase ya define `__eq__()`, este parámetro es ignorado.
- `order`: Si es verdadero (`False` es el valor por defecto), los métodos `__lt__()`, `__le__()`, `__gt__()` y `__ge__()` serán generados. Estos métodos comparan la clase como si fuera una tupla con sus campos, en orden. Ambas instancias en la comparación deben ser del mismo tipo. Si `order` es verdadero y `eq` falso, se lanza una excepción `ValueError`.
Si la clase ya define `__lt__()`, `__le__()`, `__gt__()` o `__ge__()`, se lanza una excepción `TypeError`.
- `unsafe_hash`: Si es `False` (por defecto), se genera el método `__hash__()` de acuerdo a los valores de `eq` y `frozen` definidos.

`__hash__()` es utilizado por la función incorporada `hash()` y cuando los objetos definidos por la clase son añadidos a colecciones hash, como por ejemplo diccionarios y conjuntos. Definir el método `__hash__()` en una clase implica que sus instancias son inmutables. La mutabilidad es una propiedad compleja, ya que depende de cómo el programador utilice el objeto, la existencia y comportamiento de `__eq__()` y del valor asignado a las flags `eq` y `frozen` en el decorador `dataclass()`.

Por defecto, `dataclass()` no añade de forma implícita el método `__hash__()` a menos que sea seguro hacerlo. Tampoco añade o cambia un método `__hash__()` previamente definido de forma explícita. Definir el atributo de clase `__hash__ = None` tiene un significado específico en Python, descrito en la documentación dedicada a `__hash__()`.

If `__hash__()` is not explicitly defined, or if it is set to `None`, then `dataclass()` may add an implicit `__hash__()` method. Although not recommended, you can force `dataclass()` to create a `__hash__()` method with `unsafe_hash=True`. This might be the case if your class is logically immutable but can nonetheless be mutated. This is a specialized use case and should be considered carefully.

A continuación se explican las reglas que se aplican en la creación implícita del método `__hash__()`. Observar que no es compatible definir explícitamente un método `__hash__()` en su clase de datos y al mismo tiempo asignar `unsafe_hash=True`; esto lanza una excepción `TypeError`.

Si `eq` y `frozen` son ambos verdaderos, `dataclass()` genera por defecto un método `hash()` por ti. En el caso que `eq` sea verdadero y `frozen` falso, a `__hash__()` se le asigna `None`, en consecuencia será *unhashable* (lo cual es deseable, ya que es mutable). Si `eq` es falso, `__hash__()` permanece sin cambios, por lo que en este caso se hará uso del método `hash()` heredado de la superclase (lo que implica que si la superclase es *object*, se aplicará el *hashing* basado en el id de los objetos).

- `frozen`: Si es verdadero (el valor por defecto es `False`), cualquier intento de asignación a un campo de la clase lanza una excepción. Esto emula el comportamiento de las instancias congeladas (*frozen*) de sólo lectura. Si `__setattr__()` o `__delattr__()` son definidos en la clase, se lanzará una excepción `TypeError`. Esto es ampliado más abajo.

Los `fields` pueden especificar un valor por defecto opcionalmente, simplemente usando la sintaxis normal de Python:

```
@dataclass
class C:
    a: int          # 'a' has no default value
    b: int = 0      # assign a default value for 'b'
```

En este ejemplo, tanto `a` como `b` serán incluidos en el método `__init__()` agregado, el cual será definido como sigue:

```
def __init__(self, a: int, b: int = 0):
```

Si, en la definición de una clase, a un campo con valor por defecto le sigue un campo sin valor por defecto será lanzada una excepción `TypeError`. Esto se aplica también a la implementación de una clase única o como resultado de herencia de clases.

`dataclasses.field(*, default=MISSING, default_factory=MISSING, repr=True, hash=None, init=True, compare=True, metadata=None)`

Para casos de uso común, estas funcionalidades son suficientes. Sin embargo, existen otras características de las clases de datos que requieren información adicional en ciertos campos. Para satisfacer esta necesidad, es posible reemplazar cualquier valor por defecto de un campo mediante una llamada a la función `field()`. Por ejemplo:

```
@dataclass
class C:
    mylist: List[int] = field(default_factory=list)

c = C()
c.mylist += [1, 2, 3]
```

Como se muestra arriba, el valor `MISSING` es un objeto centinela utilizado para detectar si los parámetros `default` y `default_factory` son provistos. Este objeto centinela es utilizado debido a que `None` es un valor válido para `default`. Ningún procedimiento debe utilizar directamente el valor `MISSING`.

Los parámetros de `field()` son:

- `default`: Si es provisto, este será el valor por defecto para este campo. Es necesario que sea definido ya que la propia llamada a `field()` reemplaza la posición normal del valor por defecto.
- `default_factory`: Si es provisto, debe ser un objeto invocable sin argumentos, el cual será llamado cuando el valor por defecto de este campo sea necesario. Además de otros propósitos, puede ser utilizado para especificar campos con valores por defecto mutables, como se explica a continuación. Especificar tanto `default` como `default_factory` resulta en un error.
- `init`: Si es verdadero (por defecto), este campo es incluido como parámetro del método `__init__()` generado.
- `repr`: Si es verdadero (por defecto), este campo es incluido en la cadena de caracteres que retorna el método `__repr__()` generado.
- `compare`: Si es verdadero (por defecto), este campo es incluido en los métodos de comparación generados (`__eq__()`, `__gt__()` y otros).
- `hash`: Su valor puede ser de tipo booleano o `None`. Si es verdadero, este campo es incluido en el método `__hash__()` generado. Si es `None` (por defecto), utiliza el valor de `compare`: normalmente éste es el comportamiento esperado. Un campo debería ser considerado para el `hash` si es compatible con operaciones de comparación. Está desaconsejado establecer este valor en algo que no sea `None`.

Una posible razón para definir `hash=False` y `compare=True` podría ser el caso en el que computar el valor `hash` para dicho campo es costoso pero el campo es necesario para los métodos de comparación, siempre que existan otros campos que contribuyen al valor `hash` del tipo. Incluso si un campo se excluye del `hash`, se seguirá utilizando a la hora de comparar.

- `metadata`: Puede ser un mapeo o `None`. `None` es tratado como un diccionario vacío. Este valor es envuelto en `MappingProxyType()` para que sea de sólo lectura y visible en el objeto `Field`. No es utilizado por las clases de datos, mas bien es provisto como un mecanismo de extensión de terceros. Varios terceros pueden tener su propia clave para utilizar como espacio de nombres en `metadata`.

Si el valor por defecto de un campo es especificado por una llamada a `field()`, los atributos de clase para este campo serán reemplazados por los especificados en el valor `default`. Si el valor de `default` no es provisto, el atributo de clase será eliminado. La idea es que, después que la ejecución del decorador `dataclass()`, todos los atributos de la clase contengan los valores por defecto de cada campo, como si fueran definidos uno por uno. Por ejemplo, luego de:

```
@dataclass
class C:
    x: int
    y: int = field(repr=False)
    z: int = field(repr=False, default=10)
    t: int = 20
```

El atributo de clase `C.z` será 10, el atributo de clase `C.t` será 20 y los atributos de clase `C.x` y `C.y` no serán definidos.

class `dataclasses.Field`

Los objetos `Field` describen cada campo definido. Estos objetos son creados internamente y son retornados por el método `fields()` definido en este módulo (explicado más abajo). Los usuarios no deben instanciar un objeto `Field` directamente. Sus atributos documentados son:

- `name`: El nombre del campo.
- `type`: El tipo del campo.
- `default`, `default_factory`, `init`, `repr`, `hash`, `compare` y `metadata` tienen los mismos valores y significados respecto a la declaración de `field()` (ver arriba).

Pueden existir otros atributos, pero son privados y no deberían ser considerados ni depender de ellos.

`dataclasses.fields` (*class_or_instance*)

Retorna una tupla de objetos *Field* que definen los campos para esta clase de datos. Acepta tanto una clase de datos como una instancia de esta. Lanza una excepción *TypeError* si se le pasa cualquier otro objeto. No retorna pseudocampos, que son *ClassVar* o *InitVar*.

`dataclasses.asdict` (*instance*, *, *dict_factory=dict*)

Convierte la clase de datos *instance* en un diccionario (usando la función fábrica *dict_factory*). Cada clase de datos es convertida a un diccionario con sus campos como parejas *name: value*. Las clases de datos, diccionarios, listas y tuplas son convertidas recursivamente. Por ejemplo:

```
@dataclass
class Point:
    x: int
    y: int

@dataclass
class C:
    mylist: List[Point]

p = Point(10, 20)
assert asdict(p) == {'x': 10, 'y': 20}

c = C([Point(0, 0), Point(10, 4)])
assert asdict(c) == {'mylist': [{'x': 0, 'y': 0}, {'x': 10, 'y': 4}]}
```

Lanza una excepción *TypeError* si *instance* no es una instancia de una clase de datos.

`dataclasses.astuple` (*instance*, *, *tuple_factory=tuple*)

Convierte la clase de datos *instance* a una tupla (usando la función fábrica *tuple_factory*). Cada clase de datos es convertida a una tupla con los valores de sus campos. Las clases de datos, diccionarios, listas y tuplas son convertidas recursivamente.

Continuando con el ejemplo anterior:

```
assert astuple(p) == (10, 20)
assert astuple(c) == ((0, 0), (10, 4),)
```

Lanza una excepción *TypeError* si *instance* no es una instancia de una clase de datos.

`dataclasses.make_dataclass` (*cls_name*, *fields*, *, *bases=()*, *namespace=None*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*)

Crea una nueva clase de datos con el nombre *cls_name*, con los campos definidos en *fields*, con las clases base dadas en *bases* e inicializada con el espacio de nombres dado en *namespace*. *fields* es un iterable que cumple con una de estas formas: *name*, (*name*, *type*) o (*name*, *type*, *Field*). Si solo *name* es proporcionado, *typing.Any* es usado para *type*. Los valores *init*, *repr*, *eq*, *order*, *unsafe_hash* y *frozen* tienen el mismo significado que en la función *dataclass()*.

Esta función no es estrictamente necesaria debido a que cualquier mecanismo de Python para crear una nueva clase con `__annotations__` puede usar la función *dataclass()* para convertir esa clase en una clase de datos. Esta función se proporciona simplemente por comodidad. Por ejemplo:

```
C = make_dataclass('C',
    [('x', int),
     'y',
     ('z', int, field(default=5))],
    namespace={'add_one': lambda self: self.x + 1})
```

Es equivalente a:

```
@dataclass
class C:
    x: int
    y: 'typing.Any'
    z: int = 5

    def add_one(self):
        return self.x + 1
```

`dataclasses.replace(instance, **changes)`

Crea un nuevo objeto del mismo tipo que `instance`, reemplazando los campos correspondientes con los valores de `changes`. Si `instance` no es una clase de datos se lanza una excepción `TypeError`. Si los valores en `changes` no especifican campos, también se lanza una excepción `TypeError`.

El objeto recién retornado es creado llamando al método `__init__()` de la clase de datos. Esto asegura que `__post_init__()`, si existe, también será llamado.

Las variables de solo inicialización sin valores predeterminados, si existen, deben especificarse en la llamada a `replace()` para que puedan pasarse a `__init__()` y `__post_init__()`.

Es un error que `changes` contenga cualquier campo que esté definido como `init=False`. Una excepción `ValueError` se lanzará en este caso.

Tenga en cuenta cómo funcionan los campos `init=False` durante una llamada a `replace()`. No se copian del objeto de origen, sino que, de inicializarse, lo hacen en `__post_init__()`. Se espera que los campos `init=False` se utilicen en contadas ocasiones y con prudencia. Si se utilizan, podría ser conveniente tener constructores de clase alternativos, o quizás un método personalizado `replace()` (o con un nombre similar) que maneje la copia de instancias.

`dataclasses.is_dataclass(class_or_instance)`

Retorna `True` si su parámetro es una clase de datos o una instancia de una, en caso contrario retorna `False`.

Si se necesita conocer si una clase es una instancia de `dataclass` (y no una clase de datos en si misma), se debe agregar una verificación adicional para `not isinstance(obj, type)`:

```
def is_dataclass_instance(obj):
    return is_dataclass(obj) and not isinstance(obj, type)
```

29.6.2 Procesamiento posterior a la inicialización

El código del método generado `__init__()` llamará a un método llamado `__post_init__()`, si `__post_init__()` está definido en la clase. Normalmente se llamará como `self.__post_init__()`. Sin embargo, si se define algún campo `InitVar`, también se pasarán a `__post_init__()` en el orden en que se definieron en la clase. Si no se genera el método `__init__()`, entonces `__post_init__()` no se llamará automáticamente.

Entre otros usos, esto permite inicializar valores de campo que dependen de uno o más campos. Por ejemplo:

```
@dataclass
class C:
    a: float
    b: float
    c: float = field(init=False)

    def __post_init__(self):
        self.c = self.a + self.b
```

Consulta la sección sobre variables de solo inicialización que hay a continuación para conocer las posibles formas de pasar parámetros a `__post_init__()`. También vea la advertencia sobre cómo `replace()` maneja los campos `init = False`.

29.6.3 Variables de clase

Uno de los dos casos donde `dataclass()` realmente inspecciona el tipo de un campo, es para determinar si dicho campo es una variable de clase como se define en [PEP 526](#). Lo hace comprobando si el tipo del campo es `typing.ClassVar`. Si un campo es una `ClassVar`, se deja de considerar como campo y los mecanismos de las clases de datos lo ignoran. Tales pseudocampos `ClassVar` no son retornados por la función del módulo `fields()`.

29.6.4 Variable de solo inicialización

El otro caso donde `dataclass()` inspecciona una anotación de tipo es para determinar si un campo es una variable de solo inicialización. Lo hace comprobando si el tipo de un campo es `dataclasses.InitVar`. Si un campo es un `InitVar`, se considera un pseudocampo llamado “campo de solo inicialización”. Como no es un campo verdadero, no es retornado por la función del módulo `fields()`. Los campos de solo inicialización se agregan como parámetros al método generado `__init__()` y se pasan al método opcional `__post_init__()`. No son utilizados de otra manera por las clases de datos.

Por ejemplo, supongamos que se va a inicializar un campo desde una base de datos, de no proporcionarse un valor al crear la clase:

```
@dataclass
class C:
    i: int
    j: int = None
    database: InitVar[DatabaseType] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

En este caso, `fields()` retornará objetos `Field` para `i` y `j`, pero no para `database`.

29.6.5 Instancias congeladas

No es posible crear objetos verdaderamente inmutables en Python. Sin embargo, se puede emular la inmutabilidad pasando `frozen=True` al decorador `dataclass()`. En este caso, las clases de datos añadirán los métodos `__setattr__()` y `__delattr__()` a la clase. Estos métodos lanzarán una excepción `FrozenInstanceError` cuando sean llamados.

Hay una pequeña penalización de rendimiento cuando se usa `frozen=True`, esto se debe a que `__init__()` no puede usar una asignación simple para inicializar campos, viéndose obligado a usar `object.__setattr__()`.

29.6.6 Herencia

Cuando la clase de datos está siendo creada por el decorador `dataclass()`, revisa todas las clases base de la clase en el MRO invertido (es decir, comenzando en `object`) y, para cada clase de datos que encuentra, agrega los campos de esa clase base a un mapeo ordenado. Después de agregar todos los campos de la clase base, agrega sus propios campos al mapeo. Todos los métodos generados utilizarán este mapeo ordenado calculado combinando los campos. Como los campos están en orden de inserción, las clases derivadas anulan las clases base. Un ejemplo:

```
@dataclass
class Base:
    x: Any = 15.0
    y: int = 0

@dataclass
```

(continué en la próxima página)

(proviene de la página anterior)

```
class C(Base):
    z: int = 10
    x: int = 15
```

La lista final de campos es, en orden, x, y, z. El tipo final de x es `int`, como se especifica en la clase C.

El método `__init__()` generado para C se verá como:

```
def __init__(self, x: int = 15, y: int = 0, z: int = 10):
```

29.6.7 Funciones fábrica por defecto

Si un `field()` especifica una `default_factory`, se llama sin argumentos cuando se necesita un valor predeterminado para el campo. Por ejemplo, para crear una nueva instancia de una lista, debe usarse:

```
mylist: list = field(default_factory=list)
```

Si un campo está excluido de `__init__()` (usando `init = False`) y el campo también especifica `default_factory`, entonces la función de fábrica predeterminada siempre se llamará desde la función generada `__init__()`. Esto sucede porque no existe otra forma de darle al campo un valor inicial.

29.6.8 Valores por defecto mutables

Python almacena los valores miembros por defecto en atributos de clase. Considera este ejemplo, sin usar clases de datos:

```
class C:
    x = []
    def add(self, element):
        self.x.append(element)

o1 = C()
o2 = C()
o1.add(1)
o2.add(2)
assert o1.x == [1, 2]
assert o1.x is o2.x
```

Tenga en cuenta que, tal como cabe esperar, las dos instancias de la clase C comparten la misma variable de clase x.

Usando clases de datos, *si* este código fuera válido:

```
@dataclass
class D:
    x: List = []
    def add(self, element):
        self.x += element
```

generaría un código similar a:

```
class D:
    x = []
    def __init__(self, x=x):
        self.x = x
    def add(self, element):
```

(continué en la próxima página)

(proviene de la página anterior)

```

        self.x += element

assert D().x is D().x

```

Este tiene el mismo problema que el ejemplo original usando la clase `C`. Es decir, dos instancias de la clase `D` que no especifican un valor para `x`, al crear una instancia de la clase, compartirán la misma copia de `x`. Debido a que las clases de datos usan simplemente el mecanismo normal de creación de clases de Python, también comparten este comportamiento. No existe una forma genérica de que las clases de datos detecten esta condición. En su lugar, las clases de datos generarán una excepción `TypeError` si detectan un parámetro predeterminado de tipo `list`, `dict` o `set` (contenedores incorporados mutables). Esta es una solución parcial, pero protege contra muchos de los errores más comunes.

Usar las funciones fábrica por defecto es una forma de crear nuevas instancias de tipos mutables como valores por defecto para campos:

```

@dataclass
class D:
    x: list = field(default_factory=list)

assert D().x is not D().x

```

29.6.9 Excepciones

exception `dataclasses.FrozenInstanceError`

Raised when an implicitly defined `__setattr__()` or `__delattr__()` is called on a dataclass which was defined with `frozen=True`. It is a subclass of `AttributeError`.

29.7 contextlib — Utilidades para declaraciones de contexto with

Código fuente: [Lib/contextlib.py](#)

Este módulo proporciona utilidades para tareas comunes que involucran la declaración `with`. Para obtener más información, consulte también *Tipos Gestores de Contexto* y `context-managers`.

29.7.1 Utilidades

Funciones y clases proporcionadas:

class `contextlib.AbstractContextManager`

Una *clase base abstracta* para clases que implementan `object.__aenter__()` y `object.__aexit__()`. Se proporciona una implementación predeterminada para `object.__aenter__()` que retorna `self` mientras que `object.__aexit__()` es un método abstracto que por defecto retorna `None`. Véase también la definición de *Tipos Gestores de Contexto*.

Nuevo en la versión 3.6.

class `contextlib.AbstractAsyncContextManager`

Una *clase base abstracta* para clases que implementan `object.__aenter__()` y `object.__aexit__()`. Se proporciona una implementación predeterminada para `object.__aenter__()` que retorna `self` mientras que `object.__aexit__()` es un método abstracto que por defecto retorna `None`. Véase también la definición de `async-context-managers`.

Nuevo en la versión 3.7.

@contextlib.contextmanager

Esta función es *decorador* que se puede usar para definir una función de fábrica para gestores de contexto de declaración `with`, sin necesidad de crear una clase o separar `__enter__()` y `__exit__()` métodos.

Si bien muchos objetos admiten de forma nativa el uso con declaraciones, a veces es necesario administrar un recurso que no sea un administrador de contexto por sí mismo y no implemente un método `close()` para usar con `contextlib.close`

Un ejemplo abstracto sería el siguiente para garantizar la gestión correcta de los recursos:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwargs):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwargs)
    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
```

La función que se está decorando debe retornar un iterador *generador* cuando se llama. Este iterador debe producir exactamente un valor, que estará vinculado a los objetivos en la `with` declaración de la cláusula `as`, si existe.

En el punto donde el generador cede, se ejecuta el bloque anidado en la palabra clave `with`. El generador se reanuda luego de salir del bloque. Si se produce una excepción no controlada en el bloque, se vuelve a plantear dentro del generador en el punto donde se produjo el rendimiento. Por lo tanto, puede usar una declaración `try...except...finally` para atrapar el error (si lo hay), o asegurarse de que se realice una limpieza. Si una excepción queda atrapada simplemente para registrarla o realizar alguna acción (en lugar de suprimirla por completo), el generador debe volver a generar esa excepción. De lo contrario, el administrador de contexto del generador indicará a la palabra clave `with` que se ha manejado la excepción, y la ejecución se reanudará con la declaración inmediatamente siguiente a la palabra clave `with`.

`contextmanager()` usa *ContextDecorator* para que los gestores de contexto que crea se puedan usar como decoradores, así como en declaraciones `with`. Cuando se usa como decorador, se crea implícitamente una nueva instancia de generador en cada llamada de función (esto permite que los gestores de contexto «de-un-tiro» creados por `contextmanager()` cumplan el requisito de que los gestores de contexto admitan múltiples invocaciones para ser utilizado como decoradores).

Distinto en la versión 3.2: Uso de *ContextDecorator*.

@contextlib.asynccontextmanager

Similar a `contextmanager()`, pero crea un administrador de contexto asíncrono.

Esta función es *decorador* que se puede utilizar para definir una función de fábrica para gestores de contexto asíncrono de declaración `async with`, sin necesidad de crear una clase o separar `__aenter__()` y métodos `__aexit__()`. Debe aplicarse a una función *asynchronous generator*.

Un ejemplo simple:

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def get_connection():
    conn = await acquire_db_connection()
    try:
        yield conn
    finally:
```

(continué en la próxima página)

(proviene de la página anterior)

```

    await release_db_connection(conn)

async def get_all_users():
    async with get_connection() as conn:
        return conn.query('SELECT ...')

```

Nuevo en la versión 3.7.

`contextlib.closing(thing)`

retorna un gestor de contexto que cierra *thing* al completar el bloque. Esto es básicamente equivalente a:

```

from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()

```

Y te permite escribir código como este:

```

from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('http://www.python.org')) as page:
    for line in page:
        print(line)

```

sin necesidad de cerrar explícitamente la `page`. Incluso si se produce un error, se llamará a `page.close()` cuando salga el bloque `with`.

`contextlib.nullcontext(enter_result=None)`

retorna un gestor de contexto que retorna *enter_result* de `__enter__`, pero de lo contrario no hace nada. Está destinado a ser utilizado como un sustituto para un administrador de contexto opcional, por ejemplo:

```

def myfunction(arg, ignore_exceptions=False):
    if ignore_exceptions:
        # Use suppress to ignore all exceptions.
        cm = contextlib.suppress(Exception)
    else:
        # Do not ignore any exceptions, cm has no effect.
        cm = contextlib.nullcontext()
    with cm:
        # Do something

```

Un ejemplo usando *enter_result*:

```

def process_file(file_or_path):
    if isinstance(file_or_path, str):
        # If string, open file
        cm = open(file_or_path)
    else:
        # Caller is responsible for closing file
        cm = nullcontext(file_or_path)

    with cm as file:
        # Perform processing on the file

```

Nuevo en la versión 3.7.

`contextlib.suppress(*exceptions)`

Retorna un administrador de contexto que suprima cualquiera de las excepciones especificadas si se producen

en el cuerpo de una instrucción *with* y luego reanuda la ejecución con la primera instrucción que sigue al final de la instrucción *with*.

Al igual que con cualquier otro mecanismo que suprima completamente las excepciones, este administrador de contexto debe usarse solo para cubrir errores muy específicos en los que se sabe que continuar silenciosamente con la ejecución del programa es lo correcto.

Por ejemplo:

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

Este código es equivalente a:

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

Este gestor de contexto es *reentrant*.

Nuevo en la versión 3.4.

`contextlib.redirect_stdout` (*new_target*)

Administrador de contexto para redirigir temporalmente `sys.stdout` a otro archivo u objeto similar a un archivo.

Esta herramienta agrega flexibilidad a las funciones o clases existentes cuya salida está programada para stdout.

Por ejemplo, la salida de `help()` normalmente se envía a `sys.stdout`. Puede capturar esa salida en una cadena redirigiendo la salida a un objeto `io.StringIO`:

```
f = io.StringIO()
with redirect_stdout(f):
    help(pow)
s = f.getvalue()
```

Para enviar la salida de `help()` a un archivo en el disco, redirija la salida a un archivo normal:

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

Para enviar la salida de `help()` a `sys.stderr`:

```
with redirect_stdout(sys.stderr):
    help(pow)
```

Tenga en cuenta que el efecto secundario global en `sys.stdout` significa que este administrador de contexto no es adecuado para su uso en el código de la biblioteca y en la mayoría de las aplicaciones con subprocesos. Tampoco tiene efecto en la salida de subprocesos. Sin embargo, sigue siendo un enfoque útil para muchos scripts de utilidad.

Este gestor de contexto es *reentrant*.

Nuevo en la versión 3.4.

`contextlib.redirect_stderr` (*new_target*)

Similar a `redirect_stdout()` pero redirigiendo `sys.stderr` a otro archivo u objeto similar a un archivo.

Este gestor de contexto es *reentrant*.

Nuevo en la versión 3.5.

class `contextlib.ContextDecorator`

Una clase base que permite que un administrador de contexto también se use como decorador.

Los gestores de contexto que heredan de `ContextDecorator` tienen que implementar `__enter__` y `__exit__` de manera normal. `__exit__` conserva su manejo opcional de excepciones incluso cuando se usa como decorador.

`ContextDecorator` es utilizado por `contextmanager()`, por lo que obtiene esta funcionalidad automáticamente.

Ejemplo de `ContextDecorator`:

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False

>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
...
Starting
The bit in the middle
Finishing
```

Este cambio es solo azúcar sintáctico para cualquier construcción de la siguiente forma:

```
def f():
    with cm():
        # Do stuff
```

`ContextDecorator` le permite escribir en su lugar:

```
@cm()
def f():
    # Do stuff
```

Deja en claro que el `cm` se aplica a toda la función, en lugar de solo una parte de ella (y guardar un nivel de sangría también es bueno).

Los gestores de contexto existentes que ya tienen una clase base pueden ampliarse utilizando `ContextDecorator` como una clase mezcla (*mixin*):

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

Nota: Como la función decorada debe poder llamarse varias veces, el gestor de contexto subyacente debe admitir el uso en múltiples declaraciones `with`. Si este no es el caso, se debe utilizar la construcción original con la declaración explícita `with` dentro de la función.

Nuevo en la versión 3.2.

class contextlib.ExitStack

Un gestor de contexto que está diseñado para facilitar la combinación programática de otros gestores de contexto y funciones de limpieza, especialmente aquellas que son opcionales o que de otro modo son impulsadas por los datos de entrada.

Por ejemplo, un conjunto de archivos puede manejarse fácilmente en una sola declaración de la siguiente manera:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # All opened files will automatically be closed at the end of
    # the with statement, even if attempts to open files later
    # in the list raise an exception
```

Cada instancia mantiene una pila de retrollamadas registradas que se llaman en orden inverso cuando la instancia se cierra (ya sea explícita o implícitamente al final de una instrucción `with`). Tenga en cuenta que las retrollamadas *no* se invocan implícitamente cuando la instancia de la pila de contexto se recolecta basura.

Este modelo de pila se utiliza para que los administradores de contexto que adquieren sus recursos en su método `__init__` (como los objetos de archivo) se puedan manejar correctamente.

Dado que las retrollamadas registradas se invocan en el orden inverso del registro, esto termina comportándose como si se hubieran utilizado múltiples instrucciones anidadas `with` con el conjunto registrado de retrollamadas. Esto incluso se extiende al manejo de excepciones: si una retrollamada interna suprime o reemplaza una excepción, las retrollamadas externas se pasarán argumentos basados en ese estado actualizado.

Esta es una API de nivel relativamente bajo que se ocupa de los detalles de desenrollar correctamente la pila de retrollamadas de salida. Proporciona una base adecuada para administradores de contexto de nivel superior que manipulan la pila de salida en formas específicas de la aplicación.

Nuevo en la versión 3.3.

enter_context (*cm*)

Ingresa a un nuevo administrador de contexto y agrega su método `__exit__()` a la pila de retrollamada. El valor de retorno es el resultado del método propio del administrador de contexto `__enter__()`.

Estos administradores de contexto pueden suprimir excepciones tal como lo harían normalmente si se usaran directamente como parte de una declaración `with`.

push (*exit*)

Agrega un método de gestor de contexto `__exit__()` a la pila de retrollamada.

Como `__enter__` *no* se invoca, este método se puede usar para cubrir parte de una implementación `__enter__()` con un método propio del gestor de contexto `__exit__()`.

Si se pasa un objeto que no es un administrador de contexto, este método supone que es una retrollamada con la misma firma que el método `__exit__()` de un gestor de contexto y lo agrega directamente a la pila de retrollamada.

Al retornar valores verdaderos, estas retrollamadas pueden suprimir excepciones de la misma manera que el gestor de contexto los métodos `__exit__()` pueden hacerlo.

El objeto pasado se retorna desde la función, lo que permite que este método se use como decorador de funciones.

callback (*callback*, *args, **kwargs)

Acepta una función de retrollamada arbitraria y argumentos y la agrega a la pila de retrollamada.

A diferencia de los otros métodos, las retrollamadas agregadas de esta manera no pueden suprimir excepciones (ya que nunca se pasan los detalles de excepción).

La retrollamada pasada se retorna desde la función, lo que permite que este método se use como decorador de funciones.

pop_all ()

Transfiere la pila de retrollamada a una instancia fresca `ExitStack` y la retorna. Esta operación no invoca retrollamadas; en cambio, ahora se invocarán cuando se cierre la nueva pila (ya sea explícita o implícitamente al final de una instrucción `with`).

Por ejemplo, un grupo de archivos se puede abrir como una operación de «todo o nada» de la siguiente manera:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # Hold onto the close method, but don't call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously opened files will be
    # closed automatically. If all files are opened successfully,
    # they will remain open even after the with statement ends.
    # close_files() can then be invoked explicitly to close them all.
```

close ()

Inmediatamente desenrolla la pila de retrollamada, invocando retrollamadas en el orden inverso de registro. Para los administradores de contexto y las retrollamadas de salida registradas, los argumentos pasados indicarán que no se produjo ninguna excepción.

class `contextlib.AsyncExitStack`

Un gestor de contexto asíncrono, similar a `ExitStack`, que admite la combinación de gestores de contexto síncrono y asíncrono, además de tener rutinas para la lógica de limpieza.

El método `close()` no está implementado, `aclose()` debe usarse en su lugar.

enter_async_context (cm)

Similar a `enter_context()` pero espera un administrador de contexto asíncrono.

push_async_exit (exit)

Similar a `push()` pero espera un gestor de contexto asíncrono o una función de rutina.

push_async_callback (callback, *args, **kwargs)

Similar a `callback()` pero espera una función de rutina.

aclose ()

Similar a `close()` pero maneja adecuadamente los objetos de espera.

Continuando con el ejemplo para `asynccontextmanager()`:

```
async with AsyncExitStack() as stack:
    connections = [await stack.enter_async_context(get_connection())
                    for i in range(5)]
    # All opened connections will automatically be released at the end of
    # the async with statement, even if attempts to open a connection
    # later in the list raise an exception.
```

Nuevo en la versión 3.7.

29.7.2 Ejemplos y recetas

Esta sección describe algunos ejemplos y recetas para hacer un uso efectivo de las herramientas proporcionadas por `contextlib`.

Apoyando un número variable de gestores de contexto

El caso de uso principal para `ExitStack` es el que se proporciona en la documentación de la clase: admite un número variable de gestores de contexto y otras operaciones de limpieza en una sola `with`. La variabilidad puede provenir de la cantidad de gestores de contexto que necesitan ser impulsados por la entrada del usuario (como abrir una colección de archivos especificada por el usuario), o de que algunos de los gestores de contexto sean opcionales:

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
        stack.callback(release_special_resource, special)
    # Perform operations that use the acquired resources
```

Como se muestra, `ExitStack` también hace que sea bastante fácil de usar `with` para administrar recursos arbitrarios que no admiten de forma nativa el protocolo de gestión de contexto.

Capturando excepciones de los métodos `__enter__`

Ocasionalmente es deseable capturar excepciones de una implementación del método `__enter__`, sin capturar inadvertidamente excepciones del cuerpo de la declaración `with` o el método `__exit__` del gestor de contexto. Al usar `ExitStack`, los pasos en el protocolo de gestor de contexto se pueden separar ligeramente para permitir esto:

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle __enter__ exception
else:
    with stack:
        # Handle normal case
```

Es probable que la necesidad de hacer esto indique que la API subyacente debería proporcionar una interfaz de administración de recursos directa para usar con `try/except/finally`, pero no todas las API están bien diseñados en ese sentido. Cuando un administrador de contexto es la única API de administración de recursos proporcionada, entonces `ExitStack` puede facilitar el manejo de diversas situaciones que no se pueden manejar directamente en una declaración `with`.

Limpieza en una implementación `__enter__`

Como se señala en la documentación de `ExitStack.push()`, este método puede ser útil para limpiar un recurso ya asignado si fallan los pasos posteriores en `__enter__()`.

Aquí hay un ejemplo de cómo hacer esto para un administrador de contexto que acepta funciones de adquisición y liberación de recursos, junto con una función de validación opcional, y las asigna al protocolo de administración de contexto:

```
from contextlib import contextmanager, AbstractContextManager, ExitStack

class ResourceManager(AbstractContextManager):
```

(continué en la próxima página)

(proviene de la página anterior)

```

def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
    self.acquire_resource = acquire_resource
    self.release_resource = release_resource
    if check_resource_ok is None:
        def check_resource_ok(resource):
            return True
    self.check_resource_ok = check_resource_ok

@contextmanager
def _cleanup_on_error(self):
    with ExitStack() as stack:
        stack.push(self)
        yield
        # The validation check passed and didn't raise an exception
        # Accordingly, we want to keep the resource, and pass it
        # back to our caller
        stack.pop_all()

def __enter__(self):
    resource = self.acquire_resource()
    with self._cleanup_on_error():
        if not self.check_resource_ok(resource):
            msg = "Failed validation for {!r}"
            raise RuntimeError(msg.format(resource))
    return resource

def __exit__(self, *exc_details):
    # We don't need to duplicate any of our resource release logic
    self.release_resource()

```

Reemplazar cualquier uso de try-finally y marcar variables

Un patrón que a veces verá es una declaración de try-finally con una variable de indicador para indicar si el cuerpo de la cláusula finally debe ejecutarse o no. En su forma más simple (que ya no puede manejarse simplemente usando una cláusula except en su lugar), se parece a esto:

```

cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()

```

Al igual que con cualquier código basado en la declaración try, esto puede causar problemas de desarrollo y revisión, porque el código de configuración y el código de limpieza pueden terminar separados por secciones de código arbitrariamente largas.

`ExitStack` hace posible registrar una retrollamada para su ejecución al final de una instrucción with, y luego decide omitir la ejecución de esa retrollamada:

```

from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()

```

Esto permite que el comportamiento de limpieza previsto se haga explícito por adelantado, en lugar de requerir una variable de indicador separada.

Si una aplicación particular usa mucho este patrón, puede simplificarse aún más por medio de una pequeña clase auxiliar:

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, /, *args, **kwargs):
        super().__init__()
        self.callback(callback, *args, **kwargs)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

Si la limpieza del recurso no está bien agrupada en una función independiente, entonces todavía es posible usar la forma decoradora de `ExitStack.callback()` para declarar la limpieza del recurso por adelantado:

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()
```

Debido a la forma en que funciona el protocolo decorador, una función de retrollamada declarada de esta manera no puede tomar ningún parámetro. En cambio, se debe acceder a los recursos que se liberarán como variables de cierre.

Usar un gestor de contexto como decorador de funciones

`ContextDecorator` hace posible usar un gestor de contexto tanto en una instrucción ordinaria `with` como también como decorador de funciones.

Por ejemplo, a veces es útil envolver funciones o grupos de declaraciones con un registrador que puede rastrear la hora de entrada y la hora de salida. En lugar de escribir tanto un decorador de funciones como un administrador de contexto para la tarea, heredar de `ContextDecorator` proporciona ambas capacidades en una sola definición:

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

Las instancias de esta clase se pueden usar como un gestor de contexto:


```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

Y también como decorador de funciones:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Tenga en cuenta que hay una limitación adicional cuando se usan administradores de contexto como decoradores de funciones: no hay forma de acceder al valor de retorno de `__enter__()`. Si se necesita ese valor, aún es necesario usar una declaración explícita `with`.

Ver también:

PEP 343 - La declaración «with» La especificación, antecedentes y ejemplos de la declaración de Python `with`.

29.7.3 Gestores de contexto de uso único, reutilizables y reentrantes

La mayoría de los gestores de contexto están escritos de una manera que significa que solo se pueden usar de manera efectiva en una declaración `with` una vez. Estos administradores de contexto de un solo uso deben crearse de nuevo cada vez que se usan; si intenta usarlos por segunda vez, se activará una excepción o, de lo contrario, no funcionará correctamente.

Esta limitación común significa que generalmente es aconsejable crear gestores de contexto directamente en el encabezado de la palabra clave `with` donde se usan (como se muestra en todos los ejemplos de uso anteriores).

Los archivos son un ejemplo de gestores de contexto de un solo uso, ya que la primera `with` cerrará el archivo, evitando cualquier otra operación de E/S que use ese objeto de archivo.

Los gestores de contexto creados usando `contextmanager()` también son gestores de contexto de un solo uso, y se quejarán de la falla del generador subyacente si se intenta usarlos por segunda vez:

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
...
RuntimeError: generator didn't yield
```

Gestores contextuales reentrantes

Los gestores de contexto más sofisticados pueden ser «reentrantes». Estos administradores de contexto no solo se pueden usar en múltiples declaraciones `with`, sino que también se pueden usar *inside* a `with` que ya está usando el mismo gestor de contexto.

`threading.RLock` es un ejemplo de un administrador de contexto reentrante, como son `suppress()` y `redirect_stdout()`. Aquí hay un ejemplo muy simple de uso reentrante:

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

Es más probable que los ejemplos del mundo real de reentrada impliquen múltiples funciones que se llaman entre sí y, por lo tanto, sean mucho más complicadas que este ejemplo.

Tenga en cuenta también que ser reentrante *no* es lo mismo que ser seguro para subprocessos. `redirect_stdout()`, por ejemplo, definitivamente no es seguro para subprocessos, ya que realiza una modificación global al estado del sistema al vincular `sys.stdout` a una secuencia diferente.

Gestores contextuales reutilizables

Distintos de los administradores de contexto de uso único y reentrante son los administradores de contexto «reutilizables» (o, para ser completamente explícitos, los administradores de contexto «reutilizables, pero no reentrantes», ya que los administradores de contexto reentrantes también son reutilizables). Estos administradores de contexto admiten que se usen varias veces, pero fallarán (o de lo contrario no funcionarán correctamente) si la instancia específica del administrador de contexto ya se ha utilizado en una declaración que contiene.

`threading.Lock` es un ejemplo de un gestor de contexto reutilizable, pero no reentrante (para un bloqueo reentrante, es necesario usar `threading.RLock` en su lugar).

Otro ejemplo de un administrador de contexto reutilizable, pero no reentrante es `ExitStack`, ya que invoca *all* las retrollamadas registradas actualmente al dejar cualquier con declaración, independientemente de dónde se agregaron esas retrollamadas:

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
>>> with stack:
```

(continué en la próxima página)

(proviene de la página anterior)

```

...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context

```

Como muestra el resultado del ejemplo, la reutilización de un solo objeto de pila en múltiples con declaraciones funciona correctamente, pero intentar anidarlos hará que la pila se borre al final de la declaración más interna, lo que es poco probable que sea un comportamiento deseable.

El uso de instancias separadas `ExitStack` en lugar de reutilizar una sola instancia evita ese problema:

```

>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer context")
...     with ExitStack() as inner_stack:
...         inner_stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context

```

29.8 abc — Clases de Base Abstracta

Código fuente: [Lib/abc.py](#)

Este módulo proporciona la infraestructura para definir *clases de base abstracta* (CBAs) en Python, como se describe en [PEP 3119](#); consulte en el PEP el porqué fue agregado a Python. (Véase también [PEP 3141](#) y el módulo `numbers` con respecto a una jerarquía de tipos para números basados en CBAs.)

The `collections` module has some concrete classes that derive from ABCs; these can, of course, be further derived. In addition, the `collections.abc` submodule 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.

Este módulo provee la metaclass `ABCMeta` para definir CBAs y una clase auxiliar `ABC` para definir CBAs alternativamente a través de herencia:

class `abc.ABC`

Una clase auxiliar que tiene una `ABCMeta` como su metaclass. Con esta clase, una clase de base abstracta puede ser creada simplemente derivándola desde `ABC` evitando el uso de metaclasses algunas veces confusos, por ejemplo:

```

from abc import ABC

class MyABC(ABC):
    pass

```

Note that the type of `ABC` is still `ABCMeta`, therefore inheriting from `ABC` requires the usual precautions regarding metaclass usage, as multiple inheritance may lead to metaclass conflicts. One may also define an abstract base class by passing the metaclass keyword and using `ABCMeta` directly, for example:

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

Nuevo en la versión 3.4.

class abc.ABCMeta

Metaclases para definir Clases de Base Abstracta (CBAs).

Utilice esta metaclass para crear una CBA. Una CBA puede ser heredada directamente y así, actuar como una clase mixta. También se puede registrar clases concretas no relacionadas (incluso clases integradas) y CBAs no relacionadas como «subclases virtuales» – estas y sus descendientes serán consideradas subclases del CBA registrado por la función integrada `issubclass()`, pero la CBA registrada no aparecerá en su *MRO* (Orden de Resolución de Métodos) ni las implementaciones de método definidas por la CBA registrada serán invocables (ni siquiera a través de `super()`).¹

Las clases creadas con una metaclass de `ABCMeta` tienen el siguiente método:

register(subclass)

Registre la *subclass* como una «subclase virtual» de esta CBA. Por ejemplo:

```
from abc import ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

Distinto en la versión 3.3: Retorna la subclase registrada, para permitir su uso como decorador de clase.

Distinto en la versión 3.4: Para detectar llamadas a `register()`, se puede usar la función `get_cache_token()`.

También se puede redefinir este método en una clase de base abstracta:

__subclasshook__(subclass)

(Debe ser definido como un método de clase.)

Compruebe si la *subclass* se considera una subclase de esta CBA. Esto significa que puede personalizar aún más el comportamiento de `issubclass` sin necesidad de llamar a `register()` en cada clase que desee considerar una subclase de la CBA. (Este método de clase es llamado desde el método `__subclasscheck__()` del CBA.)

Este método debe retornar `True`, `False` o `NotImplemented`. Si retorna `True`, la *subclass* se considera una subclase de esta CBA. Si retorna `False`, la *subclass* no se considera una subclase de esta CBA, incluso si normalmente fuese una. Si retorna `NotImplemented`, la comprobación de subclase se continúa con el mecanismo usual.

Para una demostración de estos conceptos, vea este ejemplo de la definición CBA:

```
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)
```

(continué en la próxima página)

¹ Los desarrolladores de C++ pueden notar que el concepto de clase base virtual de Python no es el mismo que en C++.

(proviene de la página anterior)

```

class MyIterable(ABC):

    @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)

```

La CBA `MyIterable` define el método iterable estándar, `__iter__()`, como un método abstracto. La implementación dada aquí aún se puede llamar desde subclases. El método `get_iterator()` también forma parte de la clase de base abstracta `MyIterable`, pero no tiene que ser reemplazado en clases derivadas no abstractas.

El método de la clase `__subclasshook__()` definido aquí dice que cualquier clase que tenga un método `__iter__()` en su `__dict__` (o en la de una de sus clases base, a la que se accede a través de la lista `__mro__`) también se considera un `MyIterable`.

Por último, la última línea convierte `Foo` en una subclase virtual de `MyIterable`, aunque no define un método `__iter__()` (utiliza el protocolo iterable al estilo antiguo, definido en términos de `__len__()` y `__getitem__()`). Tenga en cuenta que esto no hará que `get_iterator` esté disponible como un método de `Foo`, por lo que es proporcionado por separado.

El módulo `abc` también proporciona el siguiente decorador:

`@abc.abstractmethod`

Un decorador que indica métodos abstractos.

El uso de este decorador requiere que la metaclass de la clase sea `ABCMeta` o se derive de esta. Una clase que tiene una metaclass derivada de `ABCMeta` no puede ser instanciada, a menos que todas sus propiedades y métodos abstractos sean anulados. Los métodos abstractos se pueden invocar usando cualquiera de los mecanismos de “super” invocación normales. `abstractmethod()` se puede utilizar para declarar métodos abstractos para propiedades y descriptores.

No se admite la adición dinámica de métodos abstractos a una clase o el intento de modificar el estado de abstracción de un método o clase una vez creado. El `abstractmethod()` sólo afecta a las subclases derivadas mediante herencia regular; las «subclases virtuales» registradas con el método `register()` de CBAs no son afectadas.

Cuando `abstractmethod()` se aplica en combinación con otros descriptores de método, se debe aplicar como el decorador más interno, como se muestra en los siguientes ejemplos de uso:

```

class C(ABC):
    @abstractmethod
    def my_abstract_method(self, ...):
        ...

    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...

    @staticmethod
    @abstractmethod

```

(continué en la próxima página)

(proviene de la página anterior)

```
def my_abstract_staticmethod(...):
    ...

@property
@abstractmethod
def my_abstract_property(self):
    ...

@my_abstract_property.setter
@abstractmethod
def my_abstract_property(self, val):
    ...

@abstractmethod
def _get_x(self):
    ...

@abstractmethod
def _set_x(self, val):
    ...

x = property(_get_x, _set_x)
```

Para interoperar correctamente con la maquinaria de clase de base abstracta, el descriptor debe identificarse como abstracto utilizando `__isabstractmethod__`. En general, este atributo debe ser `True` si alguno de los métodos utilizados para componer el descriptor es abstracto. Por ejemplo, la clase de propiedad integrada de Python `property` hace el equivalente de:

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for
                    f in (self._fget, self._fset, self._fdel))
```

Nota: A diferencia de los métodos abstractos de Java, estos métodos abstractos pueden tener una implementación. Esta implementación se puede llamar a través del mecanismo `super()` de la clase que lo invalida. Esto podría ser útil como un *end-point* para una super llamada en un *framework* que use herencia múltiple cooperativa.

El módulo `abc` también es compatible con los siguientes decoradores heredados:

`@abc.abstractmethod`

Nuevo en la versión 3.2.

Obsoleto desde la versión 3.3: Ahora es posible utilizar `classmethod` con `abstractmethod()`, lo cual hace que este decorador sea redundante.

Una subclase de la `classmethod()` incorporada, indicando un método de clase abstracto. De otra forma, es similar a `abstractmethod()`.

Este caso especial está obsoleto, ya que el decorador `classmethod()` ahora es identificado correctamente como abstracto cuando se aplica a un método abstracto:

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...
```

`@abc.abstractstaticmethod`

Nuevo en la versión 3.2.

Obsoleto desde la versión 3.3: Ahora es posible utilizar `staticmethod` con `abstractmethod()`, haciendo que este decorador sea redundante.

Una subclase de la `staticmethod()` incorporada, indicando un método estático abstracto. De otra forma, es similar a `abstractmethod()`.

Este caso especial está obsoleto, ya que el decorador `staticmethod()` ahora es identificado correctamente como abstracto cuando se aplica a un método abstracto:

```
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...
```

@abc.abstractmethod

Obsoleto desde la versión 3.3: Ahora es posible utilizar `property`, `property.getter()`, `property.setter()` y `property.deleter()` con `abstractmethod()`, lo cual hace que este decorador sea redundante.

Una subclase de la `property()` integrada, que indica una propiedad abstracta.

Este caso especial está obsoleto, ya que el decorador `property()` ahora es identificado correctamente como abstracto cuando es aplicado a un método abstracto:

```
class C(ABC):
    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
```

En el ejemplo anterior se define una propiedad de solo lectura; también se puede definir una propiedad abstracta de lectura y escritura marcando adecuadamente uno o varios de los métodos subyacentes como abstractos:

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
    def x(self, val):
        ...
```

Si solo algunos componentes son abstractos, solo estos componentes necesitan ser actualizados para crear una propiedad concreta en una subclase:

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

El módulo `abc` también proporciona las siguientes funciones:

abc.get_cache_token()

Retorna el token de caché de la clase base abstracta actual.

El token es un objeto opaco (que admite pruebas de igualdad) que identifica la versión actual de la caché de clases de base abstractas para subclases virtuales. El token cambia con cada llamada a `ABCMeta.register()` en cualquier CBA.

Nuevo en la versión 3.4.

Notas al pie

29.9 atexit — Gestores de Salida

El módulo `atexit` define funciones para registrar y cancelar el registro de las funciones de limpieza. Las funciones así registradas se ejecutan automáticamente cuando el intérprete se detiene normalmente. El módulo `atexit` realiza estas funciones en el orden inverso en el que se registraron; si ingresa A, B, y C, cuando el intérprete se detenga, se ejecutarán en el orden C, B, A.

Nota: Las funciones registradas a través de este módulo no se invocan cuando el programa es eliminado por una señal no gestionada por Python, cuando se detecta un error fatal interno en Python o cuando se llama a la función `os._exit()`.

Distinto en la versión 3.7: Cuando se usan con sub-intérpretes API C, las funciones registradas son locales para el intérprete en el que se registraron.

`atexit.register(func, *args, **kwargs)`

Registra *func* como una función que se ejecutará cuando el intérprete se detenga. Cualquier argumento opcional que deba pasarse a *func* debe pasarse como un argumento para la función `register()`. Es posible registrar las mismas funciones y argumentos más de una vez.

En la finalización normal del programa (por ejemplo, si se llama a la función `sys.exit()` o finaliza la ejecución del módulo principal), todas las funciones registradas se invocan en el orden último en entrar, primero en salir. Se supone que los módulos de nivel más bajo normalmente se importarán antes que los módulos de nivel alto y, por lo tanto, se limpiarán al final.

Si se lanza una excepción durante la ejecución de los gestores de salida, se muestra un seguimiento de llamada (a menos que se haya lanzado `SystemExit`) y se guarda la información de la excepción. Después de que todos los controladores de fin de programa hayan tenido la oportunidad de ejecutarse, la última excepción que se lanzó se vuelve a lanzar.

Esta función retorna *func*, lo que hace posible usarlo como decorador.

`atexit.unregister(func)`

Elimina *func* de la lista de funciones que se ejecutarán cuando el intérprete se detenga. Después de llamar a la función `unregister()`, se garantiza que *func* no se llamará cuando el intérprete se detenga, incluso si se ha registrado más de una vez. `unregister()` no hace nada y permanece en silencio en caso de que *func* no se haya registrado previamente.

Ver también:

Módulo `readline` Un ejemplo útil del uso de `atexit` para leer y escribir archivos de historial `readline`.

29.9.1 Ejemplo con atexit

El siguiente ejemplo simple muestra cómo un módulo puede inicializar un contador desde un archivo cuando se importa, y guardar el valor del contador actualizado automáticamente cuando finaliza el programa, sin necesidad de que la aplicación realice una llamada explícita en este módulo cuando el intérprete se detiene.

```
try:
    with open("counterfile") as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n
```

(continué en la próxima página)

(proviene de la página anterior)

```
def savecounter():
    with open("counterfile", "w") as outfile:
        outfile.write("%d" % _count)

import atexit
atexit.register(savecounter)
```

Los argumentos posicionales y de palabras clave también se pueden pasar a `register()` para volver a pasar a la función registrada cuando se llama:

```
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')
```

Usar como un *decorator*:

```
import atexit

@atexit.register
def goodbye():
    print("You are now leaving the Python sector.")
```

Esto solo funciona con funciones que se pueden invocar sin argumentos.

29.10 traceback — Imprimir o recuperar un seguimiento de pila

Código fuente: [Lib/traceback.py](#)

Este módulo brinda una interfaz estándar para extraer, formatear y mostrar trazas de pilas de programas de Python. Dicho módulo copia el comportamiento del intérprete de Python cuando muestra una traza de pila. Es útil a la hora de querer mostrar trazas de pilas bajo el control del programa, como si de un *wrapper* alrededor del intérprete se tratara.

El módulo utiliza objetos *traceback* — Este es el tipo de objeto que se almacena en la variable `sys.last_traceback` y es retornada como el tercer elemento de `sys.exc_info()`.

El módulo define las siguientes funciones:

`traceback.print_tb(tb, limit=None, file=None)`

Muestra hasta *limit* entradas de trazas de pila desde el objeto *traceback tb* (empezando desde el marco de la llamada) si *limit* es positivo. De lo contrario, muestra las últimas `abs(limit)` entradas. Si *limit* es omitido o `None`, todas las entradas se muestran. Si *file* es omitido o `None`, la salida va a `sys.stderr`; de lo contrario, debería ser un archivo o un objeto de tipo similar a un archivo para recibir la salida.

Distinto en la versión 3.5: Soporte para *limit* negativo añadido

`traceback.print_exception(etype, value, tb, limit=None, file=None, chain=True)`

Muestra información de excepciones y entradas de trazas de pila desde el objeto *traceback tb* a *file*. Esto difiere de `print_tb()` de las siguientes maneras:

- si *tb* no es `None`, muestra una cabecera `Traceback (most recent call last):`
- muestra la excepción *etype* y *value* después de la traza de pila

- si `type(value)` es `SyntaxError` y `value` tiene el formato apropiado, muestra la línea donde el error sintáctico ha ocurrido con un cursor indicando la posición aproximada del error.

El argumento opcional `limit` tiene el mismo significado que para `print_tb()`. Si `chain` es verdad (por defecto), entonces excepciones encadenadas (`__cause__` o `__context__` atributos de la excepción) también se imprimirán, como lo hace el propio intérprete al imprimir una excepción no controlada.

Distinto en la versión 3.5: El argumento `etype` es ignorado e infiere desde el tipo de `value`.

`traceback.print_exc(limit=None, file=None, chain=True)`

Esto es un atajo para `print_exception(*sys.exc_info(), limit, file, chain)`.

`traceback.print_last(limit=None, file=None, chain=True)`

Esto es un atajo para `print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file, chain)`. En general, solo funciona después de que una excepción ha alcanzado un `prompt` interactivo (ver `sys.last_type`).

`traceback.print_stack(f=None, limit=None, file=None)`

Muestra hasta `limit` entradas de trazas de pila (empezando desde el punto de invocación) si `limit` es positivo. De lo contrario, muestra las últimas `abs(limit)` entradas. Si `limit` es omitido o `None`, todas las entradas se muestran. El argumento opcional `f` puede ser usado para especificar un marco de pila alternativo para empezar. El argumento opcional `file` tiene el mismo significado que para `print_tb()`.

Distinto en la versión 3.5: Soporte para `limit` negativo añadido

`traceback.extract_tb(tb, limit=None)`

Retorna un objeto `StackSummary` representando una lista de entradas de trazas de pila «pre-procesadas» extraídas del objeto `traceback.tb`. Esto es útil para el formateo alternativo de trazas de pila. El argumento opcional `limit` tiene el mismo significado que para `print_tb()`. Una entrada de traza de pila «pre-procesada» es un objeto `FrameSummary` que contiene los atributos `filename`, `lineno`, `name`, y `line` representando la información que normalmente es mostrada por una traza de pila. El atributo `line` es una cadena con espacios en blanco iniciales y finales *stripped*; si la fuente no está disponible, es `None`.

`traceback.extract_stack(f=None, limit=None)`

Extrae el seguimiento de pila crudo desde el marco de pila actual. El valor retornado tiene el mismo formato que para `extract_tb()`. Los argumentos opcionales `f` y `limit` tienen el mismo significado que para `print_stack()`.

`traceback.format_list(extracted_list)`

Dada una lista de tuplas u objetos `FrameSummary` según lo retornado por `extract_tb()` o `extract_stack()`, retorna una lista de cadenas preparadas para ser mostradas. Cada cadena en la lista resultante corresponde con el elemento con el mismo índice en la lista de argumentos. Cada cadena finaliza en una nueva línea; las cadenas pueden contener nuevas líneas internas también, para aquellos elementos cuya línea de texto de origen no es `None`.

`traceback.format_exception_only(etype, value)`

Formatea la parte de excepción de un seguimiento de pila. Los argumentos son el tipo de excepción y el valor como los dados por `sys.last_type` y `sys.last_value`. El valor retornado es una lista de cadenas, acabando cada una en una nueva línea. Normalmente, la lista contiene una sola cadena; sin embargo, para las excepciones `SyntaxError`, esta lista contiene múltiples líneas que (cuando se muestran), imprimen información detallada sobre dónde ha ocurrido el error sintáctico. El mensaje indicador de qué excepción ha ocurrido es siempre la última cadena de la lista.

`traceback.format_exception(etype, value, tb, limit=None, chain=True)`

Formatea una traza de pila y la información de la excepción. Los argumentos tienen el mismo significado que los argumentos correspondientes a `print_exception()`. El valor retornado es una lista de cadenas, acabando cada una en una nueva línea y algunas contienen nuevas líneas internas. Cuando estas líneas son concatenadas y mostradas, exactamente el mismo texto es mostrado como hace `print_exception()`.

Distinto en la versión 3.5: El argumento `etype` es ignorado e infiere desde el tipo de `value`.

`traceback.format_exc(limit=None, chain=True)`

Esto es como `print_exc(limit)` pero retorna una cadena en lugar de imprimirlo en un archivo.

`traceback.format_tb(tb, limit=None)`

Un atajo para `format_list(extract_tb(tb, limit))`.

`traceback.format_stack(f=None, limit=None)`

Un atajo para `format_list(extract_stack(f, limit))`.

`traceback.clear_frames(tb)`

Limpia las variables locales de todos los marcos de pila en un seguimiento de pila *tb* llamando al método `clear()` de cada objeto de marco.

Nuevo en la versión 3.4.

`traceback.walk_stack(f)`

Recorre una pila siguiendo `f.f_back` desde el marco dado, produciendo el marco y el número de línea de cada marco. Si *f* es `None`, la pila actual es usada. Este auxiliar es usado con `StackSummary.extract()`.

Nuevo en la versión 3.5.

`traceback.walk_tb(tb)`

Recorre un seguimiento de pila siguiendo `tb.next` produciendo el marco y el número de línea de cada marco. Este auxiliar es usado con `StackSummary.extract()`.

Nuevo en la versión 3.5.

El módulo también define las siguientes clases:

29.10.1 Objetos `TracebackException`

Nuevo en la versión 3.5.

Los objetos `TracebackException` son creados a partir de excepciones reales para capturar datos para su posterior impresión de una manera ligera.

class `traceback.TracebackException(exc_type, exc_value, exc_traceback, *, limit=None, lookup_lines=True, capture_locals=False)`

Captura una excepción para su posterior procesamiento. *limit*, *lookup_lines* y *capture_locals* son como para la clase `StackSummary`.

Tenga en cuenta que cuando se capturan locales, también se muestran en el rastreo.

__cause__

Una clase `TracebackException` del original `__cause__`.

__context__

Una clase `TracebackException` del original `__context__`.

__suppress_context__

El valor `__suppress_context__` de la excepción original.

stack

Una clase `StackSummary` representando el seguimiento de pila.

exc_type

La clase del seguimiento de pila original.

filename

Para errores sintácticos - el nombre del archivo donde el error ha ocurrido.

lineno

Para errores sintácticos - el número de línea donde el error ha ocurrido.

text

Para errores sintácticos - el texto donde el error ha ocurrido.

offset

Para errores sintácticos - el *offset* en el texto donde el error ha ocurrido.

msg

Para errores sintácticos - el mensaje de error del compilador.

classmethod from_exception (*exc*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

Captura una excepción para su posterior procesamiento. *limit*, *lookup_lines* y *capture_locals* son como para la clase *StackSummary*.

Tenga en cuenta que cuando se capturan locales, también se muestran en el rastreo.

format (*, *chain=True*)

Formatea la excepción.

Si *chain* no es *True*, `__cause__` y `__context__` no serán formateados.

El valor retornado es un generador de cadenas, donde cada una acaba en una nueva línea y algunas contienen nuevas líneas internas. `print_exception()` es un contenedor alrededor de este método que simplemente muestra las líneas de un archivo.

El mensaje que indica qué excepción ocurrió siempre es la última cadena en la salida.

format_exception_only ()

Formatea la parte de la excepción de un seguimiento de pila.

El valor retornado es un generador de cadenas, donde cada una acaba en una nueva línea.

Normalmente, el generador emite una sola cadena, sin embargo, para excepciones *SyntaxError*, este emite múltiples líneas que (cuando son mostradas) imprimen información detallada sobre dónde ha ocurrido el error sintáctico.

El mensaje que indica qué excepción ocurrió siempre es la última cadena en la salida.

29.10.2 Objetos *StackSummary*

Nuevo en la versión 3.5.

Los objetos *StackSummary* representan una pila de llamadas lista para formatear.

class `traceback.StackSummary`**classmethod extract** (*frame_gen*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

Construye un objeto *StackSummary* desde un generador de marcos (tal como es retornado por `walk_stack()` o `walk_tb()`).

Si *limit* es suministrado, solo esta cantidad de cuadros son tomados de *frame_gen*. Si *lookup_lines* es *False*, los objetos *FrameSummary* retornados aún no han leído sus líneas, lo que hace que el costo de crear *StackSummary* será más barato (lo que puede ser valioso si no se formatea). Si *capture_locals* es *True*, las variables locales en cada *FrameSummary* son capturadas como representaciones de objetos.

classmethod from_list (*a_list*)

Construye un objeto *StackSummary* desde una lista suministrada de objetos *FrameSummary* o una lista antigua de tuplas. Cada tupla debe ser una 4-tupla con nombre de archivo, número de líneas, nombre, línea como los elementos.

format ()

Retorna una lista de cadenas lista para mostrarse. Cada cadena en la lista resultante corresponde a un único marco de la pila. Cada cadena termina en una nueva línea; las cadenas también pueden contener nuevas líneas internas, para aquellos elementos con líneas de texto fuente.

Para secuencias largas del mismo marco y línea, se muestran las primeras repeticiones, seguidas de una línea de resumen que indica el número exacto de repeticiones adicionales.

Distinto en la versión 3.6: Las secuencias largas de cuadros repetidos ahora se abrevian.

29.10.3 Objetos `FrameSummary`

Nuevo en la versión 3.5.

Los objetos `FrameSummary` representan un único marco en el seguimiento de pila.

class `traceback.FrameSummary` (*filename*, *lineno*, *name*, *lookup_line=True*, *locals=None*, *line=None*)

Representa un único marco en el seguimiento de pila o pila que es formateado o mostrado. Opcionalmente, puede tener una versión en cadena de los marcos locales incluidos en él. Si *lookup_line* es `False`, el código fuente no se busca hasta que `FrameSummary` tenga el atributo *line* accedido (lo que también sucede cuando lo conviertes en una tupla). *line* puede proporcionarse directamente y evitará que se realicen búsquedas de línea. *locals* es un diccionario de variables locales opcional y, si se proporciona, las representaciones de variables se almacenan en el resumen para su posterior visualización.

29.10.4 Ejemplos de seguimiento de pila

Este ejemplo sencillo implementa un bucle de lectura, evaluación e impresión básico, similar a (pero menos útil) el bucle del intérprete interactivo estándar de Python. Para una implementación más completa del bucle del intérprete, ir al módulo `code`

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)
```

El siguiente ejemplo demuestra las diferentes manera para mostrar y formatear la excepción y el seguimiento de pila:

```
import sys, traceback

def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]

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:")
    # exc_type below is ignored on 3.5 and later
    traceback.print_exception(exc_type, exc_value, exc_traceback,
                             limit=2, file=sys.stdout)

    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
```

(continué en la próxima página)

(proviene de la página anterior)

```

print(formatted_lines[0])
print(formatted_lines[-1])
print("*** format_exception:")
# exc_type below is ignored on 3.5 and later
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)

```

La salida para el ejemplo podría ser similar a esto:

```

*** 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:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in bright_side_of_death>]
*** 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

```

El siguiente ejemplo muestra las diferentes maneras de imprimir y formatear la pila:

```

>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...

```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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']
```

Este último ejemplo demuestra las últimas funciones de formateo:

```
>>> 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']
```

29.11 `__future__` — Definiciones de declaraciones futuras

Código fuente: `Lib/__future__.py`

`__future__` es un módulo real y tiene tres propósitos:

- Para evitar confundir las herramientas existentes que analizan las declaraciones de importación y esperan encontrar los módulos que están importando.
- Para garantizar que las declaraciones futuras se ejecuten en versiones anteriores a 2.1 al menos produzcan excepciones en tiempo de ejecución (la importación de `__future__` fallará, porque no había ningún módulo con ese nombre antes de 2.1).
- Documentar cuándo se introdujeron cambios incompatibles y cuándo serán — o fueron — obligatorios. Esta es una forma de documentación ejecutable y se puede inspeccionar mediante programación importando `__future__` y examinando su contenido.

Cada declaración en `__future__.py` tiene la forma:

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                        CompilerFlag)
```

donde, normalmente, *OptionalRelease* es menor que *MandatoryRelease* y ambos son 5-tuplas de la misma forma que `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 registra la primera versión en la que se aceptó la característica.

En el caso de un *MandatoryRelease* que aún no se ha producido, *MandatoryRelease* predice el lanzamiento en el que la característica pasará a formar parte del lenguaje.

De otro modo, *MandatoryRelease* registra cuándo la característica se convirtió en parte del lenguaje; en versiones en o después de este, los módulos ya no necesitan una declaración futura para usar la característica en cuestión, pero pueden continuar usando dichas importaciones.

MandatoryRelease también puede ser `None`, lo que significa que se eliminó una característica planificada.

Las instancias de la clase `_Feature` tienen dos métodos correspondientes, `getOptionalRelease()` y `getMandatoryRelease()`.

CompilerFlag es el indicador (campo de bits) que debe pasarse en el cuarto argumento a la función incorporada `compile()` para habilitar la característica en código compilado dinámicamente. Esta bandera se almacena en el atributo `compiler_flag` en las instancias `_Feature`.

Ninguna descripción de característica se eliminará de `__future__`. Desde su introducción en Python 2.1, las siguientes características han encontrado su camino en el lenguaje usando este mecanismo:

característica	opcional en	obligatorio en	efecto
nested_scopes	2.1.0b1	2.2	PEP 227 : Ámbitos anidados estáticamente
generadores	2.2.0a1	2.3	PEP 255 : Generadores simples
división	2.2.0a2	3.0	PEP 238 : Cambio de operador de división
absolute_import	2.5.0a1	3.0	PEP 328 : Importaciones: Multilínea y Absoluto/Relativo
with_statement	2.5.0a1	2.6	PEP 343 : La declaración «with»
print_function	2.6.0a2	3.0	PEP 3105 : Hacer de <code>print</code> una función
unicode_literals	2.6.0a2	3.0	PEP 3112 : Bytes literales en Python 3000
generator_stop	3.5.0b1	3.7	PEP 479 : Manejo de <code>StopIteration</code> dentro de generadores
anotaciones	3.7.0b1	3.10	PEP 563 : Evaluación pospuesta de anotaciones

Ver también:

future Cómo trata el compilador las importaciones futuras.

29.12 gc — Interfaz del recolector de basura

Este módulo proporciona una interfaz para el recolector de basura opcional (recolector de basura cíclico generacional). Proporciona la capacidad de deshabilitar el recolector, ajustar la frecuencia de recolección y establecer opciones de depuración. También proporciona acceso a objetos inaccesibles (*unreachables*) que el recolector encontró pero no pudo liberar. Dado que el recolector de basura complementa el conteo de referencias, que ya se utiliza en Python, es posible desactivarlo siempre que se esté seguro de que el programa no crea referencias cíclicas. La recolección automática se puede desactivar llamando a `gc.disable()`. Para depurar un programa con fugas de memoria, se debe llamar a `gc.set_debug(gc.DEBUG_LEAK)`. Se debe tener en cuenta que la llamada anterior incluye `gc.DEBUG_SAVEALL`, lo que hace que los objetos recolectados se guarden en `gc.garbage` para su inspección.

El módulo `gc` proporciona las siguientes funciones:

`gc.enable()`

Habilita la recolección automática de basura.

`gc.disable()`

Deshabilita la recolección automática de basura.

`gc.isenabled()`

Retorna `True` si la recolección automática está habilitada.

`gc.collect (generation=2)`

Sin argumentos, ejecuta una recolección completa. El argumento opcional *generation* debe ser un número entero que especifica qué generación recolectar (de 0 a 2). Una excepción `ValueError` será lanzada si el número de generación no es válido. Se retorna el número de objetos inaccesibles encontrados.

Las listas libres mantenidas para varios tipos incorporados son borradas cada vez que se ejecuta una recolección completa o una recolección de la generación más alta (2). No obstante, no todos los elementos de algunas listas libres pueden ser liberados, particularmente `float`, debido a su implementación particular.

`gc.set_debug (flags)`

Establece las flags de depuración para la recolección de basura. La información de depuración se escribirá en `sys.stderr`. A continuación se puede consultar una lista con las flags de depuración disponibles, que se pueden combinar mediante operaciones de bits para controlar la depuración.

`gc.get_debug ()`

Retorna las flags de depuración actualmente establecidas.

`gc.get_objects (generation=None)`

Retorna una lista de todos los objetos rastreados por el recolector, excluyendo la lista retornada. Si *generation* no es `None`, retorna solo los objetos rastreados por el recolector que pertenecen a esa generación.

Distinto en la versión 3.8: Se agregó el parámetro *generation*.

Raises an *auditing event* `gc.get_objects` with argument *generation*.

`gc.get_stats ()`

Retorna una lista de tres diccionarios, uno por cada generación, que contienen estadísticas de recolección desde el inicio del intérprete. El número de claves puede cambiar en el futuro, pero actualmente cada diccionario contendrá los siguientes elementos:

- `collections` es el número de veces que se recolectó esta generación;
- `collected` es el número total de objetos recolectados dentro de esta generación;
- `uncollectable` es el número total de objetos que se consideraron no recolectables (y por lo tanto, se movieron a la lista *garbage*) dentro de esta generación.

Nuevo en la versión 3.4.

`gc.set_threshold (threshold0[, threshold1[, threshold2]])`

Establece los umbrales de recolección (la frecuencia de recolección). Si se establece *threshold0* en cero se deshabilita la recolección.

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. With the third generation, things are a bit more complicated, see [Collecting the oldest generation](#) for more information.

`gc.get_count ()`

Retorna los recuentos de la recolección actual como una tupla de la forma (`count0`, `count1`, `count2`).

`gc.get_threshold ()`

Retorna los umbrales de la recolección actual como una tupla de la forma (`threshold0`, `threshold1`, `threshold2`).

`gc.get_referrers (*objs)`

Retorna la lista de objetos que referencian de forma directa a cualquiera de *objs*. Esta función solo localizará aquellos contenedores que admitan la recolección de basura; no se localizarán los tipos de extensión que referencian a otros objetos pero que no admiten la recolección de basura.

Téngase en cuenta que los objetos que ya se han desreferenciado, pero que tienen referencias cíclicas y aún no han sido recolectados por el recolector de basura pueden enumerarse entre las referencias resultantes. Para obtener solo los objetos activos actualmente, llame a `collect()` antes de llamar a `get_referrers()`.

Advertencia: Se debe tener un especial cuidado cuando se usan objetos retornados por `get_referrers()` dado que algunos de ellos aún podrían estar en construcción y por tanto en un estado temporalmente inválido. Debe evitarse el uso de `get_referrers()` para cualquier propósito que no sea la depuración.

Raises an *auditing event* `gc.get_referrers` with argument `objs`.

`gc.get_referents(*objs)`

Retorna una lista de objetos a los que hace referencia directamente cualquiera de los argumentos. Las referencias retornadas son aquellos objetos visitados por los métodos `tp_traverse` a nivel de C de los argumentos (si los hay) y es posible que no todos los objetos sean directamente accesibles realmente. Los métodos `tp_traverse` solo son compatibles con los objetos que admiten recolección de basura y solo se requieren para visitar objetos que pueden estar involucrados en un ciclo de referencias. Lo que quiere decir que, por ejemplo, si se puede acceder directamente a un número entero desde uno de los argumento, ese objeto entero puede aparecer o no en la lista de resultados.

Raises an *auditing event* `gc.get_referents` with argument `objs`.

`gc.is_tracked(obj)`

Retorna `True` si el recolector de basura rastrea actualmente el objeto y `False` en caso contrario. Como regla general, las instancias de tipos atómicos no se rastrean y las instancias de tipos no atómicos (contenedores, objetos definidos por el usuario ...) sí. Sin embargo, algunas optimizaciones específicas de tipo pueden estar presentes para suprimir el impacto del recolector de basura en instancias simples (por ejemplo, diccionarios que contienen solo claves y valores atómicos):

```
>>> 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
```

Nuevo en la versión 3.1.

`gc.freeze()`

Congela todos los objetos rastreados por el recolector de basura - los mueve a una generación permanente e ignora todas las recolecciones futuras. Esto se puede usar antes de una llamada a `fork()` de POSIX para hacer el recolector de basura amigable con *copy-on-write* («copiar al escribir») o para acelerar la recolección. Además, la recolección antes de una llamada `fork()` de POSIX puede liberar páginas para futuras asignaciones, lo que también puede causar *copy-on-write*, por lo que se recomienda deshabilitar el recolector de basura en el proceso principal, congelar antes de la bifurcación y habilitarlo posteriormente en el proceso secundario.

Nuevo en la versión 3.7.

`gc.unfreeze()`

Descongela los objetos de la generación permanente y vuelve a colocarlos en la generación más antigua.

Nuevo en la versión 3.7.

`gc.get_freeze_count()`

Retorna el número de objetos de la generación permanente.

Nuevo en la versión 3.7.

Las siguientes variables se proporcionan para acceso de solo lectura (se pueden mutar los valores pero no se deben vincular de nuevo):

`gc.garbage`

Una lista de objetos que el recolector consideró inaccesibles pero que no pudieron ser liberados (objetos que no se pueden recolectar). A partir de Python 3.4, esta lista debe estar vacía la mayor parte del tiempo, excepto cuando se utilizan instancias de tipos de extensión C en los que la ranura (*slot*) `tp_del` no sea `NULL`.

Si se establece `DEBUG_SAVEALL`, todos los objetos inaccesibles se agregarán a esta lista en lugar de ser liberados.

Distinto en la versión 3.2: Si la lista no está vacía en el momento del *interpreter shutdown*, se emite un `ResourceWarning`, que es silencioso de forma predeterminada. Si se establece `DEBUG_UNCOLLECTABLE`, además se imprimen todos los objetos no recolectables.

Distinto en la versión 3.4: Cumpliendo con **PEP 442**, los objetos con un método `__del__()` ya no terminan en `gc.garbage`.

`gc.callbacks`

Una lista de retrollamadas que el recolector de basura invocará antes y después de la recolección. Las retrollamadas serán llamadas con dos argumentos, *phase* e *info*.

phase puede tomar uno de estos valores:

«start»: la recolección de basura está a punto de comenzar.

«stop»: la recolección de basura ha terminado.

info es un diccionario que proporciona información adicional para la retrollamada. Las siguientes claves están definidas actualmente:

«generation»: la generación más antigua que ha sido recolectada.

«collected»: cuando *phase* es «stop», el número de objetos recolectados satisfactoriamente.

«uncollectable»: cuando *phase* es «stop», el número de objetos que no pudieron ser recolectados y fueron ubicados en `garbage`.

Las aplicaciones pueden agregar sus propias retrollamadas a esta lista. Los principales casos de uso son:

Recopilar estadísticas sobre la recolección de basura, como la frecuencia con la que se recolectan varias generaciones y cuánto tiempo lleva la recolección.

Permitir que las aplicaciones identifiquen y borren sus propios tipos no recolectables cuando aparecen en `garbage`.

Nuevo en la versión 3.3.

Las siguientes constantes son proporcionadas para ser usadas con `set_debug()`:

`gc.DEBUG_STATS`

Imprime estadísticas durante la recolección. Esta información puede resultar útil para ajustar la frecuencia de recolección.

`gc.DEBUG_COLLECTABLE`

Imprime información sobre los objetos recolectables encontrados.

`gc.DEBUG_UNCOLLECTABLE`

Imprime información sobre los objetos no recolectables encontrados (objetos inaccesibles, pero que el recolector no puede liberar). Estos objetos se agregarán a la lista `garbage`.

Distinto en la versión 3.2: También imprime el contenido de la lista `garbage` durante *interpreter shutdown*, si no está vacía.

`gc.DEBUG_SAVEALL`

Cuando se establece, todos los objetos inaccesibles encontrados se agregarán a `garbage` en lugar de ser liberados. Esto puede resultar útil para depurar un programa con fugas de memoria.

`gc.DEBUG_LEAK`

Las flags de depuración necesarias para que el recolector imprima información sobre un programa con fugas de memoria (igual a `DEBUG_COLLECTABLE` | `DEBUG_UNCOLLECTABLE` | `DEBUG_SAVEALL`).

29.13 `inspect` — Inspeccionar objetos vivos

Código fuente: [Lib/inspect.py](#)

El módulo `inspect` proporciona varias funciones útiles para ayudar a obtener información sobre objetos vivos como módulos, clases, métodos, funciones, `tracebacks`, objetos de marco y objetos de código. Por ejemplo, puede ayudarte a examinar el contenido de una clase, recuperar el código fuente de un método, extraer y dar formato a la lista de argumentos de una función, u obtener toda la información que necesitas para mostrar un `traceback` detallado.

Hay cuatro tipos principales de servicios que ofrece este módulo: comprobación de tipos, obtención del código fuente, inspección de clases y funciones, y examinar la pila del intérprete.

29.13.1 Tipos y miembros

La función `getmembers()` recupera los miembros de un objeto como una clase o un módulo. Las funciones cuyos nombres comienzan con «is» se proveen principalmente como opciones convenientes para el segundo argumento de `getmembers()`. También ayudan a determinar cuándo se puede esperar encontrar los siguientes atributos especiales:

Tipo	Atributo	Descripción
módulo	<code>__doc__</code>	cadena de caracteres de documentación
	<code>__file__</code>	nombre de archivo (falta para los módulos incorporados)
	<code>__name__</code>	nombre con el que se definió esta clase
clase	<code>__doc__</code>	cadena de caracteres de documentación
	<code>__name__</code>	nombre con el que se definió esta clase
	<code>__qualname__</code>	nombre calificado
	<code>__module__</code>	nombre del módulo en el que se definió esta clase
método	<code>__doc__</code>	cadena de caracteres de documentación
	<code>__name__</code>	nombre con el que se definió este método
	<code>__qualname__</code>	nombre calificado
	<code>__func__</code>	objeto función que contiene la implementación del método
	<code>__self__</code>	instancia a la que este método está ligado, o <code>None</code>
	<code>__module__</code>	nombre del módulo en el cual este método fue definido
función	<code>__doc__</code>	cadena de caracteres de documentación
	<code>__name__</code>	nombre con el que se definió esta función
	<code>__qualname__</code>	nombre calificado
	<code>__code__</code>	objeto de código que contiene la función compilada <i>bytecode</i>
	<code>__defaults__</code>	tupla de cualquier valor por defecto para los parámetros de posición o de palabras clave
	<code>__kwdefaults__</code>	mapeo de cualquier valor predeterminado para parámetros de sólo palabras clave
	<code>__globals__</code>	namespace global en el que se definió esta función
traceback	<code>__annotations__</code>	mapeo de los nombres de parámetros a las anotaciones; la tecla <code>"return"</code> está reservada para
	<code>__module__</code>	nombre del módulo en el cual esta función fue definida
	<code>tb_frame</code>	enmarcar el objeto a este nivel
	<code>tb_lasti</code>	índice del último intento de instrucción en código de bytes
marco	<code>tb_lineno</code>	número de línea actual en el código fuente de Python
	<code>tb_next</code>	el siguiente objeto de <code>traceback</code> interno (llamado por este nivel)
	<code>f_back</code>	el siguiente objeto exterior del marco (el que llama a este marco)
	<code>f_builtins</code>	construye el namespace visto por este marco
	<code>f_code</code>	objeto de código que se ejecuta en este marco

Cont

Tabla 1 – proviene de la página anterior

Tipo	Atributo	Descripción
	<code>f_globals</code>	el namespace global visto por este marco
	<code>f_lasti</code>	índice del último intento de instrucción en código de bytes
	<code>f_lineno</code>	número de línea actual en el código fuente de Python
	<code>f_locals</code>	el namespace local visto por este marco
	<code>f_trace</code>	función de rastreo para este marco, o <code>None</code>
code	<code>co_argcount</code>	número de argumentos (sin incluir los argumentos de palabras clave, <code>*</code> o <code>**</code> args)
	<code>co_code</code>	cadena de bytecode compilados en bruto
	<code>co_cellvars</code>	tupla de nombres de variables de celda (referenciados por contener alcances)
	<code>co_consts</code>	tupla de constantes utilizadas en el bytecode
	<code>co_filename</code>	nombre del archivo en el que este objeto código fue creado
	<code>co_firstlineno</code>	número de la primera línea del código fuente de Python
	<code>co_flags</code>	mapa de bits de los flags <code>CO_*</code> , leer más aquí
	<code>co_inotab</code>	mapeo codificado de los números de línea a los índices de bytecode
	<code>co_freevars</code>	tupla de nombres de variables libres (referenciados a través del cierre de una función)
	<code>co_posonlyargcount</code>	número de argumentos solo posicionales
	<code>co_kwonlyargcount</code>	número de argumentos de sólo palabras clave (sin incluir el <code>**</code> arg)
	<code>co_name</code>	nombre con el que se definió este objeto de código
	<code>co_names</code>	tupla de nombres de variables locales
	<code>co_nlocals</code>	número de variables locales
generador	<code>co_stacksize</code>	se requiere espacio en la pila de máquina virtual
	<code>co_varnames</code>	tupla de nombres de argumentos y variables locales
	<code>__name__</code>	nombre
	<code>__qualname__</code>	nombre calificado
	<code>gi_frame</code>	marco
	<code>gi_running</code>	¿Está el generador en ejecución?
	<code>gi_code</code>	code
	<code>gi_yieldfrom</code>	el objeto siendo iterado por <code>yield from</code> , o <code>None</code>
	<code>__name__</code>	nombre
	<code>__qualname__</code>	nombre calificado
corutina	<code>cr_await</code>	objeto al que se espera, o <code>None</code>
	<code>cr_frame</code>	marco
	<code>cr_running</code>	¿Está la corutina en ejecución?
	<code>cr_code</code>	code
	<code>cr_origin</code>	donde se creó la corutina, o <code>None</code> . Ver sys.set_coroutine_origin_tracking_d
	<code>__doc__</code>	cadena de caracteres de documentación
incorporado	<code>__name__</code>	nombre original de esta función o método
	<code>__qualname__</code>	nombre calificado
	<code>__self__</code>	instancia a la que está ligada un método, o <code>None</code>

Distinto en la versión 3.5: Agrega atributos `__qualname__` y `gi_yieldfrom` a los generadores.

El atributo `__name__` de los generadores se establece ahora a partir del nombre de la función, en lugar del nombre del código, y ahora puede ser modificado.

Distinto en la versión 3.7: Agrega el atributo `cr_origin` a las corutinas.

`inspect.getmembers(object[, predicate])`

Retorna todos los miembros de un objeto en una lista de pares (`name`, `value`) ordenados por nombre. Si se proporciona el argumento *predicate* opcional, que se llamará con el objeto `value` de cada miembro, solo se incluyen los miembros para los que el predicado retorna un valor verdadero.

Nota: `getmembers()` sólo retornará los atributos de clase definidos en la metaclass cuando el argumento sea una clase y esos atributos hayan sido listados en la costumbre de la metaclass `__dir__()`.

`inspect.getmodulename(path)`

Retorna el nombre del módulo nombrado por el *ruta* de archivo, sin incluir los nombres de los paquetes adjuntos. La extensión del archivo se comprueba con todas las entradas en `importlib.machinery.all_suffixes()`. Si coincide, el componente final de la ruta se retorna con la extensión eliminada. En caso contrario, se retorna `None`.

Ten en cuenta que esta función *sólo* retorna un nombre significativo para los módulos reales de Python - las rutas que potencialmente se refieren a los paquetes de Python seguirán retornando `None`.

Distinto en la versión 3.3: La función se basa directamente en `importlib`.

`inspect.ismodule(object)`

Retorna `True` si el objeto es un módulo.

`inspect.isclass(object)`

Retorna `True` si el objeto es una clase, ya sea incorporada o creada en código Python.

`inspect.ismethod(object)`

Retorna `True` si el objeto es un método ligado escrito en Python.

`inspect.isfunction(object)`

Retorna `True` si el objeto es una función de Python, que incluye funciones creadas por una expresión *lambda*.

`inspect.isgeneratorfunction(object)`

Retorna `True` si el objeto es una función generadora de Python.

Distinto en la versión 3.8: Funciones envueltas en `functools.partial()` ahora retornan `True` si la función envuelta es una función Python generadora.

`inspect.isgenerator(object)`

Retorna `True` si el objeto es un generador.

`inspect.iscoroutinefunction(object)`

Retorna `True` si el objeto es una *coroutine function* (una función definida con una sintaxis `async def`).

Nuevo en la versión 3.5.

Distinto en la versión 3.8: Funciones envueltas en `functools.partial()` ahora retornan `True` si la función envuelta es un *coroutine function*.

`inspect.iscoroutine(object)`

Retorna verdadero si el objeto es un *coroutine* creado por una función `async def`.

Nuevo en la versión 3.5.

`inspect.isawaitable(object)`

Retorna `True` si el objeto puede ser usado en la expresión `await`.

También se puede utilizar para distinguir las corutinas basadas en generadores de los generadores normales:

```
def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

Nuevo en la versión 3.5.

`inspect.isasyncgenfunction(object)`

Retorna `True` si el objeto es una función *asynchronous generator*, por ejemplo:

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

Nuevo en la versión 3.6.

Distinto en la versión 3.8: Funciones envueltas en `functools.partial()` ahora retornan `True` si la función envuelta es una función *asynchronous generator*.

`inspect.isasyncgen(object)`

Retorna verdadero si el objeto es un *asynchronous generator iterator* creado por una función *asynchronous generator*.

Nuevo en la versión 3.6.

`inspect.istraceback(object)`

Retorna `True` si el objeto es un `traceback`.

`inspect.isframe(object)`

Retorna `True` si el objeto es un marco.

`inspect.iscode(object)`

Retorna `True` si el objeto es un código.

`inspect.isbuiltin(object)`

Retorna `True` si el objeto es una función incorporada o un método ligado incorporado.

`inspect.isroutine(object)`

Retorna `True` si el objeto es una función o método definido por el usuario o incorporado.

`inspect.isabstract(object)`

Retorna `True` si el objeto es una clase base abstracta.

`inspect.ismethoddescriptor(object)`

Retorna `True` si el objeto es un descriptor de método, pero no si `ismethod()`, `isclass()`, `isfunction()` o `isbuiltin()` son verdaderos.

Esto, por ejemplo, es cierto para `int.__add__`. Un objeto que pasa esta prueba tiene un método `__get__()` pero no un método `__set__()`, pero más allá de eso el conjunto de atributos varía. Un atributo `__name__` suele ser sensato, y `__doc__` a menudo lo es.

Los métodos implementados a través de descriptores que también pasan una de las otras pruebas retornan `False` a la prueba `ismethoddescriptor()`, simplemente porque las otras pruebas prometen más – puede, por ejemplo, contar con tener el atributo `__func__` (etc) cuando un objeto pasa `ismethod()`.

`inspect.isdatadescriptor(object)`

Retorna `True` si el objeto es un descriptor de datos.

Los descriptores de datos tienen un método `__set__` o un método `__delete__`. Los ejemplos son propiedades (definidas en Python), conjuntos y miembros. Los dos últimos están definidos en C y hay pruebas más específicas disponibles para esos tipos, lo cual es robusto en todas las implementaciones de Python. Típicamente, los descriptores de datos también tendrán los atributos `__name__` y `__doc__` (las propiedades, conjuntos y miembros tienen ambos atributos), pero esto no está garantizado.

`inspect.isgetsetdescriptor(object)`

Retorna `True` si el objeto es un descriptor de conjunto.

CPython implementation detail: conjuntos son atributos definidos en módulos de extensión a través de estructuras `PyGetSetDef`. Para implementaciones de Python sin tales tipos, este método siempre retornará `False`.

`inspect.ismemberdescriptor(object)`

Retorna `True` si el objeto es un descriptor de miembro.

CPython implementation detail: Los descriptores de miembros son atributos definidos en los módulos de extensión a través de las estructuras `PyMemberDef`. Para implementaciones de Python sin tales tipos, este método siempre retornará `False`.

29.13.2 Recuperar el código fuente

`inspect.getdoc(object)`

Obtener la cadena de documentación de un objeto, limpiado con `cleandoc()`. Si no se proporciona la cadena de documentación de un objeto y el objeto es una clase, un método, una propiedad o un descriptor, recupera la cadena de documentación de la jerarquía de la herencia.

Distinto en la versión 3.5: Las cadenas de documentación son ahora heredadas, si no anuladas.

`inspect.getcomments(object)`

Retorna en una sola cadena las líneas de comentarios que preceden inmediatamente al código fuente del objeto (para una clase, función o método), o en la parte superior del archivo fuente de Python (si el objeto es un módulo). Si el código fuente del objeto no está disponible, retorna `None`. Esto podría suceder si el objeto ha sido definido en C o en el shell interactivo.

`inspect.getfile(object)`

Retorna el nombre del archivo (de texto o binario) en el que se definió un objeto. Esto fallará con un `TypeError` si el objeto es un módulo, clase o función incorporada.

`inspect.getmodule(object)`

Intenta adivinar en qué módulo se definió un objeto.

`inspect.getsourcefile(object)`

Retorna el nombre del archivo fuente de Python en el que se definió un objeto. Esto fallará con un `TypeError` si el objeto es un módulo, clase o función incorporada.

`inspect.getsourcelines(object)`

Retorna una lista de líneas de origen y el número de línea de inicio de un objeto. El argumento puede ser un objeto módulo, clase, método, función, traceback, marco o código. El código fuente es retornado como una lista de las líneas correspondientes al objeto y el número de línea que indica dónde se encontró la primera línea de código en el archivo fuente original. Un `OSError` es lanzado si el código fuente no puede ser recuperado.

Distinto en la versión 3.3: `OSError` se eleva en lugar de `IOError`, ahora un alias del primero.

`inspect.getsource(object)`

Retorna el texto del código fuente de un objeto. El argumento puede ser un objeto de módulo, clase, método, función, rastreo, marco o código. El código fuente se retorna como una sola cadena. Un `OSError` es lanzado si el código fuente no puede ser recuperado.

Distinto en la versión 3.3: `OSError` se eleva en lugar de `IOError`, ahora un alias del primero.

`inspect.cleandoc(doc)`

Limpiar la indentación de los docstrings que están indentados para alinearse con los bloques de código.

Todos los espacios blancos principales se eliminan de la primera línea. Cualquier espacio blanco principal que pueda ser uniformemente removido de la segunda línea en adelante es removido. Las líneas vacías al principio y al final se eliminan posteriormente. Además, todas las pestañas se expanden a los espacios.

29.13.3 Introspección de los invocables con el objeto Signature

Nuevo en la versión 3.3.

El objeto Signature representa la firma de llamada de un objeto invocable y su anotación de retorno. Para recuperar un objeto Signature, utilice la función `signature()`.

`inspect.signature(callable, *, follow_wrapped=True)`

Retorna un objeto `Signature` para el callable dado:

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass
>>> sig = signature(foo)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> str(sig)
'(a, *, b:int, **kwargs)'

>>> str(sig.parameters['b'])
'b:int'

>>> sig.parameters['b'].annotation
<class 'int'>
```

Acepta un amplio rango de invocables de Python, desde funciones y clases simples hasta objetos *functools.partial()*.

Lanza *ValueError* si no se puede proporcionar un signature, y *TypeError* si ese tipo de objeto no es soportado.

Una barra (/) en la signature de una función denota que los parámetros anteriores a ella son sólo posicionales. Para más información, ver la pregunta frecuente en parámetros solo posicionales.

Nuevo en la versión 3.5: parámetro *follow_wrapped*. Pasa *False* para obtener un signature de callable específicamente (*callable.__wrapped__* no se usará para desenvolver los invocables decorados.)

Nota: Algunos invocables pueden no ser introspeccionables en ciertas implementaciones de Python. Por ejemplo, en CPython, algunas funciones incorporadas definidas en C no proporcionan metadatos sobre sus argumentos.

class `inspect.Signature` (*parameters=None*, *, *return_annotation=Signature.empty*)

Un objeto *Signature* representa la firma de llamada de una función y su anotación de retorno. Por cada parámetro aceptado por la función, almacena un objeto *Parameter* en su colección *parameters*.

El argumento opcional *parámetros* es una secuencia de objetos *Parameter*, que se valida para comprobar que no hay parámetros con nombres duplicados, y que los parámetros están en el orden correcto, es decir, primero sólo de posición, luego de posición o palabra clave, y que los parámetros con valores por defecto siguen a los parámetros sin valores por defecto.

El argumento opcional *return_annotation*, puede ser un objeto Python arbitrario, es la anotación «return» del invocable.

Los objetos signature son *inmutables*. Usar *Signature.replace()* para hacer una copia modificada.

Distinto en la versión 3.5: Los objetos *Signature* se pueden seleccionar y se pueden manipular.

empty

Un marcador especial de clase para especificar la ausencia de una anotación de retorno.

parameters

Un mapeo ordenado de los nombres de los parámetros a los correspondientes objetos *Parameter*. Los parámetros aparecen en estricto orden de definición, incluyendo parámetros de sólo palabras clave.

Distinto en la versión 3.7: Python sólo garantizó explícitamente que conservaba el orden de declaración de los parámetros de sólo palabras clave a partir de la versión 3.7, aunque en la práctica este orden siempre se había conservado en Python 3.

return_annotation

La anotación de «retorno» para el invocable. Si el invocable no tiene ninguna anotación de «return», este atributo se establece en *Signature.empty*.

bind(*args, **kwargs)

Crear un mapeo de argumentos posicionales y de palabras clave a los parámetros. Retorna *BoundArguments* si **args* y ***kwargs* coinciden con el signature, o lanza un *TypeError*.

bind_partial (*args, **kwargs)

Funciona de la misma manera que `Signature.bind()`, pero permite la omisión de algunos argumentos requeridos (imita el comportamiento de `functools.partial()`.) Retorna `BoundArguments`, o lanza un `TypeError` si los argumentos pasados no coinciden con la firma.

replace ([, parameters][, return_annotation])

Crear una nueva instancia `Signature` basada en la instancia sobre la que se invocó el reemplazo. Es posible pasar diferentes parámetros y/o `return_annotation` para sobrescribir las propiedades correspondientes de la firma base. Para eliminar `return_annotation` del `Signature` copiado, pasar en `Signature.empty`.

```
>>> def test(a, b):
...     pass
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

classmethod from_callable (obj, *, follow_wrapped=True)

Retorna un objeto `Signature` (o su subclase) para un determinado `obj` invocable. Pasa `follow_wrapped=False` para obtener una firma de `obj` sin desenvolver su cadena `__wrapped__`.

Este método simplifica la subclasificación de `Signature`:

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(min)
assert isinstance(sig, MySignature)
```

Nuevo en la versión 3.5.

class inspect.Parameter (name, kind, *, default=Parameter.empty, annotation=Parameter.empty)

Los objetos parámetros son *inmutables*. En lugar de modificar un objeto Parámetro, puedes usar `Parameter.replace()` para crear una copia modificada.

Distinto en la versión 3.5: Los objetos Parámetro se pueden seleccionar y manipular.

empty

Un marcador especial de clase para especificar la ausencia de valores predeterminados y anotaciones.

name

El nombre del parámetro como una cadena. El nombre debe ser un identificador Python válido.

CPython implementation detail: CPython genera nombres de parámetros implícitos de la forma `.0` en los objetos de código utilizados para implementar expresiones de comprensiones y generadores.

Distinto en la versión 3.6: Los nombres de estos parámetros son expuestos por este módulo como nombres como `implicit0`.

default

El valor por defecto del parámetro. Si el parámetro no tiene un valor por defecto, este atributo se establece en `Parameter.empty`.

annotation

La anotación para el parámetro. Si el parámetro no tiene ninguna anotación, este atributo se establece como `Parameter.empty`.

kind

Describe cómo los valores de los argumentos están vinculados al parámetro. Valores posibles (accesibles a través de `Parameter`, como `Parameter.KEYWORD_ONLY`):

Nombre	Significado
<i>POSITIONAL_ONLY</i>	El valor debe proporcionarse como un argumento posicional. Los parámetros solo posicionales son aquellos que aparecen antes de una entrada / (si está presente) en una definición de función de Python. aceptan sólo uno o dos parámetros) los aceptan.
<i>POSITIONAL_OR_KEYWORD</i>	El valor puede ser suministrado como una palabra clave o como un argumento posicional (este es el comportamiento estándar de unión para las funciones implementadas en Python)
<i>VAR_POSITIONAL</i>	Una tupla de argumentos posicionales que no están ligados a ningún otro parámetro. Esto corresponde a un parámetro <i>*args</i> en una definición de función Python.
<i>KEYWORD_ONLY</i>	El valor debe ser suministrado como argumento de la palabra clave. Los parámetros de sólo palabras clave son los que aparecen después de una entrada <i>*</i> o <i>*args</i> en una definición de función Python.
<i>VAR_KEYWORD</i>	Un dictado de argumentos de palabras clave que no están ligadas a ningún otro parámetro. Esto corresponde a un parámetro <i>**kwargs</i> en una definición de función Python.

Ejemplo: imprimir todos los argumentos de sólo palabras clave sin valores por defecto:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...         param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

kind.description

Describe un valor enum como *Parameter.kind*.

Nuevo en la versión 3.8.

Ejemplo: imprimir todas las descripciones de los argumentos:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     print(param.kind.description)
positional or keyword
positional or keyword
keyword-only
keyword-only
```

replace (**[, name][, kind][, default][, annotation]*)

Crear una nueva instancia de parámetros basada en la instancia en la que se invocó la sustitución. Para anular un atributo *Parameter*, pasa el argumento correspondiente. Para eliminar un valor por defecto y/o una anotación de un parámetro, pasa *Parameter.empty*.

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
'foo=42'
```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> str(param.replace()) # Will create a shallow copy of 'param'
'foo=42'

>>> str(param.replace(default=Parameter.empty, annotation='spam'))
"foo:'spam'"
```

Distinto en la versión 3.4: En Python 3.3 se permitía que los objetos `Parámetro` tuvieran el `name` puesto en `None` si su `kind` estaba puesto en `POSITIONAL_ONLY`. Esto ya no está permitido.

class inspect.BoundArguments

Resultado de una llamada `Signature.bind()` o `Signature.bind_partial()`. Mantiene el mapeo de los argumentos a los parámetros de la función.

arguments

Un mapeo ordenado y mutable (`collections.OrderedDict`) de los nombres de los parámetros a los valores de los argumentos. Contiene sólo argumentos explícitamente vinculados. Los cambios en `arguments` se reflejarán en `args` y `kwargs`.

Debe ser usado en conjunto con `Signature.parameters` para cualquier propósito de procesamiento de argumentos.

Nota: Los argumentos para los cuales `Signature.bind()` o `Signature.bind_partial()` se basaban en un valor por defecto se saltan. Sin embargo, si es necesario, use `BoundArguments.apply_defaults()` para añadirlos.

args

Una tupla de valores de argumentos posicionales. Calculados dinámicamente a partir del atributo `arguments`.

kwargs

Un diccionario de valores de argumentos de palabras clave. Calculados dinámicamente a partir del atributo `arguments`.

signature

Una referencia al objeto padre `Signature`.

apply_defaults()

Establece valores por defecto para los argumentos que faltan.

Para los argumentos de posición variable (`*args`) el valor por defecto es una tupla vacía.

Para los argumentos de palabras clave variables (`**kwargs`) el valor por defecto es un diccionario vacío.

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
OrderedDict([('a', 'spam'), ('b', 'ham'), ('args', ())])
```

Nuevo en la versión 3.5.

Las propiedades `args` y `kwargs` pueden ser usadas para invocar funciones:

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

Ver también:

PEP 362 - Función Objeto Signature. La especificación detallada, los detalles de implementación y los ejemplos.

29.13.4 Clases y funciones

`inspect.getclasstree(classes, unique=False)`

Organizar la lista de clases dada en una jerarquía de listas anidadas. Cuando aparece una lista anidada, contiene clases derivadas de la clase cuya entrada precede inmediatamente a la lista. Cada entrada es una tupla de 2 valores que contienen una clase y una tupla de sus clases base. Si el argumento *unique* es cierto, aparece exactamente una entrada en la estructura retornada para cada clase de la lista dada. De lo contrario, las clases que utilizan la herencia múltiple y sus descendientes aparecerán varias veces.

`inspect.getargspec(func)`

Obtener los nombres y valores por defecto de los parámetros de una función de Python. A *named tuple* `ArgSpec(args, varargs, keywords, defaults)` es retornado. *args* es una lista de los nombres de los parámetros. *varargs* y *keywords* son los nombres de los parámetros * y ** o `None`. *defaults* es una tupla de valores de los argumentos por defecto o `None` si no hay argumentos por defecto; si esta tupla tiene *n* elementos, corresponden a los últimos *n* elementos listados en *args*.

Obsoleto desde la versión 3.0: Use `getfullargspec()` para una API actualizada que suele ser un sustituto de la que no se necesita, pero que también maneja correctamente las anotaciones de la función y los parámetros de sólo palabras clave.

Alternativamente, use `signature()` y *Objeto Signature*, que proporcionan una API de introspección más estructurada para los invocables.

`inspect.getfullargspec(func)`

Obtener los nombres y valores por defecto de los parámetros de una función de Python. Se retorna un *named tuple*:

```
FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults,
             annotations)
```

args es una lista de los nombres de los parámetros posicionales. *varargs* es el nombre del parámetro * o `None` si no se aceptan argumentos posicionales arbitrarios. *varkw* es el nombre del parámetro ** o `None` si no se aceptan argumentos de palabras clave arbitrarias. *defaults* es una *n*-tupla de valores de argumentos por defecto que corresponden a los últimos parámetros de posición *n*, o `None` si no hay tales valores por defecto definidos. *kwonlyargs* es una lista de nombres de parámetros de sólo palabras clave en orden de declaración. *kwonlydefaults* es un diccionario que asigna los nombres de los parámetros de *kwonlyargs* a los valores por defecto utilizados si no se suministra ningún argumento. *annotations* es un diccionario que asigna los nombres de los parámetros a las anotaciones. La tecla especial "return" se utiliza para informar de la anotación del valor de retorno de la función (si existe).

Observe que `signature()` y *Objeto Signature* proporcionan la API recomendada para la introspección invocable, y soportan comportamientos adicionales (como los argumentos de sólo posición) que a veces se encuentran en las API de los módulos de extensión. Esta función se conserva principalmente para su uso en el código que necesita mantener la compatibilidad con la API de módulos de `inspect` de Python 2.

Distinto en la versión 3.4: Esta función se basa ahora en `signature()`, pero sigue ignorando los atributos `__wrapped__` e incluye el primer parámetro ya ligado en la salida del `signature` para los métodos ligados.

Distinto en la versión 3.6: Este método fue documentado anteriormente como obsoleto en favor de `signature()` en Python 3.5, pero esa decisión ha sido revocada para restaurar una interfaz estándar claramente soportada para el código de una sola fuente en Python 2/3 que se aleja de la API heredada `getargspec()`.

Distinto en la versión 3.7: Python sólo garantizó explícitamente que conservaba el orden de declaración de los parámetros de sólo palabras clave a partir de la versión 3.7, aunque en la práctica este orden siempre se había conservado en Python 3.

`inspect.getargvalues(frame)`

Obtener información sobre los argumentos pasados en un marco particular. Un *named tuple* `ArgInfo(args, varargs, keywords, locals)` es retornado. *args* es una lista de los nombres de los argumentos. *varargs* y *keywords* son los nombres de los argumentos * y ** o `None`. *locals* es el diccionario local del marco dado.

Nota: Esta función fue inadvertidamente marcada como obsoleta en Python 3.5.

`inspect.formatargspec` (*args*[, *varargs*, *varkw*, *defaults*, *kwonlyargs*, *kwonlydefaults*, *annotations*[, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*, *formatreturns*, *formatannotations*]])

Formatea un bonito argumento de los valores retornados por `getfullargspec()`.

Los primeros siete argumentos son (*args*, *varargs*, *varkw*, *defaults*, *kwonlyargs*, *kwonlydefaults*, *annotations*).

Los otros seis argumentos son funciones que se llaman para convertir los nombres de los argumentos, el nombre del argumento *, el nombre del argumento **, los valores por defecto, la anotación de retorno y las anotaciones individuales en cadenas, respectivamente.

Por ejemplo:

```
>>> from inspect import formatargspec, getfullargspec
>>> def f(a: int, b: float):
...     pass
...
>>> formatargspec(*getfullargspec(f))
'(a: int, b: float)'
```

Obsoleto desde la versión 3.5: Usar `signature()` y *Objeto Signature*, que proporcionan un mejor API de introspección para los invocables.

`inspect.formatargvalues` (*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*])

Formatea una bonita especificación de argumentos de los cuatro valores retornados por `getargvalues()`. Los argumentos de formato* son las correspondientes funciones de formato opcionales que se llaman para convertir nombres y valores en cadenas.

Nota: Esta función fue inadvertidamente marcada como obsoleta en Python 3.5.

`inspect.getmro` (*cls*)

Retorna una tupla de clases base de *cls*, incluyendo *cls*, en orden de resolución de métodos. Ninguna clase aparece más de una vez en esta tupla. Obsérvese que el orden de resolución de los métodos depende del tipo de *cls*. A menos que se utilice un meta tipo muy peculiar definido por el usuario, *cls* será el primer elemento de la tupla.

`inspect.getcallargs` (*func*, *l*, **args*, ***kwds*)

Ata los *args* y *kwds* a los nombres de los argumentos de la función o método *func* de Python, como si se llamara con ellos. Para los métodos ligados, liga también el primer argumento (típicamente llamado `self`) a la instancia asociada. Se retorna un diccionario, mapeando los nombres de los argumentos (incluyendo los nombres de los argumentos * y **, si los hay) a sus valores de *args* y *kwds*. En caso de invocar *func* incorrectamente, es decir, siempre que `func(*args, **kwds)` plantee una excepción por firma incompatible, se plantea una excepción del mismo tipo y del mismo o similar mensaje. Por ejemplo:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'
```

Nuevo en la versión 3.2.

Obsoleto desde la versión 3.5: Usa `Signature.bind()` y `Signature.bind_partial()` en su lugar.

`inspect.getclosurevars(func)`

Obtener el mapeo de referencias de nombres externos en una función o método *func* de Python a sus valores actuales. Un *named tuple* `ClosureVars(nonlocals, globals, builtins, unbound)` es retornado. *nonlocals* asigna los nombres referidos a las variables de cierre léxicas, *globals* a los globals de los módulos de la función y *builtins* a los builtins visibles desde el cuerpo de la función. *unbound* es el conjunto de nombres referenciados en la función que no pudieron ser resueltos en absoluto dados los actuales globals y builtins del módulo.

`TypeError` es lanzado si *func* no es una función o método de Python.

Nuevo en la versión 3.3.

`inspect.unwrap(func, *, stop=None)`

Obtiene el objeto envuelto por *func*. Sigue la cadena de atributos `__wrapped__` retornando el último objeto de la cadena.

stop es una retrollamada opcional que acepta un objeto de la cadena envuelta como único argumento que permite terminar el desenvolvolvimiento antes de tiempo si la retrollamada retorna un valor real. Si la retrollamada nunca retorna un valor verdadero, el último objeto de la cadena se retorna como de costumbre. Por ejemplo, `signature()` utiliza esto para detener el desenvolvolvimiento si algún objeto de la cadena tiene definido el atributo `__signature__`.

`ValueError` es lanzado si se encuentra un ciclo.

Nuevo en la versión 3.4.

29.13.5 La pila del interprete

Cuando las siguientes funciones retornan «registros de cuadro», cada registro es un *named tuple* `FrameInfo(frame, filename, lineno, function, code_context, index)`. La *tuple* contiene el objeto marco, el nombre de archivo, el número de línea de la línea actual, el nombre de la función, una lista de líneas de contexto del código fuente, y el índice de la línea actual dentro de esa lista.

Distinto en la versión 3.5: Retorna una *tuple* con nombre en lugar de una *tuple*.

Nota: Mantener referencias a los objetos marco, como se encuentra en el primer elemento de los registros marco que estas funciones retornan, puede hacer que su programa cree ciclos de referencia. Una vez creado un ciclo de referencia, la vida útil de todos los objetos a los que se puede acceder desde los objetos que forman el ciclo puede ser mucho mayor, incluso si el detector de ciclos opcional de Python está activado. Si es necesario crear tales ciclos, es importante asegurarse de que se rompen explícitamente para evitar la destrucción retardada de los objetos y el aumento del consumo de memoria que se produce.

Aunque el detector de ciclos los captará, la destrucción de los marcos (y las variables locales) puede hacerse determinísticamente eliminando el ciclo en una cláusula de `finally`. Esto también es importante si el detector de ciclos fue desactivado cuando se compiló Python o usando `gc.disable()`. Por ejemplo:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

Si quieres mantener el marco alrededor (por ejemplo para imprimir una traceback más tarde), también puedes romper los ciclos de referencia utilizando el método `frame.clear()`.

El argumento opcional de *context*, apoyado por la mayoría de estas funciones, especifica el número de líneas de contexto a retornar, que se centran en la línea actual.

`inspect.getframeinfo(frame, context=1)`

Obtener información sobre un marco o un objeto de traceback. Un *named tuple* `Traceback(filename, lineno, function, code_context, index)` es retornado.

`inspect.getouterframes(frame, context=1)`

Obtener una lista de registros marco para un marco y de todos los marcos exteriores. Estos marcos representan las llamadas que conducen a la creación de *frame*. La primera entrada de la lista retornada representa *frame*; la última entrada representa la llamada más exterior en la pila de *frame*.

Distinto en la versión 3.5: Una lista de *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` es retornada.

`inspect.getinnerframes(traceback, context=1)`

Consigue una lista de registros de marcos para un marco de traceback y todos los marcos internos. Estos marcos representan llamadas hechas como consecuencia de *frame*. La primera entrada de la lista representa *traceback*; la última entrada representa donde se lanzó la excepción.

Distinto en la versión 3.5: Una lista de *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` es retornada.

`inspect.currentframe()`

Retorna el objeto marco para el marco de la pila del que llama.

CPython implementation detail: Esta función se basa en el soporte del marco de la pila de Python en el intérprete, que no está garantizado que exista en todas las implementaciones de Python. Si se ejecuta en una implementación sin soporte de marcos de pila de Python, esta función retorna `None`.

`inspect.stack(context=1)`

Retorna una lista de registros de marco para la pila del que llama. La primera entrada de la lista retornada representa al que llama; la última entrada representa la llamada más exterior de la pila.

Distinto en la versión 3.5: Una lista de *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` es retornada.

`inspect.trace(context=1)`

Retorna una lista de registros de marco para la pila entre el marco actual y el marco en la que se lanzó una excepción que se está manejando actualmente. La primera entrada de la lista representa al que llama; la última entrada representa el lugar donde se lanzó la excepción.

Distinto en la versión 3.5: Una lista de *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` es retornada.

29.13.6 Obteniendo atributos estáticamente

Tanto `getattr()` como `hasattr()` pueden desencadenar la ejecución del código al buscar o comprobar la existencia de atributos. Los descriptores, como las propiedades, serán invocados y se podrá llamar a `__getattr__()` y `__getattribute__()`.

Para los casos en los que se quiera una introspección pasiva, como las herramientas de documentación, esto puede ser un inconveniente. `getattr_static()` tiene la misma firma que `getattr()` pero evita la ejecución de código cuando obtiene atributos.

`inspect.getattr_static(obj, attr, default=None)`

Recuperar los atributos sin activar la búsqueda dinámica a través del protocolo descriptor, `__getattr__()` o `__getattribute__()`.

Nota: es posible que esta función no pueda recuperar todos los atributos que `getattr` puede recuperar (como los atributos creados dinámicamente) y puede encontrar atributos que `getattr` no puede (como los descriptores que lanzan `AttributeError`). También puede retornar objetos descriptores en lugar de miembros de la instancia.

Si la instancia `__dict__` es ensombrecida por otro miembro (por ejemplo una propiedad) entonces esta función no podrá encontrar miembros de la instancia.

Nuevo en la versión 3.2.

`getattr_static()` no resuelve los descriptores, por ejemplo los descriptores de ranura o los descriptores de `getset` en los objetos implementados en C. El objeto descriptor se retorna en lugar del atributo subyacente.

Puedes manejar esto con un código como el siguiente. Tenga en cuenta que la invocación de los descriptores de `getset` arbitrarios pueden desencadenar la ejecución del código:

```
# example code for resolving the builtin descriptor types
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
        # in which case the descriptor itself will
        # have to do
        pass
```

29.13.7 Estado actual de los Generadores y las Corutinas

Al implementar los programadores de corutinas y para otros usos avanzados de los generadores, es útil determinar si un generador se está ejecutando actualmente, si está esperando para iniciarse o reanudarse o si ya ha terminado. `getgeneratorstate()` permite determinar fácilmente el estado actual de un generador.

`inspect.getgeneratorstate(generator)`
Obtener el estado actual de un generador-iterador.

Los posibles estados son:

- `GEN_CREATED`: Esperando para iniciar la ejecución.
- `GEN_RUNNING`: Actualmente está siendo ejecutado por el intérprete.
- `GEN_SUSPENDED`: Actualmente suspendido en una expresión `yield`.
- `GEN_CLOSED`: La ejecución se ha completado.

Nuevo en la versión 3.2.

`inspect.getcoroutinestate(coroutine)`
Obtener el estado actual de un objeto de corutina. La función está pensada para ser usada con objetos de corutina creados por las funciones `async def`, pero aceptará cualquier objeto de corutina que tenga los atributos `cr_running` y `cr_frame`.

Los posibles estados son:

- `CORO_CREATED`: Esperando para iniciar la ejecución.
- `CORO_RUNNING`: Actualmente está siendo ejecutado por el intérprete.
- `CORO_SUSPENDED`: Actualmente suspendido en una expresión de espera.
- `CORO_CLOSED`: La ejecución se ha completado.

Nuevo en la versión 3.5.

También se puede consultar el estado interno actual del generador. Esto es mayormente útil para fines de prueba, para asegurar que el estado interno se actualiza como se espera:

`inspect.getgeneratorlocals(generator)`

Consigue el mapeo de las variables vivas locales en *generator* a sus valores actuales. Se retorna un diccionario que mapea de los nombres de las variables a los valores. Esto es el equivalente a llamar `locals()` en el cuerpo del generador, y se aplican todas las mismas advertencias.

Si *generator* es un *generator* sin marco asociado actualmente, entonces se retorna un diccionario vacío. `TypeError` es lanzado si *generator* no es un objeto generador de Python.

CPython implementation detail: Esta función se basa en que el generador exponga un marco de pila de Python para la introspección, lo cual no está garantizado en todas las implementaciones de Python. En tales casos, esta función siempre retornará un diccionario vacío.

Nuevo en la versión 3.3.

`inspect.getcoroutinelocals(coroutine)`

Esta función es análoga a `getgeneratorlocals()`, pero funciona para los objetos de corutina creados por funciones `async def`.

Nuevo en la versión 3.5.

29.13.8 Objetos de código Bit Flags

Los objetos de código Python tienen un atributo `co_flags`, que es un mapa de bits de los siguientes flags:

`inspect.CO_OPTIMIZED`

El objeto del código está optimizado, usando locales rápidas (*fast locals*).

`inspect.CO_NEWLOCALS`

Si se establece, se creará un nuevo diccionario para el marco `f_locals` cuando se ejecute el objeto código.

`inspect.CO_VARARGS`

El objeto del código tiene un parámetro posicional variable (similar a `*args`).

`inspect.CO_VARKEYWORDS`

El objeto del código tiene un parámetro de palabra clave variable (similar a `**kwargs`).

`inspect.CO_NESTED`

El flag se fija cuando el objeto del código es una función anidada.

`inspect.CO_GENERATOR`

El flag se fija cuando el objeto del código es una función generadora, es decir, un objeto generador es retornado cuando el objeto del código se ejecuta.

`inspect.CO_NOFREE`

El flag se configura si no hay variables libres o de celda.

`inspect.CO_COROUTINE`

El flag se configura cuando el objeto del código es una función de corutina. Cuando el objeto código se ejecuta, retorna un objeto de corutina. Ver [PEP 492](#) para más detalles.

Nuevo en la versión 3.5.

`inspect.CO_ITERABLE_COROUTINE`

El flag se utiliza para transformar generadores en corutinas basadas en generadores. Los objetos generadores con este flag pueden ser usados en la expresión `await`, y objetos de corutina `yield from`. Ver [PEP 492](#) para más detalles.

Nuevo en la versión 3.5.

`inspect.CO_ASYNC_GENERATOR`

El flag se configura cuando el objeto del código es una función generadora asíncrona. Cuando el objeto código se ejecuta, retorna un objeto generador asíncrono. Ver [PEP 525](#) para más detalles.

Nuevo en la versión 3.6.

Nota: Los flags son específicos de CPython, y no pueden ser definidas en otras implementaciones de Python. Además, los flags son un detalle de la implementación, y pueden ser eliminados o desaprobados en futuras versiones de Python. Se recomienda utilizar las APIs públicas del módulo `inspect` para cualquier necesidad de introspección.

29.13.9 Interfaz de la línea de comando

El módulo `inspect` también proporciona una capacidad básica de introspección desde la línea de comandos.

Por defecto, acepta el nombre de un módulo e imprime la fuente de ese módulo. Una clase o función dentro del módulo puede imprimirse en su lugar añadiendo dos puntos y el nombre calificado del objeto objetivo.

--details

Imprimir información sobre el objeto especificado en lugar del código fuente

29.14 `site` — Enlace (*hook*) de configuración específico del sitio

Código Fuente: `Lib/site.py`

Este módulo se importa automáticamente durante la inicialización. La importación automática se puede suprimir utilizando la opción del intérprete opción `-S`.

La importación de este módulo agregará rutas específicas del sitio a la ruta de búsqueda del módulo y agregará algunas incorporaciones, a menos que `-S` se haya utilizado. En este caso, este módulo se puede importar de forma segura sin modificaciones automáticas en la ruta de búsqueda del módulo o adiciones a los módulos incorporados. Para activar explícitamente las adiciones habituales específicas del sitio, llame a `site.principal()` función.

Distinto en la versión 3.3: Importar el módulo utilizado para activar la manipulación de rutas incluso cuando se utiliza `-S`.

Comienza construyendo hasta cuatro directorios a partir de una parte de cabeza y otra de cola. Para la parte de la cabeza, usa `sys.prefix` y `sys.exec_prefix`; se saltan las cabezas vacías. Para la parte de la cola, usa la cadena vacía y luego `lib/site-packages` (en Windows) o usa `lib/pythonX.Y/site-packages` (en Unix y Macintosh). Para cada una de las distintas combinaciones de cabeza y cola, ve si se refiere a un directorio existente y, de ser así, lo agrega a `sys.path` y también inspecciona la ruta recién agregada para los archivos de configuración.

Distinto en la versión 3.5: Se ha eliminado la compatibilidad con el directorio «site-python».

Si un archivo llamado «pyenv.cfg» existe un directorio por encima de `sys.executable`, `sys.prefix` y `sys.exec_prefix` se establecen en ese directorio y también se comprueba si hay paquetes de sitio (`sys.base_prefix` y `sys.base_exec_prefix` siempre serán los prefijos «reales» de la instalación de Python). Si «pyenv.cfg» (un archivo de configuración de arranque) contiene la clave «include-system-site-packages» configurada con un valor diferente a «true» (no distingue entre mayúsculas y minúsculas), no se buscarán los prefijos de nivel de sistema para paquetes de sitio; de lo contrario lo harán.

Un archivo de configuración de ruta es un archivo cuyo nombre tiene la forma `name.pth` y existe en uno de los cuatro directorios mencionados anteriormente; su contenido son elementos adicionales (uno por línea) que se agregarán a `sys.path`. Los elementos que no existen nunca se agregan a `sys.path` y no se comprueba que el elemento se refiera a un directorio en lugar de a un archivo. No se agrega ningún elemento a `sys.path` más de una vez. Se omiten las líneas en blanco y las que comienzan con `#`. Se ejecutan las líneas que comienzan con `import` (seguidas de un espacio o tabulación).

Nota: Una línea ejecutable en un archivo `.pth` se ejecuta en cada inicio de Python, independientemente de si se va a utilizar un módulo en particular. Por tanto, su impacto debería reducirse al mínimo. El propósito principal de las líneas ejecutables es hacer que los módulos correspondientes se puedan importar (cargar ganchos de importación de terceros, ajustar `PATH`, etc.). Se supone que cualquier otra inicialización se debe realizar en la importación real

de un módulo, siempre y cuando ocurra. Limitar un fragmento de código a una sola línea es una medida deliberada para desalentar la inclusión de algo más complejo aquí.

Por ejemplo, supongamos que `sys.prefix` y `sys.exec_prefix` están configurados en `/usr/local`. La biblioteca Python X.Y se instala en `/usr/local/lib/pythonX.Y`. Supongamos que tiene un subdirectorio `/usr/local/lib/pythonX.Y/site-packages` con tres subsubdirectorios, `foo`, `bar` y `spam`, y dos archivos de configuración de ruta, `foo.pth` y `bar.pth`. Suponga `foo.pth` contiene lo siguiente:

```
# foo package configuration

foo
bar
bletch
```

y `bar.pth` contiene:

```
# bar package configuration

bar
```

Luego, los siguientes directorios específicos de la versión se agregan a `sys.path`, en este orden:

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

Tenga en cuenta que `bletch` se omite porque no existe; el directorio `bar` precede al directorio `foo` porque `bar.pth` viene alfabéticamente antes de `foo.pth`; y `spam` se omite porque no se menciona en ninguno de los archivos de configuración de ruta.

Después de estas manipulaciones de ruta, se intenta importar un módulo llamado `sitecustomize`, que puede realizar personalizaciones arbitrarias específicas del sitio. Normalmente lo crea un administrador del sistema en el directorio `site-packages`. Si esta importación falla con un `ImportError` o su excepción de subclase, y el atributo `name` de la excepción es igual a `'sitecustomize'`, se ignora silenciosamente. Si Python se inicia sin flujos de salida disponibles, como con `pythonw.exe` en Windows (que se usa de forma predeterminada para iniciar IDLE), se ignora el intento de salida de `sitecustomize`. Cualquier otra excepción provoca una falla silenciosa y quizás misteriosa del proceso.

Después de esto, se intenta importar un módulo llamado `usercustomize`, que puede realizar personalizaciones arbitrarias específicas del usuario, si `ENABLE_USER_SITE` es verdadero. Este archivo está destinado a ser creado en el directorio `site-packages` del usuario (ver más abajo), que es parte de `sys.path` a menos que esté desactivado por `-s`. Si esta importación falla con una excepción `ImportError` o su subclase, y el atributo `name` de la excepción es igual a `'usercustomize'`, se ignora silenciosamente.

Tenga en cuenta que para algunos sistemas que no son Unix, `sys.prefix` y `sys.exec_prefix` están vacíos y se omiten las manipulaciones de la ruta; sin embargo, se sigue intentando importar `sitecustomize` y `usercustomize`.

29.14.1 Configuración de *Readline*

En los sistemas que admiten *readline*, este módulo también importará y configurará el módulo `rlcompleter`, si Python se inicia en modo interactivo y sin la opción `-S`. El comportamiento predeterminado es habilitar la finalización de tabulación y usar `~/.python_history` como archivo de guardado del historial. Para deshabilitarlo, borre (o anule) el atributo `sys.__interactivehook__` en su `sitecustomize` o `usercustomize` module o su archivo `PYTHONSTARTUP`.

Distinto en la versión 3.4: La activación de `rlcompleter` y el historial se hizo automática.

29.14.2 Contenido del módulo

`site.PREFIXES`

Una lista de prefijos para directorios de `site-packages`.

`site.ENABLE_USER_SITE`

Flag que muestra el estado del directorio `site-packages` del usuario. `True` significa que está habilitado y se agregó a `sys.path`. `False` significa que fue deshabilitado por solicitud del usuario (con `-s` o `PYTHONNOUSERSITE`). `None` significa que fue deshabilitado por razones de seguridad (falta de coincidencia entre la identificación de usuario o grupo y la identificación efectiva) o por un administrador.

`site.USER_SITE`

Ruta a los paquetes de sitio del usuario para Python en ejecución. Puede ser `None` si aún no se ha llamado a `getusersitepackages()`. El valor predeterminado es `~/.local/lib/pythonX.Y/site-packages` para compilaciones de Mac OS X sin marco y UNIX, `~/Library/Python/X.Y/lib/python/site-packages` para compilaciones de framework de Mac y `%APPDATA%\Python\PythonXY\site-packages` en Windows. Este directorio es un directorio de sitio, lo que significa que `.pth` se procesarán los archivos que contiene.

`site.USER_BASE`

Ruta al directorio base para los paquetes de sitio del usuario. Puede ser `None` si aún no se ha llamado a `getuserbase()`. El valor predeterminado es `~/.local` para las compilaciones que no son de marco de UNIX y Mac OS X, `~/Library/Python/X.Y` para las compilaciones de framework de Mac y `%APPDATA%\Python` para Windows. Distutils utiliza este valor para calcular los directorios de instalación de scripts, archivos de datos, módulos de Python, etc. para el esquema de instalación del usuario. Vea también `PYTHONUSERBASE`.

`site.main()`

Agrega todos los directorios estándar específicos del sitio a la ruta de búsqueda del módulo. Esta función se llama automáticamente cuando se importa este módulo, a menos que el intérprete de Python se haya iniciado con la opción `-S`.

Distinto en la versión 3.3: Esta función solía llamarse incondicionalmente.

`site.addsitedir(sitedir, known_paths=None)`

Agrega un directorio a `sys.path` y procesa sus archivos `.pth`. Típicamente utilizado en `sitecustomize` o `usercustomize` (ver arriba).

`site.getsitepackages()`

Retorna una lista que contiene todos los directorios *site-packages* globales.

Nuevo en la versión 3.2.

`site.getuserbase()`

Retorna la ruta del directorio base del usuario `USER_BASE`. Si este aún no fue inicializado, esta función también lo configurará, respetando `PYTHONUSERBASE`.

Nuevo en la versión 3.2.

`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 `USER_BASE`. To determine if the user-specific `site-packages` was added to `sys.path` `ENABLE_USER_SITE` should be used.

Nuevo en la versión 3.2.

29.14.3 Interfaz de línea de comando

El módulo *site* también proporciona una forma de obtener los directorios del usuario desde la línea de comandos:

```
$ python3 -m site --user-site
/home/user/.local/lib/python3.3/site-packages
```

Si se llama sin argumentos, imprimirá el contenido de *sys.path* en la salida estándar, seguido del valor de *USER_BASE* y si el directorio existe, entonces lo mismo para *USER_SITE*, y finalmente el valor de *ENABLE_USER_SITE*.

--user-base

Imprime la ruta al directorio base del usuario.

--user-site

Imprime la ruta al directorio *site-packages* del usuario.

Si se dan ambas opciones, la ruta del directorio base y la del directorio *site-packages* del usuario se imprimirán (siempre en este orden), separados por *os.pathsep*.

Si se da alguna opción, el script saldrá con uno de estos valores: “0” si el directorio *site-packages* del usuario está habilitado, “1” si fue deshabilitado por el usuario, “2” si está deshabilitado por razones de seguridad o por un administrador, y un valor mayor que 2 si hay un error.

Ver también:

PEP 370 - Directorio *site-packages* de cada usuario

Intérpretes de Python personalizados

Los módulos descritos en este capítulo permiten escribir interfaces similares al intérprete interactivo de Python. Si desea un intérprete de Python que admita alguna característica especial además del lenguaje Python, debe mirar el módulo `code`. (El módulo `codeop` es de más bajo nivel y se utiliza para soportar la compilación de un fragmento posiblemente incompleto de código Python).

La lista completa de módulos descritos en este capítulo es:

30.1 `code` — Clases básicas de intérpretes

Código fuente:: [Lib/code.py](#)

El módulo de `code` proporciona facilidades para implementar bucles de lectura-evaluación-impresión en Python. Se incluyen dos clases y funciones de conveniencia que se pueden usar para crear aplicaciones que brinden un *prompt* interactivo del intérprete.

class `code.InteractiveInterpreter` (*locals=None*)

Esta clase se ocupa del estado del intérprete y del análisis sintáctico (el espacio de nombres del usuario); no se ocupa del almacenamiento en búfer de entrada o de la solicitud o la denominación de archivos de entrada (el nombre de archivo siempre se pasa explícitamente). El argumento opcional *locals* especifica el diccionario en el que se ejecutará el código; por defecto es un diccionario recién creado con la clave `'__name__'` establecida en `'__console__'` y la clave `'__doc__'` en `None`.

class `code.InteractiveConsole` (*locals=None, filename="<console>"*)

Emula estrictamente el comportamiento del intérprete interactivo de Python. Esta clase se basa en `InteractiveInterpreter` y agrega solicitudes y búfer de entrada usando los conocidos `sys.ps1` y `sys.ps2`.

`code.interact` (*banner=None, readfunc=None, local=None, exitmsg=None*)

Función de conveniencia para ejecutar un ciclo de lectura-evaluación-impresión (*read-eval-print*). Esto crea una nueva instancia de `InteractiveConsole` y establece *readfunc* -si se proporciona- para ser utilizado como el método `InteractiveConsole.raw_input()`. Si se proporciona *local*, se pasa al constructor `InteractiveConsole` para usarlo como el espacio de nombres predeterminado para el bucle del intérprete. El método `interact()` de la instancia se ejecuta con *banner* y *exitmsg*, estos se pasan como bandera y mensaje de salida para usar nuevamente, si se proporciona. El objeto de la consola se descarta después de su uso.

Distinto en la versión 3.6: Se agregó el parámetro *exitmsg*.

`code.compile_command(source, filename=<input>, symbol="single")`

Esta función es útil para programas que desean emular el bucle principal del intérprete de Python (también conocido como bucle *read-eval-print*). La parte complicada es determinar cuándo el usuario ha ingresado un comando incompleto que se puede completar ingresando más texto (en lugar de un comando completo o un error de sintaxis). Esta función *casi* siempre toma la misma decisión que el bucle principal del intérprete real.

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 'single' (the default), 'eval' or 'exec'.

Retorna un objeto de código (lo mismo que `compile(source, filename, symbol)`) si el comando está completo y es válido; `None` si el comando está incompleto; lanza *SyntaxError* si el comando está completo y contiene un error de sintaxis, o lanza *OverflowError* o *ValueError* si el comando contiene un literal no válido.

30.1.1 Objetos de intérprete interactivo

`InteractiveInterpreter.runsource(source, filename=<input>, symbol="single")`

Compila y ejecuta alguna fuente en el intérprete. Los argumentos son los mismos que para `compile_command()`; el valor predeterminado para *filename* es '<input>', y para *symbol* es 'single'. Puede suceder una de varias cosas:

- La entrada es incorrecta; `compile_command()` generó una excepción (*SyntaxError* o *OverflowError*). Se imprime un seguimiento de sintaxis llamando al método `showsyntaxerror()`. `runsource()` retorna `False`.
- La entrada está incompleta y se requiere más información; `compile_command()` retorna `None`. `runsource()` retorna `True`.
- La entrada está completa; `compile_command()` retornó un objeto de código. El código se ejecuta llamando a `runcode()` (que también maneja excepciones en tiempo de ejecución, excepto *SystemExit*). `runsource()` retorna `False`.

El valor de retorno se puede usar para decidir si usar `sys.ps1` o `sys.ps2` para solicitar la siguiente línea.

`InteractiveInterpreter.runcode(code)`

Ejecuta un objeto de código. Cuando ocurre una excepción, se llama a `showtraceback()` para mostrar un seguimiento del código. Se capturan todas las excepciones excepto *SystemExit*, que puede propagarse.

Una nota sobre *KeyboardInterrupt*: esta excepción puede ocurrir en otra parte de este código y no siempre se detecta. Quien llama la función debe estar preparado para manejar esto.

`InteractiveInterpreter.showsyntaxerror(filename=None)`

Muestra el error de sintaxis que acaba de ocurrir. Esto no muestra la traza de seguimiento porque no la hay para errores de sintaxis. Si se proporciona *filename*, se incluirá en la excepción en lugar del nombre de archivo predeterminado proporcionado por el analizador de Python, ya que siempre usa '<string>' cuando lee de una cadena de caracteres. La salida se escribe mediante el método `write()`.

`InteractiveInterpreter.showtraceback()`

Muestra la excepción que acaba de ocurrir. Eliminamos el primer elemento de la pila porque está dentro de la implementación del intérprete. La salida se escribe mediante el método `write()`.

Distinto en la versión 3.5: Se muestra la traza de seguimiento encadenada completa en lugar de solo la traza primaria de pila.

`InteractiveInterpreter.write(data)`

Escribe una cadena de caracteres en el flujo de error estándar (`sys.stderr`). Las clases derivadas deben reemplazar esto para proporcionar el manejo de salida apropiado según sea necesario.

30.1.2 Objetos de consola interactiva

La clase `InteractiveConsole` es una subclase de `InteractiveInterpreter`, por lo que ofrece todos los métodos de los objetos del intérprete, así como las siguientes adiciones.

`InteractiveConsole.interact (banner=None, exitmsg=None)`

Emula estrictamente la consola interactiva de Python. El argumento opcional *banner* especifica la bandera a imprimir antes de la primera interacción; de forma predeterminada, imprime una bandera similar al impreso por el intérprete estándar de Python, seguido del nombre de clase del objeto de la consola entre paréntesis (para no confundir esto con el intérprete real, ¡ya que está muy cerca!)

El argumento opcional *exitmsg* especifica un mensaje de salida que se imprime al salir. Pase la cadena de caracteres vacía para suprimir el mensaje de salida. Si no se proporciona *exitmsg* o es `None`, se imprime el mensaje predeterminado.

Distinto en la versión 3.4: Para suprimir la impresión de cualquier bandera, pase una cadena de caracteres vacía.

Distinto en la versión 3.6: Imprime un mensaje de salida al salir.

`InteractiveConsole.push (line)`

Envía una línea de texto fuente al intérprete. La línea no debe tener un salto de línea al final; puede tener nuevas líneas internas. La línea se agrega a un búfer y se llama al método `runsource()` del intérprete con el contenido concatenado del búfer y la nueva fuente. Si indica que el comando se ejecutó o no es válido, el búfer se restablece; de lo contrario, el comando está incompleto y el búfer se deja como estaba después de agregar la línea. Si se requieren más entradas el valor de retorno es `True`, `False` si la línea se procesó de alguna manera (esto es lo mismo que `runsource()`).

`InteractiveConsole.resetbuffer ()`

Elimina cualquier texto fuente no gestionado del búfer de entrada.

`InteractiveConsole.raw_input (prompt="")`

Escribe un *prompt* y lee una línea. La línea retornada no incluye el salto de línea final. Cuando el usuario ingresa la secuencia de teclas EOF, se lanza `EOFError`. La implementación base lee de `sys.stdin`; una subclase puede reemplazar esto con una implementación diferente.

30.2 codeop — Compile Python code

Source code: `Lib/codeop.py`

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="<input>", symbol="single")`

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), as a sequence of statements ('exec') or as an *expression* ('eval'). Any other value will cause *ValueError* to be raised.

Nota: 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.

Importando módulos

Los módulos descritos en este capítulo proporcionan nuevas formas de importar otros módulos de Python y ganchos para personalizar los procesos de importación.

La lista completa de módulos descritos en este capítulo es:

31.1 `zipimport` — Importar módulos desde archivos zip

Código fuente: [Lib/zipimport.py](#)

Este módulo añade la capacidad de importar módulos de Python (`*.py`, `*.pyc`) y paquetes de archivos de formato ZIP. Por lo general, no es necesario utilizar el módulo `zipimport` explícitamente; se utiliza automáticamente por el mecanismo incorporado `import` para los ítems `sys.path` que son rutas a archivos ZIP.

Típicamente `sys.path` es una lista de cadenas con nombres de directorios. Este módulo también permite a un elemento de `sys.path` ser una cadena con la que se nombre a un archivo ZIP. El archivo ZIP puede contener una estructura de subdirectorios para soportar la importación de paquetes, y una ruta dentro del archivo puede ser especificada para únicamente importar desde un subdirectorio. Por ejemplo, la ruta `example.zip/lib/` sólo importaría desde el subdirectorio `lib/` dentro del archivo.

Cualquier archivo puede estar presente en el archivo ZIP, pero únicamente los archivos `.py` y `.pyc` están disponibles para importar. La importación ZIP de módulos dinámicos (`.pyd`, `.so`) no está permitida. Cabe señalar que si un archivo ZIP contiene solamente archivos `.py`, Python no intentará modificar el archivo agregando los correspondientes archivos `.pyc`, esto quiere decir que si un archivo ZIP no contiene archivos `.pyc` la importación puede ser algo lenta.

Distinto en la versión 3.8: Anteriormente, los archivos ZIP con un comentario de archivo no eran compatibles.

Ver también:

PKZIP Nota de aplicación Documentación sobre el formato de archivo ZIP por Phil Katz, el creador del formato y algoritmos utilizados.

PEP 273 - Importar módulos de archivos Zip Escrito por James C. Ahlstrom, quien también proporcionó una implementación. Python 2.3 sigue la especificación en PEP 273, pero utiliza una implementación escrita por Just van Rossum que utiliza los ganchos importados descritos en PEP 302.

PEP 302 - Nuevos ganchos de importación El PEP para agregar los ganchos de importación que ayudan a este módulo a funcionar.

Este módulo define una excepción:

exception `zipimport.ZipImportError`

Excepción lanzada por objetos `zipimporter`. Es una subclase de `ImportError`, por lo que también puede ser capturada como `ImportError`.

31.1.1 Objetos `zipimporter`

`zipimporter` es la clase para importar archivos ZIP.

class `zipimport.zipimporter` (*archivepath*)

Crea una nueva instancia `zipimporter`. *archivepath* debe ser una ruta a un archivo ZIP, o a una ruta específica dentro de un archivo ZIP. Por ejemplo, un *archivepath* de `foo/bar.zip/lib` buscará módulos en el directorio `lib` dentro del archivo ZIP `foo/bar.zip` (siempre que exista).

`ZipImportError` es lanzada si *archivepath* no apunta a un archivo ZIP válido.

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*)

Retorna el objeto de código para el módulo especificado. Lanza `ZipImportError` si el módulo no pudo ser encontrado.

get_data (*pathname*)

Retorna los datos asociados con *pathname*. Lanza `OSError` si el archivo no fue encontrado.

Distinto en la versión 3.3: `IOError` solía lanzarse en lugar de `OSError`.

get_filename (*fullname*)

Retorna el valor que se le habría asignado a `__file__` si el módulo especificado fue importado. Lanza `ZipImportError` si el módulo no pudo ser encontrado.

Nuevo en la versión 3.1.

get_source (*fullname*)

Retorna el código fuente para el módulo especificado. Lanza `ZipImportError` si el módulo no pudo ser encontrado, retorna `None` si el archivo no contiene al módulo, pero no tiene fuente para ello.

is_package (*fullname*)

Retorna `True` si el módulo especificado por *fullname* es un paquete. Lanza `ZipImportError` si el módulo no pudo ser encontrado.

load_module (*fullname*)

Cargue el módulo especificado por *fullname*. *fullname* debe ser el nombre completo de módulo (punteado). Retorna el módulo importado, o lanza `ZipImportError` si no fue encontrado.

archive

El nombre de archivo del archivo ZIP asociado del importador, sin una posible sub-ruta.

prefix

La sub-ruta dentro del archivo ZIP donde se buscan los módulos. Esta es la cadena vacía para objetos `zipimporter` la cual apunta a la raíz del archivo ZIP.

Los atributos `archive` y `prefix`, cuando son combinados con una barra diagonal, son iguales al argumento original *archivepath* dado al constructor `zipimporter`.

31.1.2 Ejemplos

Este es un ejemplo que importa un módulo de un archivo ZIP - tenga en cuenta que el módulo `zipimport` no está usado explícitamente.

```
$ 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'
```

31.2 pkgutil — Utilidad de extensión de paquete

Código fuente: [Lib/pkgutil.py](#)

Este módulo proporciona utilidades para el sistema de importación, en particular soporte para paquetes.

class `pkgutil.ModuleInfo` (*module_finder, name, ispkg*)

Una tupla nombrada que contiene un breve resumen de la información del módulo.

Nuevo en la versión 3.6.

`pkgutil.extend_path` (*path, name*)

Extiende la ruta de búsqueda de los módulos que componen un paquete. El uso previsto es colocar el siguiente código en `__init__.py` de un paquete:

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

Esto agregará al `__path__` del paquete todos los subdirectorios de directorios en `sys.path` con el nombre del paquete. Esto es útil si se desea distribuir diferentes partes de un único paquete lógico como varios directorios.

También busca archivos `*.pkg` donde `*` coincide con el argumento *name*. Esta característica es similar a archivos `*.pth` (vea el módulo *site* para más información) excepto que no incluye líneas de casos especiales que comienzan con `import`. Un archivo `*.pkg` es confiable al pie de la letra: además de buscar duplicados, todas las entradas encontradas en un `*.pkg` se agregan a la ruta, independientemente de si existen en el sistema de archivos. (Esto es una funcionalidad).

Si la ruta de entrada no es una lista (como es el caso de los paquetes congelados), se retorna sin cambios. La ruta de entrada no se modifica; se retorna una copia ampliada. Los elementos solo se adjuntan a la copia al final.

Se supone que `sys.path` es una secuencia. Los elementos de `sys.path` que no sean cadenas de caracteres que se refieran a directorios existentes se ignoran. Los elementos Unicode en `sys.path` que causan errores cuando se utilizan como nombres de archivo pueden hacer que esta función lance una excepción (en línea con el comportamiento de `os.path.isdir()`).

class `pkgutil.ImpImporter` (*dirname=None*)

Buscador **PEP 302** que envuelve el algoritmo de importación «clásico» de Python.

Si *dirname* es una cadena de caracteres, se crea un buscador **PEP 302** que busca ese directorio. Si *dirname* es `None`, se crea un buscador **PEP 302** que busca en el actual `sys.path`, más cualquier módulo que esté congelado o incorporado.

Tenga en cuenta que `ImpImporter` no admite actualmente el uso de ubicación `sys.meta_path`.

Obsoleto desde la versión 3.3: Esta emulación ya no es necesaria, ya que ahora lo es el mecanismo de importación estándar totalmente compatible con **PEP 302** y disponible en `importlib`.

class `pkgutil.ImpLoader` (*fullname, file, filename, etc*)
Loader that wraps Python's «classic» import algorithm.

Obsoleto desde la versión 3.3: Esta emulación ya no es necesaria, ya que ahora lo es el mecanismo de importación estándar totalmente compatible con **PEP 302** y disponible en `importlib`.

`pkgutil.find_loader` (*fullname*)

Recupera un módulo *loader* para un *fullname* dado.

Este es un contenedor de compatibilidad con versiones anteriores de `importlib.util.find_spec()` que convierte la mayoría de los errores en `ImportError` y solo retorna el cargador en lugar del completo `ModuleSpec`.

Distinto en la versión 3.3: Actualizado para basarse directamente en `importlib` en lugar de depender del paquete interno **PEP 302** emulación de importación.

Distinto en la versión 3.4: Actualizado basado en **PEP 451**

`pkgutil.get_importer` (*path_item*)

Recupera un *finder* para el *path_item*.

El buscador retornado se almacena en caché en `sys.path_importer_cache` si fue creado recientemente por un enlace de ruta.

La caché (o parte de ella) puede ser borrada manualmente si el escaneo de `sys.path_hooks` es necesario.

Distinto en la versión 3.3: Actualizado para basarse directamente en `importlib` en lugar de depender del paquete interno **PEP 302** emulación de importación.

`pkgutil.get_loader` (*module_or_name*)

Obtiene un objeto *loader* para *module_or_name*.

Si se puede acceder al módulo o paquete a través del mecanismo de importación normal, se devuelve un contenedor alrededor de la parte relevante de esa maquinaria. Retorna `None` si el módulo no se puede encontrar o importar. Si el módulo nombrado aún no se ha importado, se importa el paquete que lo contiene (si lo hay), para establecer el paquete `__path__`.

Distinto en la versión 3.3: Actualizado para basarse directamente en `importlib` en lugar de depender del paquete interno **PEP 302** emulación de importación.

Distinto en la versión 3.4: Actualizado basado en **PEP 451**

`pkgutil.iter_importers` (*fullname=""*)

Cede (*yield*) objetos *finder* para el nombre de módulo dado.

Si el nombre completo contiene un `"."`, los buscadores serán para el paquete que contiene el nombre completo; de lo contrario, serán todos los buscadores de nivel superior registrados (es decir, los de `sys.meta_path` y `sys.path_hooks`).

Si el módulo nombrado está en un paquete, ese paquete se importa como un efecto secundario de invocar esta función.

Si no se especifica ningún nombre de módulo, se generan todos los buscadores de nivel superior.

Distinto en la versión 3.3: Actualizado para basarse directamente en `importlib` en lugar de depender del paquete interno **PEP 302** emulación de importación.

`pkgutil.iter_modules` (*path=None, prefix=""*)

Cede (*yield*) `ModuleInfo` para todos los submódulos en *path*, o, si *path* es `None`, todos los módulos de nivel superior en `sys.path`.

path tendría que ser `None` o una list de rutas en las que buscar módulos.

prefix es una cadena para mostrar delante de cada nombre de módulo en la salida.

Nota: Sólo funciona para un *finder* que define un método `iter_modules()`. Esta interfaz no es estándar, por lo que el módulo también proporciona implementaciones para `importlib.machinery.FileFinder` y `zipimport.zipimporter`.

Distinto en la versión 3.3: Actualizado para basarse directamente en `importlib` en lugar de depender del paquete interno **PEP 302** emulación de importación.

`pkgutil.walk_packages` (*path=None*, *prefix=""*, *onerror=None*)

Cede (yield) `ModuleInfo` para todos los módulos de forma recursiva en *path*, o, si *path* es `None`, todos los módulos accesibles.

path tendría que ser `None` o una list de rutas en las que buscar módulos.

prefix es una cadena para mostrar delante de cada nombre de módulo en la salida.

Note que esta función debe importar todos los *packages* (¡no todos los módulos!) en el *path* especificado, para acceder al atributo `__path__` para encontrar submódulos.

onerror es una función que se llama con un argumento (el nombre del paquete que se estaba importando) si se produce alguna excepción al intentar importar un paquete. Si no se proporciona ninguna función *onerror*, los `ImportError` se capturan e ignoran, mientras que todas las demás excepciones se propagan, terminando la búsqueda.

Ejemplos:

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')

```

Nota: Sólo funciona para un *finder* que define un método `iter_modules()`. Esta interfaz no es estándar, por lo que el módulo también proporciona implementaciones para `importlib.machinery.FileFinder` y `zipimport.zipimporter`.

Distinto en la versión 3.3: Actualizado para basarse directamente en `importlib` en lugar de depender del paquete interno **PEP 302** emulación de importación.

`pkgutil.get_data` (*package*, *resource*)

Obtiene un recurso de un paquete.

Esto es un contenedor para la API *loader* `get_data`. El argumento *package* debe ser el nombre de un paquete, en formato de módulo estándar (`foo.bar`). El argumento *resource* debe tener la forma de un nombre de archivo relativo, utilizando `/` como separador de ruta. El nombre del directorio principal `..` no está permitido, ni tampoco un nombre raíz (empezando por `/`).

La función retorna una cadena de caracteres binaria que es el contenido del recurso especificado.

Para los paquetes ubicados en el sistema de archivos, que ya se han importado, este es el equivalente aproximado de:

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()

```

Si el paquete no puede ser localizado o cargado, o usa un *loader* el cuál no soporta `get_data`, entonces retorna `None`. En particular, el término *loader* para *namespace packages* no soporta `get_data`.

31.3 modulefinder — Buscar módulos usados por un script

Código fuente: [Lib/modulefinder.py](#)

Este módulo provee una clase `ModuleFinder` que puede ser usada para determinar el conjunto de módulos importados en un script. `modulefinder.py` puede también ejecutarse como un script, dando el nombre de un archivo de Python como argumento, tras lo cual se imprimirá un reporte de los módulos importados.

`modulefinder.AddPackagePath(pkg_name, path)`

Registra que el paquete llamado `pkg_name` pueda ser encontrado en el `path` especificado.

`modulefinder.ReplacePackage(oldname, newname)`

Permite especificar que el módulo llamado `oldname` es de hecho el paquete llamado `newname`.

class `modulefinder.ModuleFinder` (`path=None`, `debug=0`, `excludes=[]`, `replace_paths=[]`)

Esta clase provee los métodos `run_script()` y `report()` que determinan el conjunto de módulos importados por un script. `path` puede ser un listado de directorios a buscar por módulos; si no es especificado, se usará `sys.path`. `debug` define el nivel de depuración; valores más altos hacen que la clase imprima mensajes de depuración acerca de lo que está haciendo. `excludes` es una lista de nombres de módulos que serán excluidos del análisis. `replace_paths` es una lista de tuplas (`oldpath`, `newpath`) que serán remplazadas en las rutas de los módulos.

report()

Imprime un reporte a la salida estándar que lista los módulos importados por el script y sus rutas, así como los módulos que faltan o parecieran faltar.

run_script(pathname)

Analiza los contenidos del archivo `pathname`, que debe contener código Python.

modules

Un diccionario que mapea los nombres de los módulos a los módulos. Vea *Ejemplo de uso de ModuleFinder*.

31.3.1 Ejemplo de uso de ModuleFinder

El script que será analizado más adelante (`bacon.py`):

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

El script que generará el reporte de `bacon.py`:

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
```

(continué en la próxima página)

(proviene de la página anterior)

```
print(', '.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('Modules not imported:')
print('\n'.join(finder.badmodules.keys()))
```

Resultado de ejemplo (puede variar dependiendo de la arquitectura):

```
Loaded modules:
_types:
copyreg: _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: _PATTERNENDERS, SRE_FLAG_UNICODE
array:
types: __module__, IntType, TypeType
-----
Modules not imported:
guido.python.ham
baconhameggs
```

31.4 runpy — Localización y ejecución de módulos Python

Código Fuente: `Lib/runpy.py`

El módulo `runpy` es usado para localizar y correr módulos *Python* sin importarlo primero. Su uso principal es implementar la opción `-m` cambiando la línea de comando que permite que los scripts se ubiquen utilizando el espacio de nombres del módulo de Python en lugar del sistema de archivos.

Tenga en cuenta que este *no* es un módulo de espacio aislado - Todo el código es ejecutado en el proceso actual, y cualquier efecto secundario (como las importaciones en cache de otros módulos) permanecerán en su lugar después de que las funciones hayan retornado.

Además, no se garantiza que las funciones y clases definidas por el código ejecutado funcionen correctamente después de que se haya devuelto la función `runpy`. Si esa limitación no es aceptable para un caso de uso determinado, es probable que `importlib` sea una opción más adecuada que este módulo.

El módulo `runpy` proporciona dos funciones:

`runpy.run_module(mod_name, init_globals=None, run_name=None, alter_sys=False)`

Ejecute el código del módulo especificado y devuelva el diccionario de globales de módulo resultante. El código del módulo se encuentra primero mediante el mecanismo de importación estándar (consulte :p ep:"302" para obtener más información) y, a continuación, se ejecuta en un espacio de nombres de módulo nuevo.

El argumento `mod_name` debe ser un nombre de módulo absoluto. Si el nombre del paquete se refiere a un paquete en lugar de un módulo normal, entonces ese paquete es importado y el submódulo `__main__` dentro de ese paquete luego se ejecuta y se devuelve el diccionario global del módulo resultante.

El argumento de diccionario opcional `init_globals` se puede utilizar para rellenar previamente el diccionario global del módulo antes de ejecutar el código. El diccionario suministrado no se modificará. Si alguna de las variables globales especiales siguientes se define en el diccionario proporcionado, esas definiciones se reemplazan por `run_module()`.

Las variables globales especiales `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` y `__package__` se establecen en el diccionario global antes del que el código del módulo sea ejecutado (tenga en cuenta que esto es un conjunto mínimo de variables - otras variables pueden establecerse implícitamente como un detalle de implementación del intérprete).

`__name__` se establece en `run_name` si el argumento opcional no es `None`, para `mod_name` + `'__main__'` si módulo nombrado es un paquete y al argumento `mod_name` en caso contrario.

`__spec__` se configura apropiadamente para el módulo *realmente* importado (es decir, `__spec__.name` siempre será un `mod_name` o `mod_name` + `'__main__'`, jamás `run_name`).

`__file__`, `__cached__`, `__loader__` y `__package__` son basados en la especificación del módulo set as normal.

Si el argumento `alter_sys` es proporcionado y evaluado a `True`, entonces `sys.argv[0]` es actualizado y el valor de `__file__` y `sys.modules[__name__]` es actualizado con un objeto de módulo temporal para el módulo que se está ejecutando. Ambas `sys.argv[0]` y `sys.modules[__name__]` son restauradas a sus valores originales antes del retorno de la función.

Tenga en cuenta que esta manipulación de `sys` no es segura para subprocessos. Otros subprocessos pueden ver el módulo parcialmente inicializado, así como la lista alterada de argumentos. Se recomienda que el módulo `sys` se deje solo al invocar esta función desde código roscado.

Ver también:

La opción `-m` ofrece una funcionalidad equivalente desde la línea de comandos.

Distinto en la versión 3.1: Se agregó la capacidad de ejecutar paquetes buscando un submódulo `__main__`.

Distinto en la versión 3.2: Se agregó la variable global `__cached__` (consultar [PEP 3147](#)).

Distinto en la versión 3.4: Se ha actualizado para aprovechar la función de especificación de módulo agregada por `:pep:451`. Esto permite que `__cached__` se establezca correctamente para que los módulos se ejecuten de esta manera, así como asegurarse de que el nombre real del módulo siempre sea accesible como `__spec__.name`.

`runpy.run_path(file_path, init_globals=None, run_name=None)`

Ejecute el código en la ubicación del sistema de archivos con nombre y devuelva el diccionario de globales de módulo resultante. Al igual que con un nombre de script proporcionado a la línea de comandos de CPython, la ruta de acceso proporcionada puede hacer referencia a un archivo de origen de Python, un archivo de código de bytes compilado o una entrada `sys.path` válida que contiene un módulo `__main__` (por ejemplo, un archivo zip que contiene un archivo `__main__.py` de nivel superior).

Para un *script* simple, el código especificado se ejecuta simplemente en un espacio de nombres de un módulo nuevo. Para un entrada `sys.path` válida (comúnmente es un archivo *zip* o un directorio), la entrada se agrega primero al comienzo de `sys.path`. La función busca y ejecuta un módulo `__main__` usando la ruta actualizada. Tenga en cuenta que no existe una protección especial contra la invocación de una entrada existente `__main__` ubicada en otro lugar en `sys.path` si no hay tal módulo en la ubicación especificada.

El argumento de diccionario opcional `init_globals` se puede utilizar para rellenar previamente el diccionario global del módulo antes de ejecutar el código. El diccionario suministrado no se modificará. Si alguna de las variables globales especiales siguientes se define en el diccionario proporcionado, esas definiciones se reemplazan por `run_path()`.

Las variables globales especiales `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` y `__package__` se establecen en el diccionario global antes del que el código del módulo sea ejecutado (tenga en cuenta que esto es un conjunto mínimo de variables - otras variables pueden establecerse implícitamente como un detalle de implementación del intérprete).

`__name__` se establece para `run_name` si el argumento opcional no es `None` y a `'<run_path>'` de lo contrario.

Si la ruta proporcionada hace referencia a un archivo *script* (ya sea como fuente o un código de *byte* pre-compilado), entonces `__file__` se establecerá en la ruta proporcionada, y `__spec__`, `__cached__`, `__loader__` y `__package__` se establecerán todos en `None`.

Si la ruta proporcionada es una referencia a una entrada *sys.path* válida, entonces `__spec__` se establece apropiadamente para la importación del módulo `__main__` (es decir, `__spec__.name` siempre deberá ser `__main__`). `__file__`, `__cached__`, `__loader__` y `__package__` estarán basadas en la especificación del módulo establecidas como normal.

También se realizan una serie de alteraciones en el módulo *sys*. En primer lugar, `sys.path` puede ser alterado como se describió anteriormente. `sys.argv[0]` se actualiza con el valor de `file_path` y `sys.modules[__name__]` se actualiza con un objeto de módulo temporal para el módulo que se está ejecutando. Todas las modificaciones de los elementos de *sys* se revierten antes de que se devuelva la función.

Tenga en cuenta que, diferente a *run_module()*, las alteraciones hechas a *sys* no son opcionales en esta función ya que estos ajustes son esenciales para permitir la ejecución de entradas *sys.path*. Como aún se aplican las limitaciones de seguridad de los subprocesos, el uso de esta función en un código procesado debe serializarse con el bloqueo de importación o delegarse a un proceso separado.

Ver también:

using-on-interface-options para una funcionalidad equivalente en la línea de comandos (`python path/to/script`).

Nuevo en la versión 3.2.

Distinto en la versión 3.4: Actualizado para aprovechar la función de especificación del módulo agregada por **PEP 451**. Esto permite que `__cached__` se configure correctamente en el caso de que `__main__` se importe de una entrada *sys.path* válida en lugar de ejecutarse directamente.

Ver también:

PEP 338 – Ejecutando módulos como *scripts* *PEP* escrito y implementado por Nick Coghlan.

PEP 366 – Importaciones relativas explícitas del módulo principal *PEP* escrito y implementado por Nick Coghlan.

PEP 451 — Un tipo *ModuleSpec* para el sistema de Importación *PEP* escrito y implementado por Eric Snow

using-on-general - Detalles de la línea de comandos *CPython*

La función `importlib.import_module()`

31.5 importlib — The implementation of import

Nuevo en la versión 3.1.

Source code: `Lib/importlib/_init__.py`

31.5.1 Introduction

The purpose of the *importlib* package is two-fold. One is to provide the implementation of the `import` statement (and thus, by extension, the `__import__()` function) in Python source code. This provides an implementation of `import` which is portable to any Python interpreter. This also provides an implementation which is easier to comprehend than one implemented in a programming language other than Python.

Two, the components to implement `import` are exposed in this package, making it easier for users to create their own custom objects (known generically as an *importer*) to participate in the import process.

Ver también:

import The language reference for the `import` statement.

Packages specification Original specification of packages. Some semantics have changed since the writing of this document (e.g. redirecting based on `None` in *sys.modules*).

The `__import__()` function The `import` statement is syntactic sugar for this function.

PEP 235 Import on Case-Insensitive Platforms
PEP 263 Defining Python Source Code Encodings
PEP 302 New Import Hooks
PEP 328 Imports: Multi-Line and Absolute/Relative
PEP 366 Main module explicit relative imports
PEP 420 Implicit namespace packages
PEP 451 A ModuleSpec Type for the Import System
PEP 488 Elimination of PYO files
PEP 489 Multi-phase extension module initialization
PEP 552 Deterministic pycs
PEP 3120 Using UTF-8 as the Default Source Encoding
PEP 3147 PYC Repository Directories

31.5.2 Functions

`importlib.__import__(name, globals=None, locals=None, fromlist=(), level=0)`
An implementation of the built-in `__import__()` function.

Nota: Programmatic importing of modules should use `import_module()` instead of this function.

`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 set to the name of 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 `import_module()` function acts as a simplifying wrapper around `importlib.__import__()`. This means all semantics of the function are derived from `importlib.__import__()`. The most important difference between these two functions is that `import_module()` returns the specified package or module (e.g. `pkg.mod`), while `__import__()` returns the top-level package or module (e.g. `pkg`).

If you are dynamically importing a module that was created since the interpreter began execution (e.g., created a Python source file), you may need to call `invalidate_caches()` in order for the new module to be noticed by the import system.

Distinto en la versión 3.3: Parent packages are automatically imported.

`importlib.find_loader(name, path=None)`

Find the loader for a module, optionally within the specified *path*. If the module is in `sys.modules`, then `sys.modules[name].__loader__` is returned (unless the loader would be `None` or is not set, in which case `ValueError` is raised). Otherwise a search using `sys.meta_path` is done. `None` is returned if no loader is found.

A dotted name does not have its parents implicitly imported as that requires loading them and that may not be desired. To properly import a submodule you will need to import all parent packages of the submodule and use the correct argument to *path*.

Nuevo en la versión 3.3.

Distinto en la versión 3.4: If `__loader__` is not set, raise `ValueError`, just like when the attribute is set to `None`.

Obsoleto desde la versión 3.4: Use `importlib.util.find_spec()` instead.

`importlib.invalidate_caches()`

Invalidate the internal caches of finders stored at `sys.meta_path`. If a finder implements `invalidate_caches()` then it will be called to perform the invalidation. This function should be called if any modules are created/installed while your program is running to guarantee all finders will notice the new module's existence.

Nuevo en la versión 3.3.

`importlib.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 (which can be different if re-importing causes a different object to be placed in `sys.modules`).

When `reload()` is executed:

- Python module's code is recompiled and the module-level code re-executed, defining a new set of objects which are bound to names in the module's dictionary by reusing the *loader* which originally loaded the module. 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.

Nuevo en la versión 3.4.

Distinto en la versión 3.7: `ModuleNotFoundError` is raised when the module being reloaded lacks a `ModuleSpec`.

31.5.3 `importlib.abc` – Abstract base classes related to import

Source code: `Lib/importlib/abc.py`

The `importlib.abc` module contains all of the core abstract base classes used by `import`. Some subclasses of the core abstract base classes are also provided to help in implementing the core ABCs.

ABC hierarchy:

```
object
+-- Finder (deprecated)
|   +-- MetaPathFinder
|   +-- PathEntryFinder
+-- Loader
    +-- ResourceLoader -----+
    +-- InspectLoader         |
        +-- ExecutionLoader --+
                                +-- FileLoader
                                +-- SourceLoader
```

class `importlib.abc.Finder`

An abstract base class representing a *finder*.

Obsoleto desde la versión 3.3: Use `MetaPathFinder` or `PathEntryFinder` instead.

abstractmethod `find_module` (*fullname*, *path=None*)

An abstract method for finding a *loader* for the specified module. Originally specified in **PEP 302**, this method was meant for use in `sys.meta_path` and in the path-based import subsystem.

Distinto en la versión 3.4: Returns `None` when called instead of raising `NotImplementedError`.

class `importlib.abc.MetaPathFinder`

An abstract base class representing a *meta path finder*. For compatibility, this is a subclass of `Finder`.

Nuevo en la versión 3.3.

find_spec (*fullname*, *path*, *target=None*)

An abstract method for finding a *spec* for the specified module. If this is a top-level import, *path* will be `None`. Otherwise, this is a search for a subpackage or module and *path* will be the value of `__path__` from the parent package. If a spec cannot be found, `None` is returned. When passed in, *target* is a module object that the finder may use to make a more educated guess about what spec to return. `importlib.util.spec_from_loader()` may be useful for implementing concrete `MetaPathFinders`.

Nuevo en la versión 3.4.

find_module (*fullname*, *path*)

A legacy method for finding a *loader* for the specified module. If this is a top-level import, *path* will be `None`. Otherwise, this is a search for a subpackage or module and *path* will be the value of `__path__` from the parent package. If a loader cannot be found, `None` is returned.

If `find_spec()` is defined, backwards-compatible functionality is provided.

Distinto en la versión 3.4: Returns `None` when called instead of raising `NotImplementedError`. Can use `find_spec()` to provide functionality.

Obsoleto desde la versión 3.4: Use `find_spec()` instead.

invalidate_caches ()

An optional method which, when called, should invalidate any internal cache used by the finder. Used by `importlib.invalidate_caches()` when invalidating the caches of all finders on `sys.meta_path`.

Distinto en la versión 3.4: Returns `None` when called instead of `NotImplemented`.

class `importlib.abc.PathEntryFinder`

An abstract base class representing a *path entry finder*. Though it bears some similarities to *MetaPathFinder*, *PathEntryFinder* is meant for use only within the path-based import subsystem provided by *PathFinder*. This ABC is a subclass of *Finder* for compatibility reasons only.

Nuevo en la versión 3.3.

find_spec (*fullname*, *target=None*)

An abstract method for finding a *spec* for the specified module. The finder will search for the module only within the *path entry* to which it is assigned. If a spec cannot be found, *None* is returned. When passed in, *target* is a module object that the finder may use to make a more educated guess about what spec to return. `importlib.util.spec_from_loader()` may be useful for implementing concrete *PathEntryFinders*.

Nuevo en la versión 3.4.

find_loader (*fullname*)

A legacy method for finding a *loader* for the specified module. Returns a 2-tuple of (*loader*, *portion*) where *portion* is a sequence of file system locations contributing to part of a namespace package. The loader may be *None* while specifying *portion* to signify the contribution of the file system locations to a namespace package. An empty list can be used for *portion* to signify the loader is not part of a namespace package. If *loader* is *None* and *portion* is the empty list then no loader or location for a namespace package were found (i.e. failure to find anything for the module).

If *find_spec()* is defined then backwards-compatible functionality is provided.

Distinto en la versión 3.4: Returns (*None*, []) instead of raising *NotImplementedError*. Uses *find_spec()* when available to provide functionality.

Obsoleto desde la versión 3.4: Use *find_spec()* instead.

find_module (*fullname*)

A concrete implementation of *Finder.find_module()* which is equivalent to *self.find_loader(fullname)[0]*.

Obsoleto desde la versión 3.4: Use *find_spec()* instead.

invalidate_caches ()

An optional method which, when called, should invalidate any internal cache used by the finder. Used by *PathFinder.invalidate_caches()* when invalidating the caches of all cached finders.

class `importlib.abc.Loader`

An abstract base class for a *loader*. See [PEP 302](#) for the exact definition for a loader.

Loaders that wish to support resource reading should implement a *get_resource_reader(fullname)* method as specified by *importlib.abc.ResourceReader*.

Distinto en la versión 3.7: Introduced the optional *get_resource_reader()* method.

create_module (*spec*)

A method that returns the module object to use when importing a module. This method may return *None*, indicating that default module creation semantics should take place.

Nuevo en la versión 3.4.

Distinto en la versión 3.5: Starting in Python 3.6, this method will not be optional when *exec_module()* is defined.

exec_module (*module*)

An abstract method that executes the module in its own namespace when a module is imported or reloaded. The module should already be initialized when *exec_module()* is called. When this method exists, *create_module()* must be defined.

Nuevo en la versión 3.4.

Distinto en la versión 3.6: *create_module()* must also be defined.

load_module (*fullname*)

A legacy method for loading a module. If the module cannot be loaded, `ImportError` is raised, otherwise the loaded module is returned.

If the requested module already exists in `sys.modules`, that module should be used and reloaded. Otherwise the loader should create a new module and insert it into `sys.modules` before any loading begins, to prevent recursion from the import. If the loader inserted a module and the load fails, it must be removed by the loader from `sys.modules`; modules already in `sys.modules` before the loader began execution should be left alone (see `importlib.util.module_for_loader()`).

The loader should set several attributes on the module. (Note that some of these attributes can change when a module is reloaded):

- **__name__** The name of the module.
- **__file__** The path to where the module data is stored (not set for built-in modules).
- **__cached__** The path to where a compiled version of the module is/should be stored (not set when the attribute would be inappropriate).
- **__path__** A list of strings specifying the search path within a package. This attribute is not set on modules.
- **__package__** The fully-qualified name of the package under which the module was loaded as a submodule (or the empty string for top-level modules). For packages, it is the same as **__name__**. The `importlib.util.module_for_loader()` decorator can handle the details for **__package__**.
- **__loader__** The loader used to load the module. The `importlib.util.module_for_loader()` decorator can handle the details for **__package__**.

When `exec_module()` is available then backwards-compatible functionality is provided.

Distinto en la versión 3.4: Raise `ImportError` when called instead of `NotImplementedError`. Functionality provided when `exec_module()` is available.

Obsoleto desde la versión 3.4: The recommended API for loading a module is `exec_module()` (and `create_module()`). Loaders should implement it instead of `load_module()`. The import machinery takes care of all the other responsibilities of `load_module()` when `exec_module()` is implemented.

module_repr (*module*)

A legacy method which when implemented calculates and returns the given module's repr, as a string. The module type's default `repr()` will use the result of this method as appropriate.

Nuevo en la versión 3.3.

Distinto en la versión 3.4: Made optional instead of an abstractmethod.

Obsoleto desde la versión 3.4: The import machinery now takes care of this automatically.

class `importlib.abc.ResourceReader`

An *abstract base class* to provide the ability to read *resources*.

From the perspective of this ABC, a *resource* is a binary artifact that is shipped within a package. Typically this is something like a data file that lives next to the `__init__.py` file of the package. The purpose of this class is to help abstract out the accessing of such data files so that it does not matter if the package and its data file(s) are stored in a e.g. zip file versus on the file system.

For any of methods of this class, a *resource* argument is expected to be a *path-like object* which represents conceptually just a file name. This means that no subdirectory paths should be included in the *resource* argument. This is because the location of the package the reader is for, acts as the «directory». Hence the metaphor for directories and file names is packages and resources, respectively. This is also why instances of this class are expected to directly correlate to a specific package (instead of potentially representing multiple packages or a module).

Loaders that wish to support resource reading are expected to provide a method called `get_resource_reader(fullname)` which returns an object implementing this ABC's interfa-

ce. If the module specified by *fullname* is not a package, this method should return *None*. An object compatible with this ABC should only be returned when the specified module is a package.

Nuevo en la versión 3.7.

abstractmethod `open_resource(resource)`

Returns an opened, *file-like object* for binary reading of the *resource*.

If the resource cannot be found, *FileNotFoundError* is raised.

abstractmethod `resource_path(resource)`

Returns the file system path to the *resource*.

If the resource does not concretely exist on the file system, raise *FileNotFoundError*.

abstractmethod `is_resource(name)`

Returns *True* if the named *name* is considered a resource. *FileNotFoundError* is raised if *name* does not exist.

abstractmethod `contents()`

Returns an *iterable* of strings over the contents of the package. Do note that it is not required that all names returned by the iterator be actual resources, e.g. it is acceptable to return names for which *is_resource()* would be false.

Allowing non-resource names to be returned is to allow for situations where how a package and its resources are stored are known a priori and the non-resource names would be useful. For instance, returning subdirectory names is allowed so that when it is known that the package and resources are stored on the file system then those subdirectory names can be used directly.

The abstract method returns an iterable of no items.

class `importlib.abc.ResourceLoader`

An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loading arbitrary resources from the storage back-end.

Obsoleto desde la versión 3.7: This ABC is deprecated in favour of supporting resource loading through *importlib.abc.ResourceReader*.

abstractmethod `get_data(path)`

An abstract method to return the bytes for the data located at *path*. Loaders that have a file-like storage back-end that allows storing arbitrary data can implement this abstract method to give direct access to the data stored. *OSError* is to be raised if the *path* cannot be found. The *path* is expected to be constructed using a module's `__file__` attribute or an item from a package's `__path__`.

Distinto en la versión 3.4: Raises *OSError* instead of *NotImplementedError*.

class `importlib.abc.InspectLoader`

An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loaders that inspect modules.

get_code (*fullname*)

Return the code object for a module, or *None* if the module does not have a code object (as would be the case, for example, for a built-in module). Raise an *ImportError* if loader cannot find the requested module.

Nota: While the method has a default implementation, it is suggested that it be overridden if possible for performance.

Distinto en la versión 3.4: No longer abstract and a concrete implementation is provided.

abstractmethod `get_source(fullname)`

An abstract method to return the source of a module. It is returned as a text string using *universal newlines*, translating all recognized line separators into `'\n'` characters. Returns *None* if no source is available (e.g. a built-in module). Raises *ImportError* if the loader cannot find the module specified.

Distinto en la versión 3.4: Raises *ImportError* instead of *NotImplementedError*.

is_package (*fullname*)

An abstract method to return a true value if the module is a package, a false value otherwise. *ImportError* is raised if the *loader* cannot find the module.

Distinto en la versión 3.4: Raises *ImportError* instead of *NotImplementedError*.

static source_to_code (*data*, *path*='<string>')

Create a code object from Python source.

The *data* argument can be whatever the *compile()* function supports (i.e. string or bytes). The *path* argument should be the «path» to where the source code originated from, which can be an abstract concept (e.g. location in a zip file).

With the subsequent code object one can execute it in a module by running `exec(code, module.__dict__)`.

Nuevo en la versión 3.4.

Distinto en la versión 3.5: Made the method static.

exec_module (*module*)

Implementation of *Loader.exec_module()*.

Nuevo en la versión 3.4.

load_module (*fullname*)

Implementation of *Loader.load_module()*.

Obsoleto desde la versión 3.4: use *exec_module()* instead.

class `importlib.abc.ExecutionLoader`

An abstract base class which inherits from *InspectLoader* that, when implemented, helps a module to be executed as a script. The ABC represents an optional **PEP 302** protocol.

abstractmethod get_filename (*fullname*)

An abstract method that is to return the value of `__file__` for the specified module. If no path is available, *ImportError* is raised.

If source code is available, then the method should return the path to the source file, regardless of whether a bytecode was used to load the module.

Distinto en la versión 3.4: Raises *ImportError* instead of *NotImplementedError*.

class `importlib.abc.FileLoader` (*fullname*, *path*)

An abstract base class which inherits from *ResourceLoader* and *ExecutionLoader*, providing concrete implementations of *ResourceLoader.get_data()* and *ExecutionLoader.get_filename()*.

The *fullname* argument is a fully resolved name of the module the loader is to handle. The *path* argument is the path to the file for the module.

Nuevo en la versión 3.3.

name

The name of the module the loader can handle.

path

Path to the file of the module.

load_module (*fullname*)

Calls super's *load_module()*.

Obsoleto desde la versión 3.4: Use *Loader.exec_module()* instead.

abstractmethod get_filename (*fullname*)

Returns *path*.

abstractmethod get_data (*path*)

Reads *path* as a binary file and returns the bytes from it.

class `importlib.abc.SourceLoader`

An abstract base class for implementing source (and optionally bytecode) file loading. The class inherits from both *ResourceLoader* and *ExecutionLoader*, requiring the implementation of:

- *ResourceLoader.get_data()*
- *ExecutionLoader.get_filename()* Should only return the path to the source file; sourceless loading is not supported.

The abstract methods defined by this class are to add optional bytecode file support. Not implementing these optional methods (or causing them to raise *NotImplementedError*) causes the loader to only work with source code. Implementing the methods allows the loader to work with source *and* bytecode files; it does not allow for *sourceless* loading where only bytecode is provided. Bytecode files are an optimization to speed up loading by removing the parsing step of Python's compiler, and so no bytecode-specific API is exposed.

path_stats (*path*)

Optional abstract method which returns a *dict* containing metadata about the specified path. Supported dictionary keys are:

- 'mtime' (mandatory): an integer or floating-point number representing the modification time of the source code;
- 'size' (optional): the size in bytes of the source code.

Any other keys in the dictionary are ignored, to allow for future extensions. If the path cannot be handled, *OSError* is raised.

Nuevo en la versión 3.3.

Distinto en la versión 3.4: Raise *OSError* instead of *NotImplementedError*.

path_mtime (*path*)

Optional abstract method which returns the modification time for the specified path.

Obsoleto desde la versión 3.3: This method is deprecated in favour of *path_stats()*. You don't have to implement it, but it is still available for compatibility purposes. Raise *OSError* if the path cannot be handled.

Distinto en la versión 3.4: Raise *OSError* instead of *NotImplementedError*.

set_data (*path*, *data*)

Optional abstract method which writes the specified bytes to a file path. Any intermediate directories which do not exist are to be created automatically.

When writing to the path fails because the path is read-only (*errno.EACCES/PermissionError*), do not propagate the exception.

Distinto en la versión 3.4: No longer raises *NotImplementedError* when called.

get_code (*fullname*)

Concrete implementation of *InspectLoader.get_code()*.

exec_module (*module*)

Concrete implementation of *Loader.exec_module()*.

Nuevo en la versión 3.4.

load_module (*fullname*)

Concrete implementation of *Loader.load_module()*.

Obsoleto desde la versión 3.4: Use *exec_module()* instead.

get_source (*fullname*)

Concrete implementation of *InspectLoader.get_source()*.

is_package (*fullname*)

Concrete implementation of *InspectLoader.is_package()*. A module is determined to be a package if its file path (as provided by *ExecutionLoader.get_filename()*) is a file named `__init__` when the file extension is removed **and** the module name itself does not end in `__init__`.

31.5.4 `importlib.resources` – Resources

Source code: [Lib/importlib/resources.py](#)

Nuevo en la versión 3.7.

This module leverages Python's import system to provide access to *resources* within *packages*. If you can import a package, you can access resources within that package. Resources can be opened or read, in either binary or text mode.

Resources are roughly akin to files inside directories, though it's important to keep in mind that this is just a metaphor. Resources and packages **do not** have to exist as physical files and directories on the file system.

Nota: This module provides functionality similar to [pkg_resources Basic Resource Access](#) without the performance overhead of that package. This makes reading resources included in packages easier, with more stable and consistent semantics.

The standalone backport of this module provides more information on [using importlib.resources](#) and [migrating from pkg_resources to importlib.resources](#).

Loaders that wish to support resource reading should implement a `get_resource_reader(fullname)` method as specified by [importlib.abc.ResourceReader](#).

The following types are defined.

`importlib.resources.Package`

The `Package` type is defined as `Union[str, ModuleType]`. This means that where the function describes accepting a `Package`, you can pass in either a string or a module. Module objects must have a resolvable `__spec__.submodule_search_locations` that is not `None`.

`importlib.resources.Resource`

This type describes the resource names passed into the various functions in this package. This is defined as `Union[str, os.PathLike]`.

The following functions are available.

`importlib.resources.open_binary(package, resource)`

Open for binary reading the *resource* within *package*.

package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). This function returns a `typing.BinaryIO` instance, a binary I/O stream open for reading.

`importlib.resources.open_text(package, resource, encoding='utf-8', errors='strict')`

Open for text reading the *resource* within *package*. By default, the resource is opened for reading as UTF-8.

package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). *encoding* and *errors* have the same meaning as with built-in [open\(\)](#).

This function returns a `typing.TextIO` instance, a text I/O stream open for reading.

`importlib.resources.read_binary(package, resource)`

Read and return the contents of the *resource* within *package* as `bytes`.

package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). This function returns the contents of the resource as `bytes`.

`importlib.resources.read_text(package, resource, encoding='utf-8', errors='strict')`

Read and return the contents of *resource* within *package* as a `str`. By default, the contents are read as strict UTF-8.

package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). *encoding* and *errors* have the same meaning as with built-in `open()`. This function returns the contents of the resource as *str*.

`importlib.resources.path(package, resource)`

Return the path to the *resource* as an actual file system path. This function returns a context manager for use in a `with` statement. The context manager provides a `pathlib.Path` object.

Exiting the context manager cleans up any temporary file created when the resource needs to be extracted from e.g. a zip file.

package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory).

`importlib.resources.is_resource(package, name)`

Return `True` if there is a resource named *name* in the package, otherwise `False`. Remember that directories are *not* resources! *package* is either a name or a module object which conforms to the `Package` requirements.

`importlib.resources.contents(package)`

Return an iterable over the named items within the package. The iterable returns *str* resources (e.g. files) and non-resources (e.g. directories). The iterable does not recurse into subdirectories.

package is either a name or a module object which conforms to the `Package` requirements.

31.5.5 `importlib.machinery` – Importers and path hooks

Source code: [Lib/importlib/machinery.py](#)

This module contains the various objects that help `import` find and load modules.

`importlib.machinery.SOURCE_SUFFIXES`

A list of strings representing the recognized file suffixes for source modules.

Nuevo en la versión 3.3.

`importlib.machinery.DEBUG_BYTECODE_SUFFIXES`

A list of strings representing the file suffixes for non-optimized bytecode modules.

Nuevo en la versión 3.3.

Obsoleto desde la versión 3.5: Use `BYTECODE_SUFFIXES` instead.

`importlib.machinery.OPTIMIZED_BYTECODE_SUFFIXES`

A list of strings representing the file suffixes for optimized bytecode modules.

Nuevo en la versión 3.3.

Obsoleto desde la versión 3.5: Use `BYTECODE_SUFFIXES` instead.

`importlib.machinery.BYTECODE_SUFFIXES`

A list of strings representing the recognized file suffixes for bytecode modules (including the leading dot).

Nuevo en la versión 3.3.

Distinto en la versión 3.5: The value is no longer dependent on `__debug__`.

`importlib.machinery.EXTENSION_SUFFIXES`

A list of strings representing the recognized file suffixes for extension modules.

Nuevo en la versión 3.3.

`importlib.machinery.all_suffixes()`

Returns a combined list of strings representing all file suffixes for modules recognized by the standard import machinery. This is a helper for code which simply needs to know if a filesystem path potentially refers to a module without needing any details on the kind of module (for example, `inspect.getmodulename()`).

Nuevo en la versión 3.3.

class `importlib.machinery.BuiltinImporter`

An *importer* for built-in modules. All known built-in modules are listed in `sys.builtin_module_names`. This class implements the `importlib.abc.MetaPathFinder` and `importlib.abc.InspectLoader` ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

Distinto en la versión 3.5: As part of **PEP 489**, the builtin importer now implements `Loader.create_module()` and `Loader.exec_module()`

class `importlib.machinery.FrozenImporter`

An *importer* for frozen modules. This class implements the `importlib.abc.MetaPathFinder` and `importlib.abc.InspectLoader` ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

Distinto en la versión 3.4: Gained `create_module()` and `exec_module()` methods.

class `importlib.machinery.WindowsRegistryFinder`

A *Finder* for modules declared in the Windows registry. This class implements the `importlib.abc.MetaPathFinder` ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

Nuevo en la versión 3.3.

Obsoleto desde la versión 3.6: Use `site` configuration instead. Future versions of Python may not enable this finder by default.

class `importlib.machinery.PathFinder`

A *Finder* for `sys.path` and package `__path__` attributes. This class implements the `importlib.abc.MetaPathFinder` ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

classmethod `find_spec(fullname, path=None, target=None)`

Class method that attempts to find a *spec* for the module specified by `fullname` on `sys.path` or, if defined, on `path`. For each path entry that is searched, `sys.path_importer_cache` is checked. If a non-`False` object is found then it is used as the *path entry finder* to look for the module being searched for. If no entry is found in `sys.path_importer_cache`, then `sys.path_hooks` is searched for a finder for the path entry and, if found, is stored in `sys.path_importer_cache` along with being queried about the module. If no finder is ever found then `None` is both stored in the cache and returned.

Nuevo en la versión 3.4.

Distinto en la versión 3.5: If the current working directory – represented by an empty string – is no longer valid then `None` is returned but no value is cached in `sys.path_importer_cache`.

classmethod `find_module(fullname, path=None)`

A legacy wrapper around `find_spec()`.

Obsoleto desde la versión 3.4: Use `find_spec()` instead.

classmethod `invalidate_caches()`

Calls `importlib.abc.PathEntryFinder.invalidate_caches()` on all finders stored in `sys.path_importer_cache` that define the method. Otherwise entries in `sys.path_importer_cache` set to `None` are deleted.

Distinto en la versión 3.7: Entries of `None` in `sys.path_importer_cache` are deleted.

Distinto en la versión 3.4: Calls objects in `sys.path_hooks` with the current working directory for `' '` (i.e. the empty string).

class `importlib.machinery.FileFinder` (*path*, **loader_details*)

A concrete implementation of `importlib.abc.PathEntryFinder` which caches results from the file system.

The *path* argument is the directory for which the finder is in charge of searching.

The *loader_details* argument is a variable number of 2-item tuples each containing a loader and a sequence of file suffixes the loader recognizes. The loaders are expected to be callables which accept two arguments of the module's name and the path to the file found.

The finder will cache the directory contents as necessary, making stat calls for each module search to verify the cache is not outdated. Because cache staleness relies upon the granularity of the operating system's state information of the file system, there is a potential race condition of searching for a module, creating a new file, and then searching for the module the new file represents. If the operations happen fast enough to fit within the granularity of stat calls, then the module search will fail. To prevent this from happening, when you create a module dynamically, make sure to call `importlib.invalidate_caches()`.

Nuevo en la versión 3.3.

path

The path the finder will search in.

find_spec (*fullname*, *target=None*)

Attempt to find the spec to handle *fullname* within *path*.

Nuevo en la versión 3.4.

find_loader (*fullname*)

Attempt to find the loader to handle *fullname* within *path*.

invalidate_caches ()

Clear out the internal cache.

classmethod `path_hook` (**loader_details*)

A class method which returns a closure for use on `sys.path_hooks`. An instance of `FileFinder` is returned by the closure using the *path* argument given to the closure directly and *loader_details* indirectly.

If the argument to the closure is not an existing directory, `ImportError` is raised.

class `importlib.machinery.SourceFileLoader` (*fullname*, *path*)

A concrete implementation of `importlib.abc.SourceLoader` by subclassing `importlib.abc.FileLoader` and providing some concrete implementations of other methods.

Nuevo en la versión 3.3.

name

The name of the module that this loader will handle.

path

The path to the source file.

is_package (*fullname*)

Return True if *path* appears to be for a package.

path_stats (*path*)

Concrete implementation of `importlib.abc.SourceLoader.path_stats()`.

set_data (*path*, *data*)

Concrete implementation of `importlib.abc.SourceLoader.set_data()`.

load_module (*name=None*)

Concrete implementation of `importlib.abc.Loader.load_module()` where specifying the name of the module to load is optional.

Obsoleto desde la versión 3.6: Use `importlib.abc.Loader.exec_module()` instead.

class `importlib.machinery.SourcelessFileLoader` (*fullname*, *path*)

A concrete implementation of `importlib.abc.FileLoader` which can import bytecode files (i.e. no source code files exist).

Please note that direct use of bytecode files (and thus not source code files) inhibits your modules from being usable by all Python implementations or new versions of Python which change the bytecode format.

Nuevo en la versión 3.3.

name

The name of the module the loader will handle.

path

The path to the bytecode file.

is_package (*fullname*)

Determines if the module is a package based on *path*.

get_code (*fullname*)

Returns the code object for *name* created from *path*.

get_source (*fullname*)

Returns None as bytecode files have no source when this loader is used.

load_module (*name=None*)

Concrete implementation of `importlib.abc.Loader.load_module()` where specifying the name of the module to load is optional.

Obsoleto desde la versión 3.6: Use `importlib.abc.Loader.exec_module()` instead.

class `importlib.machinery.ExtensionFileLoader` (*fullname, path*)

A concrete implementation of `importlib.abc.ExecutionLoader` for extension modules.

The *fullname* argument specifies the name of the module the loader is to support. The *path* argument is the path to the extension module's file.

Nuevo en la versión 3.3.

name

Name of the module the loader supports.

path

Path to the extension module.

create_module (*spec*)

Creates the module object from the given specification in accordance with **PEP 489**.

Nuevo en la versión 3.5.

exec_module (*module*)

Initializes the given module object in accordance with **PEP 489**.

Nuevo en la versión 3.5.

is_package (*fullname*)

Returns True if the file path points to a package's `__init__` module based on `EXTENSION_SUFFIXES`.

get_code (*fullname*)

Returns None as extension modules lack a code object.

get_source (*fullname*)

Returns None as extension modules do not have source code.

get_filename (*fullname*)

Returns *path*.

Nuevo en la versión 3.4.

class `importlib.machinery.ModuleSpec` (*name, loader, *, origin=None, loader_state=None, is_package=None*)

A specification for a module's import-system-related state. This is typically exposed as the module's `__spec__` attribute. In the descriptions below, the names in parentheses give the corresponding attribute

available directly on the module object. E.g. `module.__spec__.origin == module.__file__`. Note however that while the *values* are usually equivalent, they can differ since there is no synchronization between the two objects. Thus it is possible to update the module's `__path__` at runtime, and this will not be automatically reflected in `__spec__.submodule_search_locations`.

Nuevo en la versión 3.4.

name

`(__name__)`

A string for the fully-qualified name of the module.

loader

`(__loader__)`

The *Loader* that should be used when loading the module. *Finders* should always set this.

origin

`(__file__)`

Name of the place from which the module is loaded, e.g. «builtin» for built-in modules and the filename for modules loaded from source. Normally «origin» should be set, but it may be `None` (the default) which indicates it is unspecified (e.g. for namespace packages).

submodule_search_locations

`(__path__)`

List of strings for where to find submodules, if a package (`None` otherwise).

loader_state

Container of extra module-specific data for use during loading (or `None`).

cached

`(__cached__)`

String for where the compiled module should be stored (or `None`).

parent

`(__package__)`

(Read-only) The fully-qualified name of the package under which the module should be loaded as a submodule (or the empty string for top-level modules). For packages, it is the same as `__name__`.

has_location

Boolean indicating whether or not the module's «origin» attribute refers to a loadable location.

31.5.6 `importlib.util` – Utility code for importers

Source code: [Lib/importlib/util.py](#)

This module contains the various objects that help in the construction of an *importer*.

`importlib.util.MAGIC_NUMBER`

The bytes which represent the bytecode version number. If you need help with loading/writing bytecode then consider `importlib.abc.SourceLoader`.

Nuevo en la versión 3.4.

`importlib.util.cache_from_source (path, debug_override=None, *, optimization=None)`

Return the [PEP 3147/PEP 488](#) path to the byte-compiled file associated with the source *path*. For example, if *path* is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see `get_tag()`; if `sys.implementation.cache_tag` is not defined then `NotImplementedError` will be raised).

The *optimization* parameter is used to specify the optimization level of the bytecode file. An empty string represents no optimization, so `/foo/bar/baz.py` with an *optimization* of `' '` will result in a bytecode path of `/foo/bar/__pycache__/baz.cpython-32.pyc`. `None` causes the interpreter's optimization level to be used. Any other value's string representation is used, so `/foo/bar/baz.py` with an *optimization* of `2` will lead to the bytecode path of `/foo/bar/__pycache__/baz.cpython-32.opt-2.pyc`. The string representation of *optimization* can only be alphanumeric, else `ValueError` is raised.

The *debug_override* parameter is deprecated and can be used to override the system's value for `__debug__`. A `True` value is the equivalent of setting *optimization* to the empty string. A `False` value is the same as setting *optimization* to `1`. If both *debug_override* and *optimization* are not `None` then `TypeError` is raised.

Nuevo en la versión 3.4.

Distinto en la versión 3.5: The *optimization* parameter was added and the *debug_override* parameter was deprecated.

Distinto en la versión 3.6: Accepts a *path-like object*.

`importlib.util.source_from_cache (path)`

Given the *path* to a [PEP 3147](#) file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to [PEP 3147](#) or [PEP 488](#) format, a `ValueError` is raised. If `sys.implementation.cache_tag` is not defined, `NotImplementedError` is raised.

Nuevo en la versión 3.4.

Distinto en la versión 3.6: Accepts a *path-like object*.

`importlib.util.decode_source (source_bytes)`

Decode the given bytes representing source code and return it as a string with universal newlines (as required by `importlib.abc.InspectLoader.get_source()`).

Nuevo en la versión 3.4.

`importlib.util.resolve_name (name, package)`

Resolve a relative module name to an absolute one.

If **name** has no leading dots, then **name** is simply returned. This allows for usage such as `importlib.util.resolve_name('sys', __spec__.parent)` without doing a check to see if the **package** argument is needed.

`ValueError` is raised if **name** is a relative module name but **package** is a false value (e.g. `None` or the empty string). `ValueError` is also raised a relative name would escape its containing package (e.g. requesting `..bacon` from within the `spam` package).

Nuevo en la versión 3.3.

`importlib.util.find_spec (name, package=None)`

Find the *spec* for a module, optionally relative to the specified **package** name. If the module is in `sys.modules`, then `sys.modules[name].__spec__` is returned (unless the *spec* would be `None` or is not set, in which case `ValueError` is raised). Otherwise a search using `sys.meta_path` is done. `None` is returned if no *spec* is found.

If **name** is for a submodule (contains a dot), the parent module is automatically imported.

name and **package** work the same as for `import_module()`.

Nuevo en la versión 3.4.

Distinto en la versión 3.7: Raises `ModuleNotFoundError` instead of `AttributeError` if **package** is in fact not a package (i.e. lacks a `__path__` attribute).

`importlib.util.module_from_spec(spec)`

Create a new module based on `spec` and `spec.loader.create_module`.

If `spec.loader.create_module` does not return `None`, then any pre-existing attributes will not be reset. Also, no `AttributeError` will be raised if triggered while accessing `spec` or setting an attribute on the module.

This function is preferred over using `types.ModuleType` to create a new module as `spec` is used to set as many import-controlled attributes on the module as possible.

Nuevo en la versión 3.5.

`@importlib.util.module_for_loader`

A *decorator* for `importlib.abc.Loader.load_module()` to handle selecting the proper module object to load with. The decorated method is expected to have a call signature taking two positional arguments (e.g. `load_module(self, module)`) for which the second argument will be the module **object** to be used by the loader. Note that the decorator will not work on static methods because of the assumption of two arguments.

The decorated method will take in the **name** of the module to be loaded as expected for a *loader*. If the module is not found in `sys.modules` then a new one is constructed. Regardless of where the module came from, `__loader__` set to `self` and `__package__` is set based on what `importlib.abc.InspectLoader.is_package()` returns (if available). These attributes are set unconditionally to support reloading.

If an exception is raised by the decorated method and a module was added to `sys.modules`, then the module will be removed to prevent a partially initialized module from being in left in `sys.modules`. If the module was already in `sys.modules` then it is left alone.

Distinto en la versión 3.3: `__loader__` and `__package__` are automatically set (when possible).

Distinto en la versión 3.4: Set `__name__`, `__loader__` `__package__` unconditionally to support reloading.

Obsoleto desde la versión 3.4: The import machinery now directly performs all the functionality provided by this function.

`@importlib.util.set_loader`

A *decorator* for `importlib.abc.Loader.load_module()` to set the `__loader__` attribute on the returned module. If the attribute is already set the decorator does nothing. It is assumed that the first positional argument to the wrapped method (i.e. `self`) is what `__loader__` should be set to.

Distinto en la versión 3.4: Set `__loader__` if set to `None`, as if the attribute does not exist.

Obsoleto desde la versión 3.4: The import machinery takes care of this automatically.

`@importlib.util.set_package`

A *decorator* for `importlib.abc.Loader.load_module()` to set the `__package__` attribute on the returned module. If `__package__` is set and has a value other than `None` it will not be changed.

Obsoleto desde la versión 3.4: The import machinery takes care of this automatically.

`importlib.util.spec_from_loader(name, loader, *, origin=None, is_package=None)`

A factory function for creating a `ModuleSpec` instance based on a loader. The parameters have the same meaning as they do for `ModuleSpec`. The function uses available *loader* APIs, such as `InspectLoader.is_package()`, to fill in any missing information on the spec.

Nuevo en la versión 3.4.

`importlib.util.spec_from_file_location(name, location, *, loader=None, submodule_search_locations=None)`

A factory function for creating a `ModuleSpec` instance based on the path to a file. Missing information will be filled in on the spec by making use of loader APIs and by the implication that the module will be file-based.

Nuevo en la versión 3.4.

Distinto en la versión 3.6: Accepts a *path-like object*.

`importlib.util.source_hash(source_bytes)`

Return the hash of `source_bytes` as bytes. A hash-based `.pyc` file embeds the `source_hash()` of the corresponding source file's contents in its header.

Nuevo en la versión 3.7.

class `importlib.util.LazyLoader(loader)`

A class which postpones the execution of the loader of a module until the module has an attribute accessed.

This class **only** works with loaders that define `exec_module()` as control over what module type is used for the module is required. For those same reasons, the loader's `create_module()` method must return `None` or a type for which its `__class__` attribute can be mutated along with not using `slots`. Finally, modules which substitute the object placed into `sys.modules` will not work as there is no way to properly replace the module references throughout the interpreter safely; `ValueError` is raised if such a substitution is detected.

Nota: For projects where startup time is critical, this class allows for potentially minimizing the cost of loading a module if it is never used. For projects where startup time is not essential then use of this class is **heavily** discouraged due to error messages created during loading being postponed and thus occurring out of context.

Nuevo en la versión 3.5.

Distinto en la versión 3.6: Began calling `create_module()`, removing the compatibility warning for `importlib.machinery.BuiltinImporter` and `importlib.machinery.ExtensionFileLoader`.

classmethod `factory(loader)`

A static method which returns a callable that creates a lazy loader. This is meant to be used in situations where the loader is passed by class instead of by instance.

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

31.5.7 Examples

Importing programmatically

To programmatically import a module, use `importlib.import_module()`.

```
import importlib

itertools = importlib.import_module('itertools')
```

Checking if a module can be imported

If you need to find out if a module can be imported without actually doing the import, then you should use `importlib.util.find_spec()`.

```
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

if name in sys.modules:
    print(f"{name!r} already in sys.modules")
elif (spec := importlib.util.find_spec(name)) is not None:
```

(continué en la próxima página)

(proviene de la página anterior)

```

    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
    sys.modules[name] = module
    spec.loader.exec_module(module)
    print(f"{name!r} has been imported")
else:
    print(f"can't find the {name!r} module")

```

Importing a source file directly

To import a Python source file directly, use the following recipe (Python 3.5 and newer only):

```

import importlib.util
import sys

# For illustrative purposes.
import tokenize
file_path = tokenize.__file__
module_name = tokenize.__name__

spec = importlib.util.spec_from_file_location(module_name, file_path)
module = importlib.util.module_from_spec(spec)
sys.modules[module_name] = module
spec.loader.exec_module(module)

```

Setting up an importer

For deep customizations of import, you typically want to implement an *importer*. This means managing both the *finder* and *loader* side of things. For finders there are two flavours to choose from depending on your needs: a *meta path finder* or a *path entry finder*. The former is what you would put on `sys.meta_path` while the latter is what you create using a *path entry hook* on `sys.path_hooks` which works with `sys.path` entries to potentially create a finder. This example will show you how to register your own importers so that import will use them (for creating an importer for yourself, read the documentation for the appropriate classes defined within this package):

```

import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)

# Setting up a meta path finder.
# Make sure to put the finder in the proper location in the list in terms of
# priority.
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location in the list in terms
# of priority.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))

```

Approximating `importlib.import_module()`

Import itself is implemented in Python code, making it possible to expose most of the import machinery through `importlib`. The following helps illustrate the various APIs that `importlib` exposes by providing an approximate implementation of `importlib.import_module()` (Python 3.4 and newer for the `importlib` usage, Python 3.6 and newer for other parts of the code).

```
import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import."""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name = absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path = parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
    else:
        msg = f'No module named {absolute_name!r}'
        raise ModuleNotFoundError(msg, name=absolute_name)
    module = importlib.util.module_from_spec(spec)
    sys.modules[absolute_name] = module
    spec.loader.exec_module(module)
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```

31.6 Using `importlib.metadata`

Nuevo en la versión 3.8.

Source code: [Lib/importlib/metadata.py](#)

Nota: Esta funcionalidad es provisional y puede desviarse de la versión habitual de la semántica de la librería estándar.

`importlib.metadata` is a library that provides for access to installed package metadata. Built in part on Python's import system, this library intends to replace similar functionality in the [entry point API](#) and [metadata API](#) of `pkg_resources`. Along with [importlib.resources](#) in Python 3.7 and newer (backported as [importlib_resources](#) for older versions of Python), this can eliminate the need to use the older and less efficient `pkg_resources` package.

By «installed package» we generally mean a third-party package installed into Python's `site-packages` directory via tools such as [pip](#). Specifically, it means a package with either a discoverable `dist-info` or `egg-info` directory, and metadata defined by [PEP 566](#) or its older specifications. By default, package metadata can live on the file system or in zip archives on `sys.path`. Through an extension mechanism, the metadata can live almost anywhere.

31.6.1 Descripción general

Supongamos que desea obtener la cadena de versión para un paquete que ha instalado con `pip`. Comenzamos creando un entorno virtual e instalando algo en él:

```
$ python3 -m venv example
$ source example/bin/activate
(example) $ pip install wheel
```

Se puede obtener la cadena de versión para `wheel` ejecutando lo siguiente:

```
(example) $ python
>>> from importlib.metadata import version
>>> version('wheel')
'0.32.3'
```

También se puede obtener el conjunto de los puntos de entrada clasificados usando el grupo, como `console_scripts`, `distutils.commands` y otros, como claves. Cada grupo contiene una secuencia de objetos *EntryPoint*.

Se pueden obtener los *metadatos para una distribución*:

```
>>> list(metadata('wheel'))
['Metadata-Version', 'Name', 'Version', 'Summary', 'Home-page', 'Author', 'Author-
→email', 'Maintainer', 'Maintainer-email', 'License', 'Project-URL', 'Project-URL
→', 'Project-URL', 'Keywords', 'Platform', 'Classifier', 'Classifier', 'Classifier
→', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier',
→'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Requires-Python',
→'Provides-Extra', 'Requires-Dist', 'Requires-Dist']
```

También se puede obtener el *número de versión de una distribución*, enumerar sus *archivos constituyentes* y obtener una lista de los *Requerimientos de la distribución* de la distribución.

31.6.2 API funcional

Este paquete provee la siguiente funcionalidad a través de su API pública.

Puntos de entrada

La función `entry_points()` retorna un diccionario con todos los puntos de entrada, utilizando el grupo como claves. Los puntos de entrada están representados por instancias de *EntryPoint*. Cada *EntryPoint* tiene los atributos `.name`, `.group` y `.value`, y un método `.load()` para resolver el valor.

```
>>> eps = entry_points()
>>> list(eps)
['console_scripts', 'distutils.commands', 'distutils.setup_keywords', 'egg_info.
→writers', 'setuptools.installation']
>>> scripts = eps['console_scripts']
>>> wheel = [ep for ep in scripts if ep.name == 'wheel'][0]
>>> wheel
EntryPoint(name='wheel', value='wheel.cli:main', group='console_scripts')
>>> main = wheel.load()
>>> main
<function main at 0x103528488>
```

`group` y `name` son valores arbitrarios definidos por el autor del paquete y, en general, un cliente deseará resolver todos los puntos de entrada para un grupo en particular. Lee [la documentación de setuptools](#) para obtener más información sobre los puntos de entrada, su definición y uso.

Metadatos de distribución

Cada distribución incluye algunos metadatos, que puede extraer utilizando la función `metadata()`:

```
>>> wheel_metadata = metadata('wheel')
```

Las claves de la estructura de datos retornada¹ nombran las palabras clave de los metadatos y sus valores se retornan sin analizar de los metadatos de distribución:

```
>>> wheel_metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

Versiones de distribución

La función `version()` es la forma más rápida para obtener el número de versión de una distribución, como una cadena de caracteres:

```
>>> version('wheel')
'0.32.3'
```

Archivos de distribución

You can also get the full set of files contained within a distribution. The `files()` function takes a distribution package name and returns all of the files installed by this distribution. Each file object returned is a `PackagePath`, a `pathlib.Path` derived object with additional `dist`, `size`, and `hash` properties as indicated by the metadata. For example:

```
>>> util = [p for p in files('wheel') if 'util.py' in str(p)][0]
>>> util
PackagePath('wheel/util.py')
>>> util.size
859
>>> util.dist
<importlib.metadata._hooks.PathDistribution object at 0x101e0cef0>
>>> util.hash
<FileHash mode: sha256 value: bYkw5oMccfazVCoYQwKkkemoVyMAFoR34mmKBx8R1NI>
```

Una vez que se tiene el archivo, también se puede leer su contenido:

```
>>> print(util.read_text())
import base64
import sys
...
def as_bytes(s):
    if isinstance(s, text_type):
        return s.encode('utf-8')
    return s
```

En el caso de que el archivo de metadatos que enumera los archivos (RECORD o SOURCES.txt) falte, `files()` retornará `None`. Para evitar esta condición, si no se sabe si la distribución de destino contiene los metadatos, se puede envolver las llamadas a `files()` con `always_iterable` u otra protección similar.

¹ Technically, the returned distribution metadata object is an `email.message.EmailMessage` instance, but this is an implementation detail, and not part of the stable API. You should only use dictionary-like methods and syntax to access the metadata contents.

Requerimientos de la distribución

Para obtener el conjunto completo de los requerimientos de una distribución, usa la función `requires()`:

```
>>> requires('wheel')
["pytest (>=3.0.0) ; extra == 'test'", "pytest-cov ; extra == 'test'"]
```

31.6.3 Distribuciones

Si bien la API de arriba es el uso más común y conveniente, se puede obtener toda esa información de la clase `Distribution`. Una instancia de `Distribution` es un objeto abstracto que representa los metadatos de un paquete de Python. Se puede obtener la instancia de `Distribución` de la siguiente forma:

```
>>> from importlib.metadata import distribution
>>> dist = distribution('wheel')
```

Por lo tanto, una forma alternativa de obtener el número de versión es mediante la instancia de `Distribución`:

```
>>> dist.version
'0.32.3'
```

Hay todo tipo de metadatos disponibles adicionales en la instancia de `Distribution`:

```
>>> dist.metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
>>> dist.metadata['License']
'MIT'
```

The full set of available metadata is not described here. See [PEP 566](#) for additional details.

31.6.4 Extendiendo el algoritmo de búsqueda

Because package metadata is not available through `sys.path` searches, or package loaders directly, the metadata for a package is found through import system finders. To find a distribution package's metadata, `importlib.metadata` queries the list of *meta path finders* on `sys.meta_path`.

El `PathFinder` predeterminado para Python incluye un enlace que llama a `importlib.metadata.MetadataPathFinder` para encontrar distribuciones cargadas desde rutas basadas en sistemas de archivos típicos.

The abstract class `importlib.abc.MetaPathFinder` defines the interface expected of finders by Python's import system. `importlib.metadata` extends this protocol by looking for an optional `find_distributions` callable on the finders from `sys.meta_path` and presents this extended interface as the `DistributionFinder` abstract base class, which defines this abstract method:

```
@abc.abstractmethod
def find_distributions(context=DistributionFinder.Context()):
    """Return an iterable of all Distribution instances capable of
    loading the metadata for packages for the indicated ``context``.
    """
```

El objeto `DistributionFinder.Context` proporciona propiedades `.path` y `.name` que indican la ruta de búsqueda y los nombres que deben coincidir y puede proporcionar otro contexto relevante.

Lo que esto significa en la práctica es que, para soportar la búsqueda de metadatos en paquetes de distribución en ubicaciones distintas al sistema de archivos, se debe subclassificar `Distribution` e implementar sus métodos abstractos. Luego, en el método `find_distributions()` de un buscador personalizado no hay más que retornar instancias de esta `Distribution` derivada.

Notas al pie

Servicios del lenguaje Python

Python proporciona una serie de módulos para ayudar a trabajar con el lenguaje Python. Estos módulos admiten tokenización, análisis, análisis sintáctico, desensamblado de código de bytes, entre otras funciones.

Estos módulos incluyen:

32.1 `parser` — Acceder a árboles de análisis sintáctico de Python

El módulo `parser` proporciona una interfaz para el analizador sintáctico interno de Python y para el compilador de código de bytes. El propósito principal de esta interfaz es permitir que el código Python edite el árbol de análisis sintáctico de una expresión Python y cree código ejecutable a partir de este. Esto es mejor que intentar analizar y modificar una cadena de caracteres que conforma un fragmento de código Python arbitrario, porque el análisis se realiza de una manera idéntica al código que construye la aplicación. También es más rápido.

Nota: A partir de Python 2.5, es más práctico realizar estas operaciones entre la generación del árbol de sintaxis abstracta (AST) y la etapa de compilación, utilizando para ello el módulo `ast`.

Hay algunas cosas a tener en cuenta sobre este módulo, que son importantes para hacer un uso correcto de las estructuras de datos creadas. Este no es un tutorial sobre cómo editar los árboles de análisis sintáctico para código Python, no obstante, sí que se proporcionan algunos ejemplos de uso del módulo `parser`.

Lo más importante es que se requiere una buena comprensión de la gramática de Python procesada por el analizador sintáctico interno. Para obtener información completa sobre la sintaxis del lenguaje, consulta `reference-index`. El analizador sintáctico en sí se crea a partir de una especificación gramatical definida en el archivo `Grammar/Grammar` en la distribución estándar de Python. Los árboles de análisis sintáctico almacenados en los objetos ST creados por este módulo son la salida real del analizador interno cuando son creados por las funciones `expr()` o `suite()`, descritas a continuación. Los objetos ST creados por `sequence2st()` simulan fielmente esas estructuras. Ten en cuenta que los valores de las secuencias que se consideran «correctas» variarán de una versión de Python a otra, a medida que se revise la gramática formal del lenguaje. Por el contrario, portar código fuente en forma de texto de una versión de Python a otra siempre permitirá crear árboles de análisis correctos en la versión de destino, con la única restricción de la migración a una versión anterior del intérprete que no admita construcciones del lenguaje más recientes. Los árboles de análisis sintáctico no suelen ser compatibles de una versión a otra, aunque el código fuente suele ser compatible con versiones posteriores dentro de una misma serie de versiones principales.

Cada elemento de las secuencias retornadas por `st2list()` o `st2tuple()` tiene una forma simple. Las secuencias que representan elementos no terminales en la gramática siempre tienen una longitud mayor que uno. El primer elemento es un número entero que identifica una regla de producción gramatical. Estos números enteros reciben nombres simbólicos en el archivo de cabecera `Include/graminit.h` de C y en el módulo `symbol` de Python. Cada elemento adicional de la secuencia representa un componente de la producción tal como se reconoce en la cadena de entrada: siempre son secuencias que tienen la misma forma que la original. Un aspecto importante de esta estructura que debe tenerse en cuenta es que las palabras clave utilizadas para identificar el tipo de nodo principal, como la palabra clave `if` en una `if_stmt`, se incluyen en el árbol de nodos sin ningún tratamiento especial. Por ejemplo, la palabra clave `if` está representada por la tupla `(1, 'if')`, donde `1` es el valor numérico asociado con todos los tokens `NAME`, incluidos los nombres de variables y funciones definidos por el usuario. En una forma alternativa, que se retorna cuando se solicita información sobre el número de línea, el mismo token podría representarse como `(1, 'if', 12)`, donde el `12` representa el número de línea en el que se encontró el símbolo terminal.

Los elementos terminales se representan de la misma manera, pero sin ningún elemento secundario y sin la adición del texto fuente que se identificó. El anterior ejemplo de la palabra clave `if` es representativo de esto. Los diversos tipos de símbolos terminales se definen en el archivo de cabecera `Include/token.h` de C y en el módulo `token` de Python.

Los objetos ST no son necesarios para soportar la funcionalidad de este módulo, pero se proporcionan para tres propósitos: para permitir que una aplicación amortice el coste de procesar árboles de análisis sintáctico complejos, para proporcionar una representación en forma de árbol de análisis sintáctico que preserve espacio en memoria, en comparación con la representación una lista o tupla de Python, y para facilitar la creación de módulos adicionales en C que manipulen árboles de análisis sintáctico. Se puede crear una simple clase «contenedora» en Python para ocultar el uso de objetos ST.

El módulo `parser` define funciones para varios propósitos distintos. Los más importantes son crear objetos ST y convertir objetos ST en otras representaciones, como árboles de análisis sintáctico y objetos de código compilado. También existen funciones que sirven para consultar el tipo de árbol de análisis sintáctico representado por un objeto ST.

Ver también:

Módulo `symbol` Constantes útiles que representan los nodos internos del árbol de análisis sintáctico.

Módulo `token` Constantes útiles que representan nodos hoja del árbol de análisis sintáctico y funciones para probar valores de nodos.

32.1.1 Crear objetos ST

Los objetos ST pueden crearse a partir del código fuente o de un árbol de análisis sintáctico. Al crear un objeto ST a partir del código fuente, se utilizan diferentes funciones para crear las formas `'eval'` y `'exec'`.

`parser.expr(source)`

La función `expr()` analiza el parámetro `source` como si fuera una entrada para `compile(source, 'file.py', 'eval')`. Si el análisis sintáctico tiene éxito, se crea un objeto ST para contener la representación del árbol de análisis sintáctico interno; de lo contrario, se lanza una excepción apropiada.

`parser.suite(source)`

La función `suite()` analiza el parámetro `source` como si fuera una entrada válida para `compile(source, 'file.py', 'exec')`. Si el análisis sintáctico tiene éxito, se crea un objeto ST para contener la representación del árbol de análisis sintáctico interno; de lo contrario, se lanza una excepción apropiada.

`parser.sequence2st(sequence)`

Esta función acepta un árbol de análisis sintáctico representado como una secuencia y construye una representación interna si es posible. Si puede validar que el árbol se ajusta a la gramática de Python y que todos los nodos son tipos de nodo válidos en la versión anfitriona de Python, se crea un objeto ST a partir de la representación interna y se retorna a quien la invocó. Si hay un problema creando la representación interna, o si el árbol no se puede validar, se lanza una excepción `ParserError`. No se debe dar por supuesto que un objeto ST creado de esta manera se compila correctamente; las excepciones normalmente generadas en el proceso de compilación aún pueden iniciarse cuando el objeto ST se pasa a `compilest()`. Esto puede indicar problemas no relacionados con la sintaxis (como una excepción `MemoryError`), pero también puede

deberse a construcciones como el resultado de analizar `del f()`, que escapa al analizador de Python pero es verificado por el compilador de código de bytes.

Las secuencias que representan tokens terminales pueden representarse como listas de dos elementos de la forma `(1, 'nombre')` o como listas de tres elementos de la forma `(1, 'nombre', 56)`. Si el tercer elemento está presente, se supone que es un número de línea válido. El número de línea puede especificarse para cualquier subconjunto de los símbolos terminales en el árbol de entrada.

`parser.tuple2st(sequence)`

Esta es la misma función que `sequence2st()`. Este punto de entrada se mantiene solo por compatibilidad con versiones anteriores.

32.1.2 Convertir objetos ST

Los objetos ST, independientemente de la entrada utilizada para crearlos, pueden convertirse en árboles de análisis sintáctico representados como árboles de listas o tuplas, o pueden compilarse en objetos de código ejecutable. Los árboles de análisis sintáctico se pueden extraer con o sin información de numeración de línea.

`parser.st2list(st, line_info=False, col_info=False)`

Esta función acepta un objeto ST de quien la invoca mediante el argumento `st` y retorna una lista de Python que representa el árbol de análisis sintáctico equivalente. La representación de la lista resultante se puede utilizar para la inspección o la creación de un nuevo árbol de análisis sintáctico en forma de lista. Esta función no falla mientras haya memoria disponible para construir la representación de la lista. Si el árbol de análisis sintáctico sólo va a ser usado con fines de inspección, se debe usar `st2tuple()` en su lugar para reducir el consumo de memoria y la fragmentación. Cuando se requiere la representación en forma de lista, esta función es significativamente más rápida que recuperar una representación en forma de tupla y convertirla posteriormente en listas anidadas.

Si `line_info` es verdadero, la información del número de línea se incluirá para todos los tokens terminales como un tercer elemento de la lista que representa el token. Ten en cuenta que el número de línea proporcionado especifica la línea en la que el token *termina*. Esta información se omite si el indicador es falso o se omite.

`parser.st2tuple(st, line_info=False, col_info=False)`

Esta función acepta un objeto ST de quien la llama mediante el argumento `st` y retorna una tupla de Python que representa el árbol de análisis sintáctico equivalente. Aparte de retornar una tupla en lugar de una lista, esta función es idéntica a `st2list()`.

Si `line_info` es verdadero, la información del número de línea se incluirá para todos los tokens terminales como un tercer elemento de la lista que representa el token. Esta información se omite si el indicador es falso o se omite.

`parser.compilest(st, filename='<syntax-tree>')`

El compilador de bytes de Python se puede invocar en un objeto ST para producir objetos de código que se pueden usar como parte de una llamada a las funciones incorporadas `exec()` o `eval()`. Esta función proporciona la interfaz para el compilador, pasando el árbol de análisis sintáctico interno de `st` al analizador sintáctico, utilizando el nombre del archivo fuente especificado por el parámetro `filename`. El valor predeterminado proporcionado para `filename` indica que la fuente era un objeto ST.

La compilación de un objeto ST puede resultar en excepciones relacionadas con la compilación. Un ejemplo sería la excepción `SyntaxError` causada por el árbol de análisis sintáctico para `del f()`: esta declaración se considera legal dentro de la gramática formal de Python pero no es una construcción del lenguaje legal. La excepción `SyntaxError` lanzada para esta condición es realmente generada por el compilador de bytes de Python, por lo que puede ser lanzada en este punto por el módulo `parser`. La mayoría de las causas de errores de compilación se pueden diagnosticar programáticamente mediante la inspección del árbol de análisis sintáctico.

32.1.3 Consultas en objetos ST

Se proporcionan dos funciones que permiten a una aplicación determinar si un ST se creó como una expresión o una suite. Ninguna de estas funciones se puede utilizar para determinar si un ST se creó a partir del código fuente a través de `expr()` o `suite()`, o desde un árbol de análisis mediante `sequence2st()`.

`parser.isexpr(st)`

Esta función retorna `True` cuando `st` representa una forma `'eval'` y retorna `False` en caso contrario. Esto es útil, ya que los objetos de código normalmente no se pueden consultar para esta información utilizando las funciones incorporadas existentes. Ten en cuenta que los objetos de código creados por `compilest()` tampoco pueden consultarse así, además, son idénticos a los creados por la función incorporada `compile()`.

`parser.issuite(st)`

Esta función es un espejo de `isexpr()`, en el sentido de que informa si un objeto ST representa una forma `'exec'`, comúnmente conocida como «suite». No es seguro asumir que esta función es equivalente a `not isexpr(st)`, ya que es posible que se admitan fragmentos sintácticos adicionales en el futuro.

32.1.4 Manejo de errores y excepciones

El módulo `parser` define una única excepción, pero también puede lanzar otras excepciones incorporadas en otras partes del entorno de ejecución de Python. Consulta cada función para obtener información sobre las excepciones que puede generar.

exception `parser.ParserError`

Excepción lanzada cuando se produce un fallo dentro del módulo `parser`. Esto generalmente se produce ante fallos de validación, en lugar de la excepción incorporada `SyntaxError`, lanzada durante el análisis normal. El argumento de la excepción puede ser una cadena de caracteres que describa la razón del error, o también una tupla que contenga la secuencia causante del fallo en el árbol de análisis pasado a `sequence2st()` y una cadena explicativa. Las llamadas a `sequence2st()` deben poder manejar ambos tipos de excepciones, mientras que las llamadas a otras funciones en el módulo solo necesitarán tener en cuenta los valores de cadena simples.

Ten en cuenta que las funciones `compilest()`, `expr()` y `suite()` pueden lanzar excepciones que normalmente son generadas por el proceso de análisis y compilación. Estas incluyen las excepciones incorporadas `MemoryError`, `OverflowError`, `SyntaxError` y `SystemError`. En estos casos, estas excepciones tienen todo el significado que normalmente se asocia a ellas. Consulta las descripciones de cada función para obtener información detallada.

32.1.5 Objetos ST

Se admiten comparaciones de orden y de igualdad entre objetos ST. También se admite la serialización de objetos ST (utilizando el módulo `pickle`).

`parser.STType`

El tipo de los objetos retornados por `expr()`, `suite()` y `sequence2st()`.

Los objetos ST tienen los siguientes métodos:

`ST.compile(filename='<syntax-tree>')`

Igual que `compilest(st, filename)`.

`ST.isexpr()`

Igual que `isexpr(st)`.

`ST.issuite()`

Igual que `issuite(st)`.

`ST.tolist(line_info=False, col_info=False)`

Igual que `st2list(st, line_info, col_info)`.

`ST.totuple(line_info=False, col_info=False)`

Igual que `st2tuple(st, line_info, col_info)`.

32.1.6 Ejemplo: Emulación de `compile()`

Si bien muchas operaciones útiles pueden tener lugar entre el análisis y la generación del código de bytes, la operación más simple es no hacer nada. Para este propósito, usar el módulo `parser` para producir una estructura de datos intermedia es equivalente al siguiente código:

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

La operación equivalente usando el módulo `parser` es algo más larga y permite que el árbol de análisis sintáctico interno intermedio se conserve como un objeto ST:

```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

Una aplicación que necesita tanto ST como objetos de código puede empaquetar este código en funciones fácilmente disponibles:

```
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` — Árboles de sintaxis abstracta

Código fuente: [Lib/ast.py](#)

El módulo `ast` ayuda a las aplicaciones de Python a procesar árboles de la gramática de sintaxis abstracta de Python. La sintaxis abstracta en sí misma puede cambiar con cada versión de Python; Este módulo ayuda a descubrir mediante programación cómo se ve la gramática actual.

Se puede generar un árbol de sintaxis abstracta pasando `ast.PyCF_ONLY_AST` como un indicador de la función incorporada `compile()`, o usando el ayudante `parse()` provisto en este módulo. El resultado será un árbol de objetos cuyas clases todas heredan de `ast.AST`. Se puede compilar un árbol de sintaxis abstracta en un objeto de código Python utilizando la función incorporada `compile()`.

32.2.1 Clases Nodo

class `ast.AST`

Esta es la base de todas las clases de nodo AST. Las clases de nodo reales se derivan del archivo `Parser/Python.asdl`, que se reproduce *abajo*. Se definen en el módulo `_ast` C y se reexportan en `ast`.

Hay una clase definida para cada símbolo del lado izquierdo en la gramática abstracta (por ejemplo, `ast.stmt` o `ast.expr`). Además, hay una clase definida para cada constructor en el lado derecho; estas clases heredan de las clases para los árboles del lado izquierdo. Por ejemplo, `ast.BinOp` hereda de `ast.expr`. Para las reglas de producción con alternativas (también conocidas como «sumas»), la clase del lado izquierdo es abstracta: solo se crean instancias de nodos de constructor específicos.

_fields

Cada clase concreta tiene un atributo `_fields` que proporciona los nombres de todos los nodos secundarios.

Cada instancia de una clase concreta tiene un atributo para cada nodo secundario, del tipo definido en la gramática. Por ejemplo, las instancias `ast.BinOp` tienen un atributo `left` de tipo `ast.expr`.

Si estos atributos están marcados como opcionales en la gramática (usando un signo de interrogación), el valor podría ser `None`. Si los atributos pueden tener cero o más valores (marcados con un asterisco), los valores se representan como listas de Python. Todos los atributos posibles deben estar presentes y tener valores válidos al compilar un AST con `compile()`.

lineno

col_offset

end_lineno

end_col_offset

Las instancias de las subclases `ast.expr` y `ast.stmt` tienen atributos `lineno`, `col_offset`, `lineno`, y `col_offset`. Los `lineno` y `end_lineno` son los números de la primera y última línea del intervalo de texto de origen (1 indexado, por lo que la primera línea es la línea 1) y el `col_offset` y `end_col_offset` son las correspondientes compensaciones de bytes UTF-8 del primer y último token que generó el nodo. El desplazamiento UTF-8 se registra porque el analizador utiliza UTF-8 internamente.

Tenga en cuenta que el compilador no requiere las posiciones finales y, por lo tanto, son opcionales. El desplazamiento final es *después* del último símbolo, por ejemplo, uno puede obtener el segmento fuente de un nodo de expresión de una línea usando `source_line[node.col_offset: node.end_col_offset]`.

El constructor de una clase `ast.T` analiza sus argumentos de la siguiente manera:

- Si hay argumentos posicionales, debe haber tantos como elementos en `T._fields`; serán asignados como atributos de estos nombres.
- Si hay argumentos de palabras clave, establecerán los atributos de los mismos nombres a los valores dados.

Por ejemplo, para crear y completar un nodo `ast.UnaryOp`, puede usar

```
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Constant()
node.operand.value = 5
node.operand.lineno = 0
node.operand.col_offset = 0
node.lineno = 0
node.col_offset = 0
```

o la más compacta

```
node = ast.UnaryOp(ast.USub(), ast.Constant(5, lineno=0, col_offset=0),
                  lineno=0, col_offset=0)
```


Distinto en la versión 3.8: La clase `ast.Constant` ahora se usa para todas las constantes.

Obsoleto desde la versión 3.8: Las clases antiguas `ast.Num`, `ast.Str`, `ast.Bytes`, `ast.NameConstant` y `ast.Ellipsis` todavía están disponibles, pero se eliminarán en futuras versiones de Python. Mientras tanto, instanciarlos retornará una instancia de una clase diferente.

32.2.2 Gramática abstracta

La gramática abstracta se define actualmente de la siguiente manera:

```
-- ASDL's 5 builtin types are:
-- identifier, int, string, object, constant

module Python
{
    mod = Module(stmt* body, type_ignore *type_ignores)
        | Interactive(stmt* body)
        | Expression(expr body)
        | FunctionType(expr* argtypes, expr returns)

    -- 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, expr? returns,
                        string? type_comment)
        | AsyncFunctionDef(identifier name, arguments args,
                           stmt* body, expr* decorator_list, expr? returns,
                           string? type_comment)

        | ClassDef(identifier name,
                    expr* bases,
                    keyword* keywords,
                    stmt* body,
                    expr* decorator_list)
        | Return(expr? value)

        | Delete(expr* targets)
        | Assign(expr* targets, expr value, string? type_comment)
        | AugAssign(expr target, operator op, expr value)
        -- 'simple' indicates that we annotate simple name without parens
        | AnnAssign(expr target, expr annotation, expr? value, int simple)

    -- use 'orelse' because else is a keyword in target languages
    | For(expr target, expr iter, stmt* body, stmt* orelse, string? type_
↪comment)
    | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, string? ↪
↪type_comment)
        | While(expr test, stmt* body, stmt* orelse)
        | If(expr test, stmt* body, stmt* orelse)
        | With(withitem* items, stmt* body, string? type_comment)
        | AsyncWith(withitem* items, stmt* body, string? type_comment)

        | Raise(expr? exc, expr? cause)
        | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
        | Assert(expr test, expr? msg)

        | Import(alias* names)
        | ImportFrom(identifier? module, alias* names, int? level)

        | Global(identifier* names)
```

(continué en la próxima página)

(proviene de la página anterior)

```

    | Nonlocal(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, int? end_lineno, int? end_col_
↪offset)

    -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
    | NamedExpr(expr target, expr value)
    | 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)
    | 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
    | Await(expr value)
    | Yield(expr? value)
    | YieldFrom(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)
    | FormattedValue(expr value, int? conversion, expr? format_spec)
    | JoinedStr(expr* values)
    | Constant(constant value, string? kind)

    -- the following expression can appear in assignment context
    | Attribute(expr value, identifier attr, expr_context ctx)
    | Subscript(expr value, slice slice, expr_context ctx)
    | Starred(expr value, 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, int? end_lineno, int? end_col_
↪offset)

    expr_context = Load | Store | Del | AugLoad | AugStore | Param

    slice = Slice(expr? lower, expr? upper, expr? step)
    | ExtSlice(slice* dims)
    | Index(expr value)

    boolop = And | Or

    operator = Add | Sub | Mult | MatMult | 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

```

(continué en la próxima página)

(proviene de la página anterior)

```

comprehension = (expr target, expr iter, expr* ifs, int is_async)

except_handler = ExceptHandler(expr? type, identifier? name, stmt* body)
    attributes (int lineno, int col_offset, int? end_lineno, int?
↳end_col_offset)

arguments = (arg* posonlyargs, arg* args, arg? vararg, arg* kwonlyargs,
    expr* kw_defaults, arg? kwarg, expr* defaults)

arg = (identifier arg, expr? annotation, string? type_comment)
    attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↳offset)

-- keyword arguments supplied to call (NULL identifier for **kwargs)
keyword = (identifier? arg, expr value)

-- import name with optional 'as' alias.
alias = (identifier name, identifier? asname)

withitem = (expr context_expr, expr? optional_vars)

type_ignore = TypeIgnore(int lineno, string tag)
}

```

32.2.3 Ayudantes de ast

Además de las clases de nodo, el módulo `ast` define estas funciones y clases de utilidad para atravesar árboles de sintaxis abstracta:

`ast.parse` (*source*, *filename*='<unknown>', *mode*='exec', *, *type_comments*=False, *feature_version*=None)
 Analiza la fuente en un nodo AST. Equivalente a `compile(source, filename, mode, ast.PyCF_ONLY_AST)`.

Si se proporciona `type_comments=True`, el analizador se modifica para verificar y retornar los comentarios de tipo según lo especificado por [PEP 484](#) y [PEP 526](#). Esto es equivalente a agregar `ast.PyCF_TYPE_COMMENTS` a los flags pasados a `compile()`. Esto informará errores de sintaxis para comentarios de tipo fuera de lugar. Sin este flag, los comentarios de tipo se ignorarán y el campo `type_comment` en los nodos AST seleccionados siempre será `None`. Además, las ubicaciones de los comentarios `# type: ignore` se retornarán como el atributo `type_ignores` de `Module` (de lo contrario, siempre es una lista vacía).

Además, si `modo` es `'func_type'`, la sintaxis de entrada se modifica para corresponder a [PEP 484](#) «comentarios de tipo de firma», por ejemplo `(str, int) -> List[str]`.

Además, establece `feature_version` en una tupla (`major`, `minor`) intentará analizar usando la gramática de esa versión de Python. Actualmente `major` debe ser igual a 3. Por ejemplo, establece `feature_version=(3, 4)` permitirá el uso de `async` y `await` como nombres de variables. La versión más baja admitida es (3, 4); la más alta es `sys.version_info[0:2]`.

Advertencia: Es posible bloquear el intérprete de Python con una cadena de caracteres suficientemente grande/compleja debido a las limitaciones de profundidad de pila en el compilador AST de Python.

Distinto en la versión 3.8: Se agregaron `type_comments`, `mode='func_type'` y `feature_version`.

`ast.literal_eval` (*node_or_string*)
 Evalúa de forma segura un nodo de expresión o una cadena de caracteres que contenga un literal de Python o un visualizador de contenedor. La cadena o nodo proporcionado solo puede consistir en las siguientes estructuras

literales de Python: cadenas de caracteres, bytes, números, tuplas, listas, diccionarios, conjuntos, booleanos y `None`.

Esto se puede usar para evaluar de forma segura las cadenas de caracteres que contienen valores de Python de fuentes no confiables sin la necesidad de analizar los valores uno mismo. No es capaz de evaluar expresiones complejas arbitrariamente, por ejemplo, que involucran operadores o indexación.

Advertencia: Es posible bloquear el intérprete de Python con una cadena de caracteres suficientemente grande/compleja debido a las limitaciones de profundidad de pila en el compilador AST de Python.

Distinto en la versión 3.2: Ahora permite bytes y establece literales.

`ast.get_docstring (node, clean=True)`

Retorna la cadena de caracteres de documentación del *node* dado (que debe ser un nodo `FunctionDef`, `AsyncFunctionDef`, `ClassDef`, o `Module`), o `None` si no tiene docstring. Si *clean* es verdadero, limpia la sangría del docstring con `inspect.cleandoc()`.

Distinto en la versión 3.5: `AsyncFunctionDef` ahora está soportada.

`ast.get_source_segment (source, node, *, padded=False)`

Obtenga el segmento de código fuente del *source* que generó *node*. Si falta información de ubicación (`lineno`, `end_lineno`, `col_offset`, o `end_col_offset`), retorna `None`.

Si *padded* es `True`, la primera línea de una declaración de varias líneas se rellenará con espacios para que coincidan con su posición original.

Nuevo en la versión 3.8.

`ast.fix_missing_locations (node)`

Cuando compila un árbol de nodos con `compile()`, el compilador espera los atributos `lineno` y `col_offset` para cada nodo que los soporta. Es bastante tedioso completar los nodos generados, por lo que este ayudante agrega estos atributos de forma recursiva donde aún no están establecidos, configurándolos en los valores del nodo principal. Funciona de forma recursiva comenzando en *node*.

`ast.increment_lineno (node, n=1)`

Incrementa el número de línea y el número de línea final de cada nodo en el árbol comenzando en *node* por *n*. Esto es útil para «mover código» a una ubicación diferente en un archivo.

`ast.copy_location (new_node, old_node)`

Copia la ubicación de origen (`lineno`, `col_offset`, `end_lineno`, y `end_col_offset`) de *old_node* a *new_node* si es posible, y retorna *new_node*.

`ast.iter_fields (node)`

Produce (*yield*) una tupla de (`fieldname`, `value`) para cada campo en `node._fields` que está presente en *node*.

`ast.iter_child_nodes (node)`

Cede todos los nodos secundarios directos de *node*, es decir, todos los campos que son nodos y todos los elementos de campos que son listas de nodos.

`ast.walk (node)`

Recursivamente produce todos los nodos descendientes en el árbol comenzando en *node* (incluido *node* en sí mismo), en ningún orden especificado. Esto es útil si solo desea modificar los nodos en su lugar y no le importa el contexto.

class `ast.NodeVisitor`

Una clase base de visitante de nodo que recorre el árbol de sintaxis abstracta y llama a una función de visitante para cada nodo encontrado. Esta función puede retornar un valor que se reenvía mediante el método `visit()`.

Esta clase está destinada a ser subclase, con la subclase agregando métodos de visitante.

visit (*node*)

Visita un nodo. La implementación predeterminada llama al método llamado `self`.

`visit_classname` donde *classname* es el nombre de la clase de nodo, o `generic_visit()` si ese método no existe.

generic_visit (*node*)

Este visitante llama `visit()` en todos los hijos del nodo.

Tenga en cuenta que los nodos secundarios de los nodos que tienen un método de visitante personalizado no se visitarán a menos que el visitante llame `generic_visit()` o los visite a sí mismo.

No use `NodeVisitor` si desea aplicar cambios a los nodos durante el recorrido. Para esto existe un visitante especial (`NodeTransformer`) que permite modificaciones.

Obsoleto desde la versión 3.8: Los métodos `visit_Num()`, `visit_Str()`, `visit_Bytes()`, `visit_NameConstant()` y `visit_Ellipsis()` están en desuso ahora y no serán llamados en futuras versiones de Python. Agregue el método `visit_Constant()` para manejar todos los nodos constantes.

class `ast.NodeTransformer`

Una subclase de `NodeVisitor` que recorre el árbol de sintaxis abstracta y permite la modificación de nodos.

La clase `NodeTransformer` recorrerá el AST y usará el valor de retorno de los métodos del visitante para reemplazar o eliminar el nodo anterior. Si el valor de retorno del método de visitante es `None`, el nodo se eliminará de su ubicación; de lo contrario, se reemplazará con el valor de retorno. El valor de retorno puede ser el nodo original, en cuyo caso no se realiza ningún reemplazo.

Aquí hay un transformador de ejemplo que reescribe todas las apariciones de búsquedas de nombres (`foo`) en `data['foo']`:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Constant(value=node.id)),
            ctx=node.ctx
        )
```

Tenga en cuenta que si el nodo en el que está operando tiene nodos secundarios, debe transformar los nodos secundarios usted mismo o llamar primero al método `generic_visit()` para el nodo.

Para los nodos que formaban parte de una colección de declaraciones (que se aplica a todos los nodos de declaración), el visitante también puede retornar una lista de nodos en lugar de solo un nodo.

Si `NodeTransformer` introduce nuevos nodos (que no eran parte del árbol original) sin darles información de ubicación (como `lineno`), `fix_missing_locations()` debería llamarse con el nuevo sub-árbol para recalcular la información de ubicación

```
tree = ast.parse('foo', mode='eval')
new_tree = fix_missing_locations(RewriteName().visit(tree))
```

Usualmente usas el transformador así:

```
node = YourTransformer().visit(node)
```

`ast.dump` (*node*, *annotate_fields=True*, *include_attributes=False*)

Retorna un volcado formateado del árbol en *node*. Esto es principalmente útil para propósitos de depuración. Si *annotate_fields* es verdadero (por defecto), la cadena de caracteres retornada mostrará los nombres y los valores de los campos. Si *annotate_fields* es falso, la cadena de resultados será más compacta omitiendo nombres de campo no ambiguos. Los atributos como los números de línea y las compensaciones de columna no se vuelcan de forma predeterminada. Si esto se desea, *include_attributes* se puede establecer en verdadero.

Ver también:

[Green Tree Snakes](#), un recurso de documentación externo, tiene buenos detalles sobre cómo trabajar con Python AST.

`ASTTokens` annotates Python ASTs with the positions of tokens and text in the source code that generated them. This is helpful for tools that make source code transformations.

`leoAst.py` unifies the token-based and parse-tree-based views of python programs by inserting two-way links between tokens and ast nodes.

`LibCST` parses code as a Concrete Syntax Tree that looks like an ast tree and keeps all formatting details. It's useful for building automated refactoring (codemod) applications and linters.

`Parso` is a Python parser that supports error recovery and round-trip parsing for different Python versions (in multiple Python versions). `Parso` is also able to list multiple syntax errors in your python file.

32.3 `symtable` — Acceso a la tabla de símbolos del compilador

Código fuente: `Lib/symtable.py`

Las tablas de símbolos son generadas por el compilador a partir del AST justo antes de que el bytecode sea generado. La tabla de símbolos es responsable de calcular el ámbito de cada identificador en el código. `symtable` provee una interfaz para examinar esas tablas.

32.3.1 Generando tablas de símbolos

`symtable.symtable` (*code*, *filename*, *compile_type*)

Retorna la `SymbolTable` del nivel más alto para el código Python *code*. *filename* es el nombre del archivo conteniendo el código. *compile_type* es como el argumento *mode* de la función `compile()`.

32.3.2 Examinando la tabla de símbolos

class `symtable.SymbolTable`

Un espacio de nombres para el bloque. El constructor no es público.

get_type()

Retorna el tipo de la tabla de símbolos. Los valores posibles son 'class', 'module' y 'function'.

get_id()

Retorna el identificador de la tabla.

get_name()

Retorna el nombre de la tabla. Este es el nombre de la clase si la tabla es para una clase, el nombre de la función si la tabla es para una función, o 'top' si la tabla es global (`get_type()` retorna 'module').

get_lineno()

Retorna el número de la primera línea en el bloque que esta tabla representa.

is_optimized()

Retorna True si los locales en esta tabla pueden ser optimizados.

is_nested()

Retorna True si el bloque es una clase o función anidadas.

has_children()

Retorna True si el bloque contiene espacios de nombres anidados en él. Estos pueden ser obtenidos con `get_children()`.

has_exec()

Retorna True si el bloque usa `exec`.

get_identifiers()

Retorna una lista con los nombres de los símbolos en esta tabla.

lookup(name)

Busca *name* en la tabla y retorna una instancia de *Symbol*.

get_symbols()

Retorna una lista de instancias de *Symbol* de los nombres en la tabla.

get_children()

Retorna una lista de las tablas de símbolos anidadas.

class symtable.Function

Un espacio de nombres para una función o método. Esta clase hereda de *SymbolTable*.

get_parameters()

Retorna una tupla conteniendo los nombres de los parámetros de esta función.

get_locals()

Retorna una tupla conteniendo los nombres de los locales en esta función.

get_globals()

Retorna una tupla conteniendo los nombres de los globales en esta función.

get_nonlocals()

Retorna una tupla conteniendo los nombres de los no locales en esta función.

get_frees()

Retorna una tupla conteniendo los nombres de las variables libres en esta función.

class symtable.Class

Un espacio de nombres de una clase. Esta clase hereda de *SymbolTable*.

get_methods()

Retorna una tupla conteniendo los nombres de los métodos declarados en la clase.

class symtable.Symbol

Una entrada en una *SymbolTable* correspondiente a un identificador en el código. El constructor no es público.

get_name()

Retorna el nombre del símbolo.

is_referenced()

Retorna *True* si el símbolo es usado en su bloque.

is_imported()

Retorna *True* si el símbolo es creado desde una instrucción *import*.

is_parameter()

Retorna *True* si el símbolo es un parámetro.

is_global()

Retorna *True* si el símbolo es global.

is_nonlocal()

Retorna *True* si el símbolo es no local.

is_declared_global()

Retorna *True* si el símbolo es declarado global con una instrucción *global*.

is_local()

Retorna *True* si el símbolo es local a su bloque.

is_annotated()

Return *True* if the symbol is annotated.

Nuevo en la versión 3.6.

is_free()

Retorna True si el símbolo es referenciado en su bloque pero no asignado.

is_assigned()

Retorna True si el símbolo es asignado en su bloque.

is_namespace()

Retorna True si la vinculación de nombres introduce un nuevo espacio de nombres.

Si el nombre es usado como objetivo de una instrucción *function* o *class* retornará verdadero.

Por ejemplo:

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

Note que un solo nombre puede estar vinculado a varios objetos. Si el resultado es True, el nombre puede estar vinculado también a otros objetos, como un entero o una lista, esto no introduce un nuevo espacio de nombres.

get_namespaces()

Retorna una lista de espacios de nombres vinculados a este nombre.

get_namespace()

Retorna el espacio de nombres vinculado a este nombre. Si más de un espacio de nombres esta vinculado se lanza un *ValueError*.

32.4 `symbol` — Constantes utilizadas con árboles de análisis de Python

Código fuente: [Lib/symbol.py](#)

Este módulo proporciona constantes que representan los valores numéricos de nodos internos del árbol de análisis. A diferencia de la mayoría de las constantes de Python, estas utilizan nombres en minúsculas. Refiérase al archivo `Grammar/Grammar` en la distribución de Python para las definiciones de los nombres en el contexto de la gramática del lenguaje. Los valores numéricos específicos que corresponden a los nombres pueden variar entre versiones de Python.

Este módulo también proporciona un objeto de datos adicional:

`symbol.sym_name`

Diccionario que mapea los valores numéricos de las constantes definidas en este módulo a cadenas de nombre, lo que permite generar una representación más legible por humanos de árboles de análisis.

32.5 `token`— Constantes usadas con árboles de sintaxis de Python

Código fuente: [Lib/token.py](#)

Este módulo provee constantes que representan los valores numéricos de nodos hoja de un árbol de sintaxis (tokens terminales). Referirse al archivo `Grammar/Grammar` en la distribución de Python para las definiciones de los nombres en el contexto de gramática del lenguaje. Los valores numéricos específicos a los que los nombres mapean pueden cambiar entre versiones de Python.

El módulo también proporciona un mapeo de códigos numéricos a nombres y algunas funciones. Las funciones asemejan definiciones en los archivos Python C encabezados.

`token.tok_name`

Diccionario que mapea los valores numéricos de las constantes definidas en este módulo a cadenas de nombres, permitiendo una representación de árboles de sintaxis a ser generados más legible para humanos.

`token.ISTERMINAL` (*x*)

Retorna `True` para valores token terminales.

`token.ISNONTERMINAL` (*x*)

Retorna `True` para valores token no terminales.

`token.ISEOF` (*x*)

Retorna `True` si *x* es el marcador indicando el final del input.

Las constantes de token son:

`token.ENDMARKER`

`token.NAME`

`token.NUMBER`

`token.STRING`

`token.NEWLINE`

`token.INDENT`

`token.DEDENT`

`token.LPAR`

Valor token para " (".

`token.RPAR`

Valor token para ") ".

`token.LSQB`

Valor token para " [".

`token.RSQB`

Valor token para "] ".

`token.COLON`

Valor token para " : ".

`token.COMMA`

Valor token para " , ".

`token.SEMI`

Valor token para " ; ".

`token.PLUS`

Valor token para " + ".

`token.MINUS`

Valor token para " - ".

`token.STAR`

Valor token para " * ".

`token.SLASH`

Valor token para " / ".

`token.VBAR`

Valor token para " | ".

`token.AMPER`

Valor token para " & ".

`token.LESS`

Valor token para " < ".

`token.GREATER`
Valor token para ">".

`token.EQUAL`
Valor token para "=".

`token.DOT`
Valor token para ".".

`token.PERCENT`
Valor token para "%".

`token.LBRACE`
Valor token para "{".

`token.RBRACE`
Valor token para "}".

`token.EQEQUAL`
Valor token para "==".

`token.NOTEQUAL`
Valor token para "!=".

`token.LESSEQUAL`
Valor token para "<=".

`token.GREATEREQUAL`
Valor token para ">=".

`token.TILDE`
Valor token para "~".

`token.CIRCUMFLEX`
Valor token para "^".

`token.LEFTSHIFT`
Valor token para "<<".

`token.RIGHTSHIFT`
Valor token para ">>".

`token.DOUBLESTAR`
Valor token para "**".

`token.PLUSEQUAL`
Valor token para "+=".

`token.MINEQUAL`
Valor token para "-=".

`token.STAREQUAL`
Valor token para "*=".

`token.SLASHEQUAL`
Valor token para "/=".

`token.PERCENTEQUAL`
Valor token para "%=".

`token.AMPEREQUAL`
Valor token para "&=".

`token.VBAREQUAL`
Valor token para "|=".

`token.CIRCUMFLEXEQUAL`
Valor token para "^=".

`token.LEFTSHIFTEQUAL`
 Valor token para "<=<=".

`token.RIGHTSHIFTEQUAL`
 Valor token para ">>=".

`token.DOUBLESTAREQUAL`
 Valor token para "**=".

`token.DOUBLESASH`
 Valor token para "//".

`token.DOUBLESASHESQUAL`
 Valor token para "//=".

`token.AT`
 Valor token para "@".

`token.ATEQUAL`
 Valor token para "@=".

`token.RARROW`
 Valor token para "->".

`token.ELLIPSIS`
 Valor token para "...".

`token.COLONEQUAL`
 Valor token para "!=".

`token.OP`

`token.AWAIT`

`token.ASYNC`

`token.TYPE_IGNORE`

`token.TYPE_COMMENT`

`token.ERRORTOKEN`

`token.N_TOKENS`

`token.NT_OFFSET`

Los siguientes tipos de valores tokens no son usados por el tokenizador C pero son necesarios para el modulo tokenizador.

`token.COMMENT`
 Valores token usados para indicar un comentario.

`token.NL`
 Valor token usado para indicar una nueva línea no terminante. El token *NEWLINE* indica el final de una línea lógica de código Python; los tokens NL son generados cuando una línea lógica de código es continuada sobre múltiples líneas físicas.

`token.ENCODING`
 Valor de token que indica la codificación usada para decodificar los bytes de origen en texto. El primer token retornado por *tokenize.tokenize()* siempre será un token ENCODING.

`token.TYPE_COMMENT`
 Valor token indicando que un tipo comentario fue reconocido. Dichos tokens solo son producidos cuando *ast.parse()* es invocado con *type_comments=True*.

Distinto en la versión 3.5: Agregados los tokens *AWAIT* y *ASYNC*.

Distinto en la versión 3.7: Agregados los tokens *COMMENT*, *NL* y *ENCODING*.

Distinto en la versión 3.7: Removidos los tokens *AWAIT* y *ASYNC*. «async» y «await» son ahora tokenizados como *NAME* tokens.

Distinto en la versión 3.8: Agregados `TYPE_COMMENT`, `TYPE_IGNORE`, `COLONEQUAL`. Agregados de regreso los tokens `AWAIT` y `ASYNC` (son necesarios para dar soporte en la sintaxis de versiones más antiguas de Python para `ast.parse()` con `feature_version` establecido a 6 o menor).

32.6 keyword — Pruebas para palabras clave en Python

Código fuente: [Lib/keyword.py](#)

Este módulo permite a un programa Python determinar si una cadena de caracteres es una palabra clave.

`keyword.iskeyword(s)`

Retorna True si `s` es una palabra clave Python.

`keyword.kwlist`

Secuencia que contiene todas las palabras clave definidas para el intérprete. Si cualquier palabra clave es definida para estar activa sólo cuando las declaraciones particulares `__future__` están vigentes, estas se incluirán también.

32.7 tokenize — Conversor a tokens para código Python

Código fuente: [Lib/tokenize.py](#)

El módulo `tokenize` provee un analizador léxico para código fuente Python, implementado en Python. Este analizador también retorna comentarios como tokens, siendo útil para implementar «pretty-printers», como `colorizers` para impresiones en pantalla.

Para simplificar el manejo de flujos de tokens, todos los tokens operator y delimiter y `Ellipsis` se retorna usando el tipo genérico `OP`. El tipo exacto se puede determinar usando la propiedad `exact_type` en la *named tuple* retornada por `tokenize.tokenize()`.

32.7.1 Convirtiendo la entrada en tokens

El punto de entrada principal es un *generador*:

`tokenize.tokenize(readline)`

El generador `tokenize()` requiere un argumento, `readline`, que debe ser un objeto invocable que provee la misma interfaz que el método `io.IOBase.readline()` de los objetos archivos. Cada llamada a la función debe retornar una línea de entrada como bytes.

El generador produce una tupla con los siguientes 5 miembros: El tipo de token, la cadena del token, una tupla (`srow`, `scol`) de enteros especificando la fila y columna donde el token empieza en el código, una (`erow`, `ecol`) de enteros especificando la fila y columna donde el token acaba en el código, y la línea en la que se encontró el token. La línea pasada (el último elemento de la tupla) es la línea *física*. La tupla se retorna como una *named tuple* con los campos: `type` `string` `start` `end` `line`.

La *named tuple* retorna tiene una propiedad adicional llamada `exact_type` que contiene el tipo de operador exacto para tokens `OP`. Para todos los otros tipos de token, `exact_type` es el valor del campo `type` de la tupla con su respectivo nombre.

Distinto en la versión 3.1: Añadido soporte para tuplas con nombre.

Distinto en la versión 3.3: Añadido soporte para `exact_type`.

`tokenize()` determina la codificación del fichero buscando una marca BOM UTF-8 o una *cookie* de codificación, de acuerdo con **PEP 263**.

`tokenize.generate_tokens(readline)`

Convertir a tokens una fuente leyendo cadenas unicode en lugar de bytes.

Como `tokenize()`, el argumento `readline` es un invocable que retorna una sola línea de entrada. Sin embargo, `generate_tokens()` espera que `readline` retorne un objeto `str` en lugar de `bytes`.

El resultado es un iterador que produce tuplas con nombre, exactamente como `tokenize()`. No produce un token `ENCODING`.

Todas las constantes del módulo `token` se exportan también desde `tokenize`.

Otra función se encarga de invertir el proceso de conversión. Esto es útil para crear herramientas que convierten a tokens un script, modificar el flujo de token, y escribir el script modificado.

`tokenize.untokenize(iterable)`

Convierte los `tokens` de vuelta en código fuente Python. El `iterable` debe retornar secuencias con al menos dos elementos, el tipo de `token` y la cadena del `token`. Cualquier otro elemento de la secuencia es ignorado.

El `script` reconstruido se retorna como una cadena simple. El resultado está garantizado de que se convierte en `tokens` de vuelta a la misma entrada, de modo que la conversión no tiene pérdida y que las conversiones de ida y de vuelta están aseguradas. La garantía aplica sólo al tipo y la cadena del `token`, ya que el espaciado entre `tokens` (posiciones de columna) pueden variar.

Retorna bytes, codificados usando el token `ENCODING`, que es el primer elemento de la secuencia retornada por `tokenize()`. Si no hay un token de codificación en la entrada, retorna una cadena.

`tokenize()` necesita detectar la codificación de los ficheros fuente que convierte a tokens. La función que utiliza para hacer esto está disponible como:

`tokenize.detect_encoding(readline)`

La función `detect_encoding()` se usa para detectar la codificación que debería usarse al leer un fichero fuente Python. Requiere un argumento, `readline`, del mismo modo que el generador `tokenize()`.

Lamará a `readline` un máximo de dos veces, retornando la codificación usada, como cadena, y una lista del resto de líneas leídas, no descodificadas de `bytes`.

Detecta la codificación a partir de la presencia de una marca BOM UTF-8 o una `cookie` de codificación, como se especifica en **PEP 263**. Si ambas están presentes pero en desacuerdo, se lanzará un `SyntaxError`. Resaltar que si se encuentra la marca BOM, se retornará `'utf-8-sig'` como codificación.

Si no se especifica una codificación, por defecto se retornará `'utf-8'`.

Usa `open()` para abrir ficheros fuente Python: Ésta utiliza `detect_encoding()` para detectar la codificación del fichero.

`tokenize.open(filename)`

Abrir un fichero en modo sólo lectura usando la codificación detectada por `detect_encoding()`.

Nuevo en la versión 3.2.

exception `tokenize.TokenError`

Lanzada cuando una expresión o docstring que puede separarse en más líneas no está completa en el fichero, por ejemplo:

```
"""Beginning of
docstring
```

o:

```
[1,
 2,
 3
```

Destacar que cadenas con comillas simples sin finalizar no lanzan un error. Se convertirán en tokens como `ERRORTOKEN`, seguido de la conversión de su contenido.

32.7.2 Uso como línea de comandos

Nuevo en la versión 3.3.

El módulo `tokenize` se puede ejecutar como script desde la línea de comandos. Es tan simple como:

```
python -m tokenize [-e] [filename.py]
```

Se aceptan las siguientes opciones:

-h, --help
muestra el mensaje de ayuda y sale

-e, --exact
muestra los nombres de token usando el tipo exacto

Si se especifica `filename.py`, se convierte su contenido a tokens por la salida estándar. En otro caso, se convierte la entrada estándar.

32.7.3 Ejemplos

Ejemplo de un script que transforma literales float en objetos Decimal:

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

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'))"

    The format of the exponent is inherited from the platform C library.
    Known cases are "e-007" (Windows) and "e-07" (not Windows). Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.

    >>> exec(s) #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).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:
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

Ejemplo de conversión desde la línea de comandos. El *script*:

```
def say_hello():
    print("Hello, World!")

say_hello()
```

se convertirá en la salida siguiente, donde la primera columna es el rango de coordenadas línea/columna donde se encuentra el token, la segunda columna es el nombre del token, y la última columna es el valor del token, si lo hay

```
$ python -m tokenize hello.py
0,0-0,0:      ENCODING      'utf-8'
1,0-1,3:      NAME          'def'
1,4-1,13:     NAME          'say_hello'
1,13-1,14:    OP            '('
1,14-1,15:    OP            ')'
1,15-1,16:    OP            ':'
1,16-1,17:    NEWLINE      '\n'
2,0-2,4:      INDENT       '    '
2,4-2,9:      NAME          'print'
2,9-2,10:     OP            '('
2,10-2,25:    STRING        '"Hello, World!'"
2,25-2,26:    OP            ')'
2,26-2,27:    NEWLINE      '\n'
3,0-3,1:      NL           '\n'
4,0-4,0:      DEDENT       ''
4,0-4,9:      NAME          'say_hello'
4,9-4,10:     OP            '('
4,10-4,11:    OP            ')'
4,11-4,12:    NEWLINE      '\n'
5,0-5,0:      ENDMARKER    ''
```

Los nombres de tipos de token exactos se pueden mostrar con la opción `-e`:

```
$ python -m tokenize -e hello.py
0,0-0,0:      ENCODING      'utf-8'
1,0-1,3:      NAME          'def'
1,4-1,13:     NAME          'say_hello'
1,13-1,14:    LPAR         '('
1,14-1,15:    RPAR         ')'
1,15-1,16:    COLON        ':'
1,16-1,17:    NEWLINE      '\n'
2,0-2,4:      INDENT       '    '
2,4-2,9:      NAME          'print'
2,9-2,10:     LPAR         '('
2,10-2,25:    STRING        '"Hello, World!'"
2,25-2,26:    RPAR         ')'
2,26-2,27:    NEWLINE      '\n'
3,0-3,1:      NL           '\n'
4,0-4,0:      DEDENT       ''
4,0-4,9:      NAME          'say_hello'
4,9-4,10:     LPAR         '('
4,10-4,11:    RPAR         ')'
4,11-4,12:    NEWLINE      '\n'
5,0-5,0:      ENDMARKER    ''
```

Ejemplo de tokenización de un fichero programáticamente, leyendo cadenas unicode en lugar de *bytes* con `generate_tokens()`:

```
import tokenize

with tokenize.open('hello.py') as f:
    tokens = tokenize.generate_tokens(f.readline)
    for token in tokens:
```

(continué en la próxima página)

(proviene de la página anterior)

```
print(token)
```

O leyendo *bytes* directamente con `tokenize()`:

```
import tokenize

with open('hello.py', 'rb') as f:
    tokens = tokenize.tokenize(f.readline)
    for token in tokens:
        print(token)
```

32.8 tabnanny — Detección de indentación ambigua

Código fuente: [Lib/tabnanny.py](#)

Por el momento, este módulo está pensado para ser llamado como un script. Sin embargo, es posible importarlo en un IDE y usar la función `check()` que se describe a continuación.

Nota: Es probable que la API proporcionada por este módulo cambie en versiones futuras; dichos cambios pueden no ser compatibles con versiones anteriores.

`tabnanny.check(file_or_dir)`

Si `file_or_dir` es un directorio y no un enlace simbólico, desciende recursivamente en el árbol de directorios nombrado por `file_or_dir`, verificando todos los archivos `.py` al mismo tiempo. Si `file_or_dir` es un archivo fuente normal de Python, se comprueba si hay problemas relacionados con los espacios en blanco. Los mensajes de diagnóstico se escriben en la salida estándar mediante la función `print()`.

`tabnanny.verbose`

Marcador que indica si se deben imprimir mensajes detallados. Esto se incrementa con la opción `-v` si se llama como un script.

`tabnanny.filename_only`

Marcador que indica si se deben imprimir solo los nombres de archivo de los archivos que contienen problemas relacionados con los espacios en blanco. Esto se establece como verdadero con la opción `-q` si se llama como un script.

exception `tabnanny.NannyNag`

Invocada por `process_tokens()` si detecta una indentación ambigua. Capturada y gestionada en `check()`.

`tabnanny.process_tokens(tokens)`

Esta función es utilizada por `check()` para procesar los tokens generados por el módulo `tokenize`.

Ver también:

Módulo `tokenize` Escáner léxico para código fuente Python.

32.9 `pyclbr` — Soporte para navegador de módulos Python

Código fuente: [Lib/pyclbr.py](#)

El módulo `pyclbr` proporciona información limitada sobre las funciones, clases y métodos definidos en un módulo de Python. La información es suficiente para implementar un navegador de módulos. La información se extrae del código fuente de Python en lugar de importar el módulo, por lo que este módulo es seguro de usar con código que no es de confianza. Esta restricción hace que sea imposible utilizar este módulo con módulos no implementados en Python, incluidos todos los módulos de extensión estándar y opcionales.

`pyclbr.readmodule` (*module*, *path=None*)

Retorna un diccionario que asigna nombres de clase a nivel de módulo con descriptores de clase. Si es posible, se incluyen descriptores para las clases base importadas. El parámetro *module* es una cadena con el nombre del módulo que se va a leer; puede ser el nombre de un módulo dentro de un paquete. Si se indica, *path* es una secuencia de rutas de directorios antepuesto a `sys.path`, que se utiliza para localizar el código fuente del módulo.

Esta función es la interfaz original y sólo se mantiene por compatibilidad. Retorna una versión filtrada de lo siguiente.

`pyclbr.readmodule_ex` (*module*, *path=None*)

Retorna un árbol basado en diccionarios que contiene un descriptor de función o clase para cada función y clase definida en el módulo con una instrucción `def` o `class`. El diccionario retornado asigna nombres de clase y función a nivel de módulo con sus descriptores. Los objetos anidados se introducen en el diccionario hijo de su elemento padre. Al igual que con `readmodule`, *module* nombra el módulo que se va a leer y *path* se antepone a `sys.path`. Si el módulo que se lee es un paquete, el diccionario retornado tiene una clave `'__path__'` cuyo valor es una lista que contiene la ruta del paquete.

Nuevo en la versión 3.7: Descriptores para definiciones anidadas. Se accede a ellos a través del nuevo atributo *children*. Cada uno tiene un nuevo atributo *parent*.

Los descriptores retornados por estas funciones son instancias de las clases `Function` y `Class`. No se espera que los usuarios creen instancias de estas clases.

32.9.1 Objetos `Function`

Las instancias de la clase `Function` describen funciones definidas por instrucciones `def`. Tienen los siguientes atributos:

`Function.file`

Nombre del archivo en el cual la función está definida.

`Function.module`

El nombre del módulo que define la función descrita.

`Function.name`

El nombre de la función.

`Function.lineno`

El número de línea en el archivo donde inicia la definición.

`Function.parent`

Para funciones en el nivel más alto, `None`. Para funciones anidadas, el padre.

Nuevo en la versión 3.7.

`Function.children`

Un diccionario asignando nombres con descriptores para las clases y funciones anidadas.

Nuevo en la versión 3.7.

32.9.2 Objetos Class

Las instancias de la clase `Class` describen clases definidas por instrucciones `class`. Tienen los mismos atributos que `Functions` y dos más.

`Class.file`

Nombre del archivo en el que la clase está definida.

`Class.module`

Nombre del módulo que define la clase descrita.

`Class.name`

El nombre de la clase.

`Class.lineno`

El número de línea en el archivo donde inicia la definición.

`Class.parent`

Para clases en el nivel más alto, `None`. Para clases anidadas, el padre.

Nuevo en la versión 3.7.

`Class.children`

Un diccionario asignando nombres con descriptores para las clases y funciones anidadas.

Nuevo en la versión 3.7.

`Class.super`

Una lista de objetos `Class` que describen las clases base inmediatas de la clase que se está describiendo. Las clases que se denominan superclases pero que no son detectables por `readmodule_ex()` se enumeran como una cadena con el nombre de clase en lugar de objetos `Class`.

`Class.methods`

Un diccionario asignando los nombres de los métodos a sus números de línea. Esto se puede derivar del reciente diccionario `children`, pero permanece por compatibilidad.

32.10 `py_compile` — Compila archivos fuente Python

Código fuente `Lib/py_compile.py`

El módulo `py_compile` provee una función para generar un archivo de código de bytes de un archivo fuente, y otra función usada cuando el módulo archivo fuente es invocado como un script.

Aunque no es necesario seguidamente, esta función puede ser útil cuando se instalan módulos para uso compartido, especialmente si algunos de los usuarios pueden no tener permisos para escribir el archivo caché de bytes en el directorio conteniendo el código fuente.

exception `py_compile.PyCompileError`

Cuando un error ocurre mientras se intenta compilar el archivo, se lanza una excepción.

`py_compile.compile(file, cfile=None, dfile=None, doraise=False, optimize=-1, invalidation_mode=PyInvalidationMode.TIMESTAMP, quiet=0)`

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 the [PEP 3147/PEP 488](#) path, ending in `.pyc`. For example, if `file` is `/foo/bar/baz.py` `cfile` will default to `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. 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. This function returns the path to byte-compiled file, i.e. whatever `cfile` value was used.

Los argumentos `doraise` y `quiet` determinan cómo los errores son gestionados mientras se compila el archivo. Si `quiet` es 0 o 1, y `doraise` es falso, la conducta por defecto es habilitada: un error cadena de caracteres es

escrito a `sys.stderr`, y la función retorna `None` en vez de una ruta. Si *doraise* es verdadero, se lanzará un `PyCompileError`. Sin embargo si *quiet* es 2, ningún mensaje es escrito y *doraise* no tiene efecto.

Si la ruta que *cfile* se convierte (sea especificada explícitamente o computada) es un enlace simbólico o archivo no regular, se lanzará `FileExistsError`. Esto es para actuar como una advertencia que la importación convertirá esas rutas en archivos regulares si ésta tiene permitido escribir archivos compilados en bytes a esas rutas. Este es un efecto secundario de importar usando el renombramiento de archivos para colocar el archivo de bytes compilado final en el lugar para prevenir problemas de escritura en archivos simultáneos.

optimize controla el nivel de optimización y si es pasado a la función construida `compile()`. El predeterminado de `-1` selecciona el nivel de optimización del intérprete actual.

invalidation_mode debería ser un miembro del enum `PycInvalidationMode` y controla cómo el caché de código de bytes generado es invalidado en el tiempo de ejecución. El predeterminado es `PycInvalidationMode.CHECKED_HASH` si la variable de entorno `SOURCE_DATE_EPOCH` está establecida, de otra manera el predeterminado es `PycInvalidationMode.TIMESTAMP`.

Distinto en la versión 3.2: Cambiado el valor por defecto de *cfile* para que cumpla [PEP 3147](#). El por defecto anterior era `file + 'c'` ('o' si la optimización estaba habilitada). También agregado el parámetro *optimize*.

Distinto en la versión 3.4: Se cambió el código para usar `importlib` para la escritura del archivo almacenado de código de bytes. Esto significa que la semántica de la creación/escritura del archivo ahora coincide con lo que `importlib` hace, por ejemplo permisos, semántica de escribir-y-mover, etc. Además se agregó la consideración de que `FileExistsError` es lanzado si *cfile* es un enlace simbólico o un archivo no regular.

Distinto en la versión 3.7: El parámetro *invalidation_mode* fue agregado como se especificó en [PEP 552](#). Si la variable de entorno `SOURCE_DATE_EPOCH` se establece, *invalidation_mode* será forzada a `PycInvalidationMode.CHECKED_HASH`.

Distinto en la versión 3.7.2: La variable de entorno `SOURCE_DATE_EPOCH` ya no sobrescribe el valor del argumento *invalidation_mode*, y en vez de eso determina su valor por defecto.

Distinto en la versión 3.8: El parámetro *quiet* fue agregado.

class `py_compile.PycInvalidationMode`

Una enumeración de métodos posibles que el intérprete puede usar para determinar si un archivo de bytes está actualizado con un archivo fuente. El archivo `.pyc` indica el modo invalidación deseado en su encabezado. Ver `pyc-invalidation` para más información en cómo Python invalida archivos `.pyc` en tiempo de ejecución.

Nuevo en la versión 3.7.

TIMESTAMP

El archivo `.pyc` incluye una marca de tiempo y tamaño del archivo fuente, el cual Python comparará contra los metadatos del archivo fuente durante el tiempo de ejecución para determinar si el archivo `.pyc` necesita ser regenerado.

CHECKED_HASH

El archivo `.pyc` incluye un hash del contenido del archivo fuente, el cual Python comparará contra la fuente durante el tiempo de ejecución para determinar si el archivo `.pyc` necesita ser regenerado.

UNCHECKED_HASH

Como `CHECKED_HASH`, el archivo `.pyc` incluye un has del contenido del archivo fuente. Sin embargo, Python asumirá durante el tiempo de ejecución que el archivo `.pyc` está actualizado y no validará el `.pyc` contra el archivo fuente en lo absoluto.

Esta opción es útil cuando los `.pucs` se mantienen actualizados al día por algún sistema externo a Python como un sistema de compilación.

`py_compile.main(args=None)`

Compila varios archivos fuente. Los archivos mencionados en *args* (o en la línea de comandos si *args* es `None`) son compilados y el código de bytes resultante es almacenado de la manera normal. Esta función no busca una estructura de directorio para localizar archivos fuente; éste sólo compila archivos nombrados explícitamente. Si `'-'` es el único parámetro en *args*, la lista de archivos es tomada de una entrada estándar.

Distinto en la versión 3.2: Agregado soporte para `'-'`.

Cuando este módulo se ejecuta como un script, el `main()` es usado para compilar todos los archivos llamados en la línea de comandos. El estado de salida es no cero si uno de los archivos no se pudo compilar.

Ver también:

Módulo `compileall` Utilidades para compilar todos los archivos fuente Python en el árbol del directorio.

32.11 `compileall` — Bibliotecas de Python de compilación de bytes

Código fuente: [Lib/compileall.py](#)

Este módulo proporciona algunas funciones de utilidad para admitir la instalación de bibliotecas Python. Estas funciones compilan archivos fuente de Python en un árbol de directorios. Este módulo se puede usar para crear los archivos de código de bytes almacenados en caché en el momento de la instalación de la biblioteca, que los hace disponibles para su uso incluso por usuarios que no tienen permiso de escritura en los directorios de la biblioteca.

32.11.1 Uso de la línea de comandos

Este módulo puede funcionar como un *script* (usando `python -m compileall`) para compilar fuentes de Python.

directory ...

file ...

Los argumentos posicionales son archivos para compilar o directorios que contienen archivos fuente, recorridos recursivamente. Si no se proporciona ningún argumento, se comporta como si la línea de comando fuera `-l <directories from sys.path>`.

-l

No se recurre en subdirectorios, solo compila archivos de código fuente contenidos en directorios nombrados o implícitos.

-f

Forzar la reconstrucción incluso si las marcas de tiempo están actualizadas.

-q

No imprimir la lista de archivos compilados. Si se pasa una vez, los mensajes de error se imprimirán. Si se pasa dos veces, (`-qq`), se suprime toda la salida.

-d destdir

Directorio antepuesto a la ruta de cada archivo que está siendo compilado. Esto aparecerá en las devoluciones de tiempo de compilación, y también se compila en el archivo de código de bytes, donde se usará en las devoluciones de seguimiento y otros mensajes en casos donde el archivo fuente no existe al momento en que el archivo de código de bytes se ejecuta.

-x regex

regex se usa para buscar la ruta completa a cada archivo considerado para compilación, y si la regex produce una coincidencia, se omite el archivo.

-i list

Leer el archivo `list` y cada línea que contiene la lista de archivos y directorios a compilar. Si `list` es `-`, leer líneas desde `stdin`.

-b

Escribir los archivos de código de byte en las locaciones y nombres de herencia, que pueden sobrescribir los archivos de código de bytes creado para otra versión de Python. El comportamiento por defecto es escribir archivos en sus locaciones y nombres **PEP 3147**, lo cual permite que archivos de código de byte de versiones múltiples de Python coexistan.

-r
 Controlar el nivel máximo de recurrencia para subdirectorios. Si esto está dado, la opción `-l` no se tendrá en cuenta. `python -m compileall <directory> -r 0` es equivalente a `python -m compileall <directory> -l`.

-j N
 Usar *workers N* para compilar los archivos dentro del directorio dado. Si `0` se usa, entonces se usa el resultado de `os.cpu_count()`.

--invalidation-mode [timestamp|checked-hash|unchecked-hash]
 Controlar cómo los archivos de código de byte generados se invalidan al momento de ejecución. El valor `timestamp` significa que se generarán los archivos `.pyc` con la marca de tiempo fuente y el tamaño insertados. Los valores `checked-hash` y `unchecked-hash` generan *pycs* basados en hash. Los *pycs* basados en hash del archivo insertan un hash de los contenidos del archivo fuente, en lugar de una marca de tiempo. Véase `pyc-invalidation` para mayor información sobre cómo Python valida archivos de cache de código de bytes. El valor por defecto es `timestamp` si la variable de entorno `SOURCE_DATE_EPOCH` no está definida, y `checked-hash` si la variable de entorno `SOURCE_DATE_EPOCH` está definida.

Distinto en la versión 3.2: Se agregaron las opciones `-i`, `-b` y `-h`.

Distinto en la versión 3.5: Se agregaron las opciones `-j`, `-r`, and `-qq`. La opción `-q` se cambió a un valor multinivel. `-b` siempre producirá un archivo de código de byte que termina en `.pyc`, nunca `.pyo`.

Distinto en la versión 3.7: Se agregó la opción `--invalidation-mode`.

No hay opción de línea de comando para controlar el nivel de optimización que usa la función `compile()` porque el intérprete de Python en sí mismo ya proporciona la opción: `python -O -m compileall`.

De manera similar, la función `compile()` respeta la configuración `sys.pycache_prefix`. El cache de código de byte generado sólo será útil si `compile()` se ejecuta con el mismo `sys.pycache_prefix` (si es que existe alguno) que se utilizará en el momento de ejecución.

32.11.2 Funciones públicas

`compileall.compile_dir(dir, maxlevels=10, ddir=None, force=False, rx=None, quiet=0, legacy=False, optimize=-1, workers=1, invalidation_mode=None)`

Descender recursivamente el árbol de directorio invocado por *dir*, compilando todos los archivos `.py` que encuentra en el camino. Devolver un valor verdadero si todos los archivos se compilan exitosamente, y un valor falso en el caso contrario.

El parámetro *maxlevels* se usar para limitar la profundidad de la recursión; toma como valor predeterminado 10.

Si *ddir* está dado, se antepone a la ruta de cada archivo que se compila para usar en los rastreos de tiempo de compilación, y también se compilar en el archivo código de byte, donde se usarán en trazas y otros mensajes en casos donde el archivo fuente no existe en el momento cuando el archivo de código de byte se ejecuta.

Si *force* es verdadero, los módulos se re-compilan aun cuando las marcas de tiempo están actualizadas.

Si *rx* está dado, su método de búsqueda en la ruta completa de cada archivo considerado para compilación, y si devuelve un valor verdadero, el archivo se saltea.

Si *quiet* es `False` o 0 (el valor predeterminado), los nombres de archivo y otra información se imprimen en salida estándar. Si se configura en 1, solo se imprimen los errores. Si se configura en 2, se suprime toda la salida.

Si *legacy* es verdadero, los archivos de código de byte se escriben en sus locaciones y nombres de herencia, que pueden sobrescribir los archivos de código de byte creado por otra versión de Python. El comportamiento por defecto es escribir archivos en sus locaciones y nombres **PEP 3147**, lo cual permite que archivos de código de byte de múltiples versiones de Python coexistan.

optimize especifica el nivel de optimización para el compilador. Se pasa a una función incorporada `compile()`.

El argumento *workers* especifica cuántos workers se usan para compilar archivos en paralelo. El comportamiento por defecto es no usar múltiples workers. Si la plataforma no puede usar workers múltiples y el argumento *workers* está dado, la compilación secuencial se usará como *fallback*. Si *workers* es 0, el número de núcleos se usa en el sistema. Si *workers* es menor que 0, se genera un *ValueError*.

invalidation_mode debería ser un miembro de la enumeración `py_compile.PycInvalidationMode` y controla cómo se invalidan los pycs generados en el momento de ejecución.

Distinto en la versión 3.2: Se agregó el parámetro *legacy* y *optimize*.

Distinto en la versión 3.5: Se agregó el parámetro *workers*.

Distinto en la versión 3.5: El parámetro *quiet* se cambió a un valor multinivel.

Distinto en la versión 3.5: El parámetro *legacy* solo escribe archivos `.pyc`, no archivos `.pyo`, no import cuál sea el valor de *optimize*.

Distinto en la versión 3.6: Acepta un *path-like object*.

Distinto en la versión 3.7: Se agregó el parámetro *invalidation_mode*.

Distinto en la versión 3.7.2: El valor predeterminado del parámetro *invalidation_mode* se actualiza a *None*.

Distinto en la versión 3.8: Configurar *workers* a 0 ahora elige el número óptimo de núcleos.

`compileall.compile_file`(*fullname*, *ddir*=None, *force*=False, *rx*=None, *quiet*=0, *legacy*=False, *optimize*=-1, *invalidation_mode*=None)

Compilar el archivo con ruta *fullname*. Retorna un valor verdadero si el archivo se compila exitosamente, y uno falso en el caso contrario.

Si *ddir* está dado, se antepone a la ruta del archivo que está siendo compilado para su uso en las trazas de tiempo de compilación, y también se compilar en el archivo de código de bytes, donde será utilizado en trazas y otros mensajes en casos donde el archivo fuente no existe en el momento en que el archivo de código de bytes es ejecutado.

Si *rx* está dado, su método de búsqueda recibe el nombre de ruta completo hacia el archivo que está siendo compilado, y si devuelve un valor verdadero, el archivo no se compila y retorna *True*.

Si *quiet* es *False* o 0 (el valor predeterminado), los nombres de archivo y otra información se imprimen en salida estándar. Si se configura en 1, solo se imprimen los errores. Si se configura en 2, se suprime toda la salida.

Si *legacy* es verdadero, los archivos de código de byte se escriben en sus locaciones y nombres de herencia, que pueden sobrescribir los archivos de código de byte creado por otra versión de Python. El comportamiento por defecto es escribir archivos en sus locaciones y nombres [PEP 3147](#), lo cual permite que archivos de código de byte de múltiples versiones de Python coexistan.

optimize especifica el nivel de optimización para el compilador. Se pasa a una función incorporada `compile()`.

invalidation_mode debería ser un miembro de la enumeración `py_compile.PycInvalidationMode` y controla cómo se invalidan los pycs generados en el momento de ejecución.

Nuevo en la versión 3.2.

Distinto en la versión 3.5: El parámetro *quiet* se cambió a un valor multinivel.

Distinto en la versión 3.5: El parámetro *legacy* solo escribe archivos `.pyc`, no archivos `.pyo`, no import cuál sea el valor de *optimize*.

Distinto en la versión 3.7: Se agregó el parámetro *invalidation_mode*.

Distinto en la versión 3.7.2: El valor predeterminado del parámetro *invalidation_mode* se actualiza a *None*.

`compileall.compile_path`(*skip_curdir*=True, *maxlevels*=0, *force*=False, *quiet*=0, *legacy*=False, *optimize*=-1, *invalidation_mode*=None)

Compila en bytes todos los archivos `.py` a lo largo de `sys.path`. Retorna un valor verdadero si todos los archivos se compilan exitosamente, y uno falso en el caso contrario.

Si `skip_curdire` es verdadero (el valor predeterminado), el directorio actual no está incluido en la búsqueda. Todos los otros parámetros se pasan a la función `compile_dir()`. Nótese que, al contrario de las otras funciones de compilación, `maxlevels` tomar 0 como valor predeterminado.

Distinto en la versión 3.2: Se agregó el parámetro `legacy` y `optimize`.

Distinto en la versión 3.5: El parámetro `quiet` se cambió a un valor multinivel.

Distinto en la versión 3.5: El parámetro `legacy` solo escribe archivos `.pyc`, no archivos `.pyo`, no importa cuál sea el valor de `optimize`.

Distinto en la versión 3.7: Se agregó el parámetro `invalidation_mode`.

Distinto en la versión 3.7.2: El valor predeterminado del parámetro `invalidation_mode` se actualiza a `None`.

Para forzar la re-compilación de los archivos `.py` en el subdirectorio `Lib/` y todos sus subdirectorios:

```
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)

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

Ver también:

Module `py_compile` Byte-compile a single source file.

32.12 dis — Desensamblador para bytecode de Python

Código fuente: `Lib/dis.py`

El módulo `dis` admite el análisis de CPython *bytecode* al desarmarlo. El bytecode de CPython que este módulo toma como entrada se define en el archivo `Include/opcode.h` y lo utilizan el compilador y el intérprete.

CPython implementation detail: Bytecode es un detalle de implementación del intérprete CPython. No se garantiza que el bytecode no se agregará, eliminará ni cambiará entre las versiones de Python. El uso de este módulo no debe considerarse para trabajar en diferentes máquinas virtuales Python o versiones de Python.

Distinto en la versión 3.6: Use 2 bytes para cada instrucción. Anteriormente, el número de bytes variaba según la instrucción.

Ejemplo: dada la función `myfunc()`:

```
def myfunc(alist):
    return len(alist)
```

el siguiente comando se puede utilizar para mostrar el desensamblaje de `myfunc()`:

```
>>> dis.dis(myfunc)
2          0 LOAD_GLOBAL              0 (len)
          2 LOAD_FAST                  0 (alist)
          4 CALL_FUNCTION              1
          6 RETURN_VALUE
```

(El «2» es un número de línea).

32.12.1 Análisis de bytecode

Nuevo en la versión 3.4.

La API de análisis de bytecode permite que partes del código Python se envuelvan en un objeto *Bytecode* que proporciona un fácil acceso a los detalles del código compilado.

class `dis.Bytecode` (*x*, *, *first_line=None*, *current_offset=None*)

Analiza el bytecode correspondiente a una función, generador, generador asíncrono, corutina, método, cadena de código fuente o un objeto de código (como lo retorna `compile()`).

Este es un contenedor conveniente para muchas de las funciones enumeradas a continuación, en particular `get_instructions()`, ya que iterar sobre una instancia de *Bytecode* produce las operaciones de bytecode como instancias de *Instruction*.

Si *first_line* no es `None`, indica el número de línea que se debe informar para la primera línea de origen en el código desmontado. De lo contrario, la información de la línea de origen (si la hay) se toma directamente del objeto de código desmontado.

Si *current_offset* no es `None`, se refiere a un desplazamiento de instrucción en el código desmontado. Establecer esto significa `dis()` mostrará un marcador de «instrucción actual» contra el código de operación especificado.

classmethod `from_traceback` (*tb*)

Construye una instancia de *Bytecode* a partir del *traceback* dado, estableciendo *current_offset* en la instrucción responsable de la excepción.

codeobj

El objeto de código compilado.

first_line

La primera línea de origen del objeto de código (si está disponible)

dis ()

Retorna una vista formateada de las operaciones de bytecode (lo mismo que impreso por `dis.dis()`, pero retornado como una cadena de caracteres multilínea).

info ()

Retorna una cadena de caracteres multilínea formateada con información detallada sobre el objeto de código, como `code_info()`.

Distinto en la versión 3.7: Esto ahora puede manejar objetos generadores asíncronos y de corutinas.

Ejemplo:

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
LOAD_GLOBAL
LOAD_FAST
CALL_FUNCTION
RETURN_VALUE
```


32.12.2 Funciones de análisis

El módulo `dis` también define las siguientes funciones de análisis que convierten la entrada directamente en la salida deseada. Pueden ser útiles si solo se realiza una sola operación, por lo que el objeto de análisis intermedio no es útil:

`dis.code_info(x)`

Retorna una cadena de caracteres multilínea formateada con información detallada del objeto de código para la función, generador, generador asíncrono, corutina, método, cadena de código fuente u objeto de código suministrados.

Tenga en cuenta que el contenido exacto de las cadenas de información de código depende en gran medida de la implementación y puede cambiar arbitrariamente en las diferentes máquinas virtuales Python o las versiones de Python.

Nuevo en la versión 3.2.

Distinto en la versión 3.7: Esto ahora puede manejar objetos generadores asíncronos y de corutinas.

`dis.show_code(x, *, file=None)`

Imprime información detallada del objeto de código para la función, método, cadena de código fuente u objeto de código suministrado en `file` (o `sys.stdout` si `file` no está especificado).

Esta es una abreviatura conveniente para `print(code_info(x), file=file)`, destinado a la exploración interactiva en el indicador del intérprete (*prompt*).

Nuevo en la versión 3.2.

Distinto en la versión 3.4: Agrega un parámetro `file`.

`dis.dis(x=None, *, file=None, depth=None)`

Desmontar el objeto `x`. `x` puede denotar un módulo, una clase, un método, una función, un generador, un generador asíncrono, una corutina, un objeto de código, una cadena de código fuente o una secuencia de bytes de código de bytes sin procesar. Para un módulo, desmonta todas las funciones. Para una clase, desmonta todos los métodos (incluidos los métodos de clase y estáticos). Para un objeto de código o secuencia de bytecode sin procesar, imprime una línea por instrucción de bytecode. También desmonta recursivamente objetos de código anidados (el código de comprensiones, expresiones generadoras y funciones anidadas, y el código utilizado para construir clases anidadas). Las cadenas de caracteres se compilan primero en objetos de código con la función incorporada `compile()` antes de desmontarse. Si no se proporciona ningún objeto, esta función desmonta el último rastreo.

El desensamblaje se escribe como texto en el argumento `file` proporcionado si se proporciona y, de lo contrario, `sys.stdout`.

La profundidad máxima de recursión está limitada por `depth` a menos que sea `None`. `depth=0` significa que no hay recursión.

Distinto en la versión 3.4: Agrega un parámetro `file`.

Distinto en la versión 3.7: Desensamblaje recursivo implementado y parámetro agregado `depth`.

Distinto en la versión 3.7: Esto ahora puede manejar objetos generadores asíncronos y de corutinas.

`dis.distb(tb=None, *, file=None)`

Desmonta la función de inicio de pila de un rastreo, utilizando el último rastreo si no se pasó ninguno. Se indica la instrucción que causa la excepción.

El desensamblaje se escribe como texto en el argumento `file` proporcionado si se proporciona y, de lo contrario, `sys.stdout`.

Distinto en la versión 3.4: Agrega un parámetro `file`.

`dis.disassemble(code, lasti=-1, *, file=None)`

`dis.disco(code, lasti=-1, *, file=None)`

Desmonta un objeto de código, que indica la última instrucción si se proporcionó `lasti`. La salida se divide en las siguientes columnas:

1. el número de línea, para la primera instrucción de cada línea

2. la instrucción actual, indicada como `-->`,
3. una instrucción etiquetada, indicada con `>>`,
4. la dirección de la instrucción,
5. el nombre del código de operación,
6. parámetros de operación, y
7. interpretación de los parámetros entre paréntesis.

La interpretación de parámetros reconoce nombres de variables locales y globales, valores constantes, objetivos de ramificación y operadores de comparación.

El desensamblaje se escribe como texto en el argumento *file* proporcionado si se proporciona y, de lo contrario, `sys.stdout`.

Distinto en la versión 3.4: Agrega un parámetro *file*.

`dis.get_instructions(x, *, first_line=None)`

Retorna un iterador sobre las instrucciones en la función, método, cadena de código fuente u objeto de código suministrado.

El iterador genera una serie de tuplas con nombre *Instruction* que dan los detalles de cada operación en el código suministrado.

Si *first_line* no es `None`, indica el número de línea que se debe informar para la primera línea de origen en el código desmontado. De lo contrario, la información de la línea de origen (si la hay) se toma directamente del objeto de código desmontado.

Nuevo en la versión 3.4.

`dis.findlinestarts(code)`

Esta función de generador utiliza los atributos `co_firstlineno` y `co_lnotab` del objeto de código *code* para encontrar los desplazamientos que son comienzos de líneas en el código fuente. Se generan como pares (`offset`, `lineno`). Ver [Objects/lnotab_notes.txt](#) para el formato `co_lnotab` y cómo decodificarlo.

Distinto en la versión 3.6: Los números de línea pueden estar disminuyendo. Antes, siempre estaban aumentando.

`dis.findlabels(code)`

Detecta todos los desplazamientos en la cadena de caracteres de código de byte compilada *code* que son objetivos de salto y retorna una lista de estos desplazamientos.

`dis.stack_effect(opcode, oparg=None, *, jump=None)`

Calcula el efecto de pila de *opcode* con el argumento *oparg*.

Si el código tiene un objetivo de salto y *jump* es `True`, *stack_effect()* retornará el efecto de pila del salto. Si *jump* es `False`, retornará el efecto de acumulación de no saltar. Y si *jump* es `None` (predeterminado), retornará el efecto de acumulación máxima de ambos casos.

Nuevo en la versión 3.4.

Distinto en la versión 3.8: Agrega un parámetro *jump*.

32.12.3 Instrucciones bytecode de Python

La función *get_instructions()* y clase *Bytecode* proporcionan detalles de las instrucciones bytecode como instancias *Instruction*:

class `dis.Instruction`

Detalles para una operación de bytecode

opcode

código numérico para la operación, correspondiente a los valores del opcode listados a continuación y los valores de bytecode en *Colecciones opcode*.

opname

nombre legible por humanos para la operación

arg

argumento numérico para la operación (si existe), de lo contrario `None`

argval

valor *arg* resuelto (si se conoce), de lo contrario igual que *arg*

argrepr

descripción legible por humanos del argumento de operación

offset

índice de inicio de operación dentro de la secuencia de bytecode

starts_line

línea iniciada por este código de operación (si existe), de lo contrario `None`

is_jump_target

`True` si otro código salta aquí, de lo contrario, `False`

Nuevo en la versión 3.4.

El compilador de Python actualmente genera las siguientes instrucciones de bytecode.

Instrucciones generales**NOP**

Código que hace nada. Utilizado como marcador de posición por el optimizador de código de bytes.

POP_TOP

Elimina el elemento de la parte superior de la pila (TOS).

ROT_TWO

Intercambia los dos elementos más apilados.

ROT_THREE

Levanta el segundo y tercer elemento de la pila una posición hacia arriba, mueve el elemento superior hacia abajo a la posición tres.

ROT_FOUR

Lifts second, third and fourth stack items one position up, moves top down to position four.

Nuevo en la versión 3.8.

DUP_TOP

Duplica la referencia en la parte superior de la pila.

Nuevo en la versión 3.2.

DUP_TOP_TWO

Duplica las dos referencias en la parte superior de la pila, dejándolas en el mismo orden.

Nuevo en la versión 3.2.

Operaciones unarias

Las operaciones unarias toman la parte superior de la pila, aplican la operación y retornan el resultado a la pila.

UNARY_POSITIVE

Implementa `TOS = +TOS`.

UNARY_NEGATIVE

Implementa `TOS = -TOS`.

UNARY_NOT

Implementa `TOS = not TOS`.

UNARY_INVERT

Implementa `TOS = ~TOS`.

GET_ITER

Implementa `TOS = iter(TOS)`.

GET_YIELD_FROM_ITER

Si `TOS` es un *iterador generador* o un objeto *corutina* se deja como está. De lo contrario, implementa `TOS = iter(TOS)`.

Nuevo en la versión 3.5.

Operaciones binarias

Las operaciones binarias eliminan el elemento superior de la pila (`TOS`) y el segundo elemento de la pila superior (`TOS1`) de la pila. Realizan la operación y retornan el resultado a la pila.

BINARY_POWER

Implementa `TOS = TOS1 ** TOS`.

BINARY_MULTIPLY

Implementa `TOS = TOS1 * TOS`.

BINARY_MATRIX_MULTIPLY

Implementa `TOS = TOS1 @ TOS`.

Nuevo en la versión 3.5.

BINARY_FLOOR_DIVIDE

Implementa `TOS = TOS1 // TOS`.

BINARY_TRUE_DIVIDE

Implementa `TOS = TOS1 / TOS`.

BINARY_MODULO

Implementa `TOS = TOS1 % TOS`.

BINARY_ADD

Implementa `TOS = TOS1 + TOS`.

BINARY_SUBTRACT

Implementa `TOS = TOS1 - TOS`.

BINARY_SUBSCR

Implementa `TOS = TOS1[TOS]`.

BINARY_LSHIFT

Implementa `TOS = TOS1 << TOS`.

BINARY_RSHIFT

Implementa `TOS = TOS1 >> TOS`.

BINARY_AND

Implementa `TOS = TOS1 & TOS`.

BINARY_XOR

Implementa `TOS = TOS1 ^ TOS`.

BINARY_OR

Implementa `TOS = TOS1 | TOS`.

Operaciones en su lugar

Las operaciones en el lugar son como operaciones binarias, ya que eliminan `TOS` y `TOS1`, y retornan el resultado a la pila, pero la operación se realiza en el lugar cuando `TOS1` lo admite, y el `TOS` resultante puede ser (pero no tiene ser) el `TOS1` original.

INPLACE_POWER

Implementa en su lugar `TOS = TOS1 ** TOS`.

INPLACE_MULTIPLY

Implementa en su lugar `TOS = TOS1 * TOS`.

INPLACE_MATRIX_MULTIPLY

Implementa en su lugar `TOS = TOS1 @ TOS`.

Nuevo en la versión 3.5.

INPLACE_FLOOR_DIVIDE

Implementa en su lugar `TOS = TOS1 // TOS`.

INPLACE_TRUE_DIVIDE

Implementa en su lugar `TOS = TOS1 / TOS`.

INPLACE_MODULO

Implementa en su lugar `TOS = TOS1 % TOS`.

INPLACE_ADD

Implementa en su lugar `TOS = TOS1 + TOS`.

INPLACE_SUBTRACT

Implementa en su lugar `TOS = TOS1 - TOS`.

INPLACE_LSHIFT

Implementa en su lugar `TOS = TOS1 << TOS`.

INPLACE_RSHIFT

Implementa en su lugar `TOS = TOS1 >> TOS`.

INPLACE_AND

Implementa en su lugar `TOS = TOS1 & TOS`.

INPLACE_XOR

Implementa en su lugar `TOS = TOS1 ^ TOS`.

INPLACE_OR

Implementa en su lugar `TOS = TOS1 | TOS`.

STORE_SUBSCR

Implementa `TOS1[TOS] = TOS2`.

DELETE_SUBSCR

Implementa `del TOS1[TOS]`.

Opcodes de corutinas**GET_AWAITABLE**

Implementa `TOS = get_awaitable(TOS)`, donde `get_awaitable(o)` retorna `o` si `o` es un objeto de corutina o un objeto generador con el indicador `CO_ITERABLE_COROUTINE`, o resuelve `o.__await__`.

Nuevo en la versión 3.5.

GET_AITER

Implementa `TOS = TOS.__aiter__()`.

Nuevo en la versión 3.5.

Distinto en la versión 3.7: Ya no se admite el retorno de objetos *awaitable* de `__aiter__`.

GET_ANEXT

Implementa `PUSH(get_awaitable(TOS.__anext__()))`. Consulte `GET_AWAITABLE` para obtener detalles sobre `get_awaitable`.

Nuevo en la versión 3.5.

END_ASYNC_FOR

Termina un bucle `async for`. Maneja una excepción planteada cuando se espera un próximo elemento. Si `TOS` es `StopAsyncIteration` desapila 7 valores de la pila y restaura el estado de excepción utilizando los tres últimos. De lo contrario, vuelva a lanzar la excepción utilizando los tres valores de la pila. Se elimina un bloque de controlador de excepción de la pila de bloques.

Nuevo en la versión 3.8.

BEFORE_ASYNC_WITH

Resuelve `__aenter__` y `__aexit__` del objeto en la parte superior de la pila. Apila `__aexit__` y el resultado de `__aenter__()` a la pila.

Nuevo en la versión 3.5.

SETUP_ASYNC_WITH

Crea un nuevo objeto marco.

Nuevo en la versión 3.5.

Opcodes misceláneos

PRINT_EXPR

Implementa la declaración de expresión para el modo interactivo. TOS se elimina de la pila y se imprime. En modo no interactivo, una declaración de expresión termina con `POP_TOP`.

SET_ADD (i)

Llama a `set.add(TOS1[-i], TOS)`. Se utiliza para implementar comprensiones de conjuntos.

LIST_APPEND (i)

Calls `list.append(TOS1[-i], TOS)`. Used to implement list comprehensions.

MAP_ADD (i)

Llama a `dict.__setitem__(TOS1[-i], TOS1, TOS)`. Se utiliza para implementar comprensiones de diccionarios.

Nuevo en la versión 3.1.

Distinto en la versión 3.8: El valor del mapa es TOS y la clave del mapa es TOS1. Antes, esos fueron revertidos.

Para todas las instrucciones `SET_ADD`, `LIST_APPEND` y `MAP_ADD`, mientras el valor agregado o el par clave/valor aparece, el objeto contenedor permanece en la pila para que quede disponible para futuras iteraciones del bucle.

RETURN_VALUE

Retorna con TOS a quien llama la función.

YIELD_VALUE

Desapila TOS y lo genera (*yield*) de un *generator*.

YIELD_FROM

Desapila TOS y delega en él como un subiterador de un *generator*.

Nuevo en la versión 3.3.

SETUP_ANNOTATIONS

Comprueba si `__anotaciones__` está definido en `locals()`, si no está configurado como un dict vacío. Este código de operación solo se emite si el cuerpo de una clase o módulo contiene *anotaciones de variables* estáticamente.

Nuevo en la versión 3.6.

IMPORT_STAR

Carga todos los símbolos que no comienzan con `'_'` directamente desde el TOS del módulo al espacio de nombres local. El módulo se desapila después de cargar todos los nombres. Este opcode implementa `from module import *`.

POP_BLOCK

Elimina un bloque de la pila de bloques. Por cuadro, hay una pila de bloques, que denota declaraciones `try`, y tal.

POP_EXCEPT

Elimina un bloque de la pila de bloques. El bloque desapilado debe ser un bloque de controlador de excepción, como se crea implícitamente al ingresar un controlador de excepción. Además de desapilar valores extraños de la pila de cuadros, los últimos tres valores desapilados se utilizan para restaurar el estado de excepción.

POP_FINALLY (preserve_tos)

Limpia la pila de valores y la pila de bloques. Si *preserve_tos* no es 0, primero se desapila TOS de la pila y se apila a la pila después de realizar otras operaciones de pila:

- Si TOS es `NULL` o un número entero (apilado por `BEGIN_FINALLY` o `CALL_FINALLY`) se desapila de la pila.
- Si TOS es un tipo de excepción (se apila cuando se ha lanzado una excepción) se desapilan 6 valores de la pila, los últimos tres valores se usan para restaurar el estado de excepción. Se elimina un bloque de controlador de excepción de la pila de bloques.

Es similar a `END_FINALLY`, pero no cambia el contador de bytecode ni lanza una excepción. Se usa para implementar `break`, `continue` y `return` en el bloque `finally`.

Nuevo en la versión 3.8.

BEGIN_FINALLY

Apila `NULL` en la pila para usarlo en `END_FINALLY`, `POP_FINALLY`, `WITH_CLEANUP_START` y `WITH_CLEANUP_FINISH`. Inicia el bloque `finally`.

Nuevo en la versión 3.8.

END_FINALLY

Termina una cláusula `finally`. El intérprete recuerda si la excepción se debe volver a ser lanzada o si se debe continuar la ejecución según el valor de TOS.

- Si TOS es `NULL` (apilado por `BEGIN_FINALLY`) continúa con la siguiente instrucción. TOS se desapila.
- Si TOS es un entero (apilado por `CALL_FINALLY`), establece el contador de bytecode en TOS. TOS se desapila.
- Si TOS es un tipo de excepción (se apila cuando se ha lanzado una excepción) se desapilan 6 valores de la pila, los primeros tres valores desapilados se utilizan para volver a lanzar la excepción y los últimos tres valores desapilados se utilizan para restaurar el estado de excepción. Se elimina un bloque de controlador de excepción de la pila de bloques.

LOAD_BUILD_CLASS

Apila `builtins.__build_class__()` en la pila. Más tarde se llama por `CALL_FUNCTION` para construir una clase.

SETUP_WITH (*delta*)

Este opcode realiza varias operaciones antes de que comience un bloque `with`. Primero, carga `__exit__()` desde el administrador de contexto y lo apila a la pila para su uso posterior por `WITH_CLEANUP_START`. Entonces, `__enter__()` se llama, y finalmente se bloquea un bloque que apunta a *delta*. Finalmente, el resultado de llamar al método `__enter__()` se apila en la pila. El siguiente opcode lo ignorará (`POP_TOP`), o lo almacenará en (una) variable (s) (`STORE_FAST`, `STORE_NAME`, o `UNPACK_SEQUENCE`).

Nuevo en la versión 3.2.

WITH_CLEANUP_START

Comienza a limpiar la pila cuando sale un bloque de instrucciones `with`.

En la parte superior de la pila están `NULL` (apilado por `BEGIN_FINALLY`) o 6 valores apilados si se ha lanzado una excepción en el bloque `with`. A continuación se muestra el gestor de contexto `__exit__()` o un método ligado `__aexit__()`.

Si TOS es `NULL`, llama a `SECOND(None, None, None)`, elimina la función de la pila, deja TOS y apila `None` a la pila. De lo contrario, llama `SEVENTH(TOP, SECOND, THIRD)`, desplaza los 3 valores inferiores de la pila hacia abajo, reemplaza el espacio vacío con `NULL` y apila TOS. Finalmente apila el resultado de la llamada.

WITH_CLEANUP_FINISH

Termina de limpiar la pila cuando sale un bloque de instrucciones `with`.

TOS es el resultado de la llamada a la función `__exit__()` o `__aexit__()` apilada por `WITH_CLEANUP_START`. `SECOND` es `None` o un tipo de excepción (se activa cuando se ha lanzado una excepción).

Desapila dos valores de la pila. Si `SECOND` no es `None` y TOS es verdadero, se desenrolla el bloque `EXCEPT_HANDLER` que se creó cuando se detectó la excepción y empuja `NULL` a la pila.

Todos los siguientes códigos de operación utilizan sus argumentos.

STORE_NAME (*namei*)

Implementa `name = TOS.namei` es el índice de *name* en el atributo `co_names` del objeto de código. El compilador intenta usar *STORE_FAST* o *STORE_GLOBAL* si es posible.

DELETE_NAME (*namei*)

Implementa `del name`, donde *namei* es el índice en atributo `co_names` del objeto de código.

UNPACK_SEQUENCE (*count*)

Descomprime TOS en *count* valores individuales, que se colocan en la pila de derecha a izquierda.

UNPACK_EX (*counts*)

Implementa la asignación con un objetivo destacado: desempaqueta un iterable en TOS en valores individuales, donde el número total de valores puede ser menor que el número de elementos en el iterable: uno de los nuevos valores será una lista de todos los elementos sobrantes.

El byte bajo de *count* es el número de valores antes del valor de la lista, el byte alto de *count* es el número de valores después de él. Los valores resultantes se colocan en la pila de derecha a izquierda.

STORE_ATTR (*namei*)

Implementa `TOS.name = TOS1`, donde *namei* es el índice del nombre en `co_names`.

DELETE_ATTR (*namei*)

Implementa `del TOS.name`, usando *namei* como índice en `co_names`.

STORE_GLOBAL (*namei*)

Funciona como *STORE_NAME*, pero almacena el nombre como global.

DELETE_GLOBAL (*namei*)

Funciona como *DELETE_NAME*, pero elimina un nombre global.

LOAD_CONST (*consti*)

Apila `co_consts[consti]` en la pila.

LOAD_NAME (*namei*)

Apila el valor asociado con `co_names[namei]` en la pila.

BUILD_TUPLE (*count*)

Crea una tupla que consume elementos *count* de la pila, y apila la tupla resultante a la pila.

BUILD_LIST (*count*)

Funciona como *BUILD_TUPLE*, pero crea una lista.

BUILD_SET (*count*)

Funciona como *BUILD_TUPLE*, pero crea un conjunto.

BUILD_MAP (*count*)

Apila un nuevo objeto de diccionario en la pila. Desapila $2 * count$ elementos para que el diccionario contenga *count* entradas: `{..., TOS3: TOS2, TOS1: TOS}`.

Distinto en la versión 3.5: El diccionario se crea a partir de elementos de la pila en lugar de crear un diccionario vacío dimensionado previamente para contener *count* elementos.

BUILD_CONST_KEY_MAP (*count*)

La versión de *BUILD_MAP* especializada para claves constantes. Desapila el elemento superior en la pila que contiene una tupla de claves, luego, a partir de TOS1, muestra los valores *count* para formar valores en el diccionario incorporado.

Nuevo en la versión 3.6.

BUILD_STRING (*count*)

Concatena *count* cadenas de caracteres de la pila y empuja la cadena de caracteres resultante en la pila.

Nuevo en la versión 3.6.

BUILD_TUPLE_UNPACK (*count*)

Desapila *count* iterables de la pila, los une en una sola tupla y apila el resultado. Implementa el desempaqueado iterable cuando se imprime una tupla `(*x, *y, *z)`.

Nuevo en la versión 3.5.

BUILD_TUPLE_UNPACK_WITH_CALL (*count*)

Esto es similar a [BUILD_TUPLE_UNPACK](#), pero se usa para la sintaxis de llamada `f(*x, *y, *z)`. El elemento de la pila en la posición `count + 1` debe ser la correspondiente invocable `f`.

Nuevo en la versión 3.6.

BUILD_LIST_UNPACK (*count*)

Esto es similar a [BUILD_TUPLE_UNPACK](#), pero apila una lista en lugar de tupla. Implementa el desempaqueto iterable en los despliegues de lista `[*x, *y, *z]`.

Nuevo en la versión 3.5.

BUILD_SET_UNPACK (*count*)

Esto es similar a [BUILD_TUPLE_UNPACK](#), pero apila un conjunto en lugar de tupla. Implementa el desempaqueto iterable en los despliegues de conjunto `{*x, *y, *z}`.

Nuevo en la versión 3.5.

BUILD_MAP_UNPACK (*count*)

Desapila *count* mapeos de la pila, los fusiona en un solo diccionario y apila el resultado. Implementa el desempaqueto del diccionario en las pantallas del diccionario `{**x, **y, **z}`.

Nuevo en la versión 3.5.

BUILD_MAP_UNPACK_WITH_CALL (*count*)

Esto es similar a [BUILD_MAP_UNPACK](#), pero se usa para la sintaxis de llamada `f(**x, **y, **z)`. El elemento de la pila en la posición `count + 2` debe ser la correspondiente invocable `f`.

Nuevo en la versión 3.5.

Distinto en la versión 3.6: La posición del invocable se determina agregando 2 al argumento opcode en lugar de codificarlo en el segundo byte del argumento.

LOAD_ATTR (*namei*)

Reemplaza TOS con `getattr(TOS, co_names[namei])`.

COMPARE_OP (*opname*)

Realiza una operación booleana. El nombre de la operación se puede encontrar en `cmp_op[opname]`.

IMPORT_NAME (*namei*)

Importa el módulo `co_names[namei]`. TOS y TOS1 aparecen y proporcionan los argumentos *fromlist* y *level* de `__import__()`. El objeto del módulo se empuja a la pila. El espacio de nombres actual no se ve afectado: para una instrucción de importación adecuada, una instrucción posterior [STORE_FAST](#) modifica el espacio de nombres.

IMPORT_FROM (*namei*)

Carga el atributo `co_names[namei]` del módulo que se encuentra en TOS. El objeto resultante se apila en la pila, para luego ser almacenado por la instrucción [STORE_FAST](#).

JUMP_FORWARD (*delta*)

Incrementa el contador de bytecode en *delta*.

POP_JUMP_IF_TRUE (*target*)

Si TOS es true, establece el contador de bytecode en *target*. TOS es desapilado (*popped*).

Nuevo en la versión 3.1.

POP_JUMP_IF_FALSE (*target*)

Si TOS es falso, establece el contador de bytecode en *target*. TOS es desapilado (*popped*).

Nuevo en la versión 3.1.

JUMP_IF_TRUE_OR_POP (*target*)

Si TOS es verdadero, establece el contador de bytecode en *target* y deja TOS en la pila. De lo contrario (TOS es falso), TOS se desapila.

Nuevo en la versión 3.1.

JUMP_IF_FALSE_OR_POP (*target*)

Si TOS es falso, establece el contador de bytecode en *target* y deja TOS en la pila. De lo contrario (TOS es verdadero), TOS se desapila.

Nuevo en la versión 3.1.

JUMP_ABSOLUTE (*target*)

Establezca el contador de bytecode en *target*.

FOR_ITER (*delta*)

TOS es un *iterador*. Llama a su método `__next__()`. Si esto produce un nuevo valor, lo apila en la pila (dejando el iterador debajo de él). Si el iterador indica que está agotado, se abre TOS y el contador de código de bytes se incrementa en *delta*.

LOAD_GLOBAL (*namei*)

Carga el nombre global `co_names[namei]` en la pila.

SETUP_FINALLY (*delta*)

Apila un bloque try de una cláusula try-finally o try-except en la pila de bloques. *delta* apunta al último bloque o al primero excepto el bloque.

CALL_FINALLY (*delta*)

Apila la dirección de la siguiente instrucción en la pila e incrementa el contador de bytecode en *delta*. Se usa para llamar al bloque finalmente como una «subrutina».

Nuevo en la versión 3.8.

LOAD_FAST (*var_num*)

Apila una referencia al local `co_varnames[var_num]` sobre la pila.

STORE_FAST (*var_num*)

Almacena TOS en el local `co_varnames[var_num]`.

DELETE_FAST (*var_num*)

Elimina la `co_varnames[var_num]` local.

LOAD_CLOSURE (*i*)

Apila una referencia a la celda contenida en la ranura *i* de la celda y el almacenamiento variable libre. El nombre de la variable es `co_cellvars[i]` si *i* es menor que la longitud de `co_cellvars`. De lo contrario, es `co_freevars[i - len(co_cellvars)]`.

LOAD_DEREF (*i*)

Carga la celda contenida en la ranura *i* de la celda y el almacenamiento variable libre. Apila una referencia al objeto que contiene la celda en la pila.

LOAD_CLASSDEREF (*i*)

Al igual que `LOAD_DEREF` pero primero verifica el diccionario local antes de consultar la celda. Esto se usa para cargar variables libres en los cuerpos de clase.

Nuevo en la versión 3.4.

STORE_DEREF (*i*)

Almacena TOS en la celda contenida en la ranura *i* de la celda y almacenamiento variable libre.

DELETE_DEREF (*i*)

Vacía la celda contenida en la ranura *i* de la celda y el almacenamiento variable libre. Utilizado por la declaración `del`.

Nuevo en la versión 3.2.

RAISE_VARARGS (*argc*)

Provoca una excepción utilizando una de las 3 formas de la declaración `raise`, dependiendo del valor de *argc*:

- 0: `raise` (vuelve a lanzar la excepción anterior)
- 1: `raise TOS` (lanza instancia de excepción o un tipo en TOS)
- 2: `raise TOS1 desde TOS` (lanza una instancia de excepción o tipo en TOS1 con `__cause__` establecida en “TOS”)

CALL_FUNCTION (*argc*)

Llama a un objeto invocable con argumentos posicionales. *argc* indica el número de argumentos posicionales. La parte superior de la pila contiene argumentos posicionales, con el argumento más a la derecha en la parte superior. Debajo de los argumentos hay un objeto invocable para llamar. `CALL_FUNCTION` saca todos los argumentos y el objeto invocable de la pila, llama al objeto invocable con esos argumentos y empuja el valor de retorno retornado por el objeto invocable.

Distinto en la versión 3.6: Este código de operación se usa solo para llamadas con argumentos posicionales.

CALL_FUNCTION_KW (*argc*)

Llama a un objeto invocable con argumentos posicionales (si los hay) y palabras clave. *argc* indica el número total de argumentos posicionales y de palabras clave. El elemento superior en la pila contiene una tupla de nombres de argumentos de palabras clave. Debajo de eso hay argumentos de palabras clave en el orden correspondiente a la tupla. Debajo de eso hay argumentos posicionales, con el parámetro más a la derecha en la parte superior. Debajo de los argumentos hay un objeto invocable para llamar. `CALL_FUNCTION_KW` saca todos los argumentos y el objeto invocable de la pila, llama al objeto invocable con esos argumentos y empuja el valor de retorno retornado por el objeto invocable.

Distinto en la versión 3.6: Los argumentos de palabras clave se empaquetan en una tupla en lugar de un diccionario, *argc* indica el número total de argumentos.

CALL_FUNCTION_EX (*flags*)

Llama a un objeto invocable con un conjunto variable de argumentos posicionales y de palabras clave. Si se establece el bit más bajo de *flags*, la parte superior de la pila contiene un objeto de mapeo que contiene argumentos de palabras clave adicionales. Debajo de eso hay un objeto iterable que contiene argumentos posicionales y un objeto invocable para llamar. `BUILD_MAP_UNPACK_WITH_CALL` y `BUILD_TUPLE_UNPACK_WITH_CALL` puede usarse para fusionar múltiples objetos de mapeo e iterables que contienen argumentos. Antes de llamar al invocable, el objeto de mapeo y el objeto iterable se «desempaquetan» y sus contenidos se pasan como palabras clave y argumentos posicionales, respectivamente. `CALL_FUNCTION_EX` saca todos los argumentos y el objeto invocable de la pila, llama al objeto invocable con esos argumentos y empuja el valor de retorno retornado por el objeto invocable.

Nuevo en la versión 3.6.

LOAD_METHOD (*namei*)

Carga un método llamado `co_names[namei]` desde el objeto TOS. TOS aparece. Este bytecode distingue dos casos: si TOS tiene un método con el nombre correcto, el bytecode apila el método no vinculado y TOS. TOS se usará como primer argumento (`self`) por `CALL_METHOD` cuando se llama al método independiente. De lo contrario, `NULL` y el objeto retornado por la búsqueda de atributos son apilados.

Nuevo en la versión 3.7.

CALL_METHOD (*argc*)

Llama a un método. *argc* es el número de argumentos posicionales. Los argumentos de palabras clave no son compatibles. Este código de operación está diseñado para usarse con `LOAD_METHOD`. Los argumentos posicionales están en la parte superior de la pila. Debajo de ellos, los dos elementos descritos en `LOAD_METHOD` están en la pila (`self` y un objeto de método independiente o `NULL` y un invocable arbitrario). Todos ellos aparecen y se apila el valor de retorno.

Nuevo en la versión 3.7.

MAKE_FUNCTION (*flags*)

Apila un nuevo objeto de función en la pila. De abajo hacia arriba, la pila consumida debe constar de valores si el argumento lleva un valor de marca especificado

- `0x01`, una tupla de valores predeterminados para solo parámetros posicionales y posicionales o de palabras clave en orden posicional
- `0x02` un diccionario de valores predeterminados de solo palabras clave
- `0x04` un diccionario de anotaciones
- `0x08` una tupla que contiene celdas para variables libres, haciendo un cierre (*closure*)
- el código asociado con la función (en TOS1)

- el *nombre calificado* de la función (en TOS)

BUILD_SLICE (*argc*)

Apila un objeto de rebanada en la pila. *argc* debe ser 2 o 3. Si es 2, se apila `slice(TOS1, TOS)`; si es 3, se apila `slice(TOS2, TOS1, TOS)`. Consulte la función incorporada `slice()` para obtener más información.

EXTENDED_ARG (*ext*)

Prefija cualquier código de operación que tenga un argumento demasiado grande para caber en el byte predefinido. *ext* contiene un byte adicional que actúa como bits más altos en el argumento. Para cada opcode, como máximo se permiten tres prefijos `EXTENDED_ARG`, formando un argumento de dos bytes a cuatro bytes.

FORMAT_VALUE (*flags*)

Se utiliza para implementar cadenas literales formateadas (cadenas de caracteres f). Desapila un *fmt_spec* opcional de la pila, luego un *value* requerido. *flags* se interpreta de la siguiente manera:

- `(flags & 0x03) == 0x00`: *value* es formateado como está.
- `(flags & 0x03) == 0x01`: llama `str()` sobre *value* antes de formatearlo.
- `(flags & 0x03) == 0x02`: llama `repr()` sobre *value* antes de formatearlo.
- `(flags & 0x03) == 0x03`: llama `ascii()` sobre *value* antes de formatearlo.
- `(flags & 0x04) == 0x04`: desapila *fmt_spec* de la pila y lo usa, de lo contrario usa un *fmt_spec* vacío.

El formateo se realiza usando `PyObject_Format()`. El resultado se apila en la pila.

Nuevo en la versión 3.6.

HAVE_ARGUMENT

Esto no es realmente un opcode. Identifica la línea divisoria entre los opcode que no usan su argumento y los que lo hacen (`< HAVE_ARGUMENT` y `>= HAVE_ARGUMENT`, respectivamente).

Distinto en la versión 3.6: Ahora cada instrucción tiene un argumento, pero los códigos de operación `<HAVE_ARGUMENT` la ignoran. Antes, solo los códigos de operación `>= HAVE_ARGUMENT` tenían un argumento.

32.12.4 Colecciones opcode

Estas colecciones se proporcionan para la introspección automática de instrucciones de bytecode:

dis.opname

Secuencia de nombres de operaciones, indexable utilizando el bytecode.

dis.opmap

Nombres de operaciones de mapeo de diccionario a bytecodes.

dis.cmp_op

Secuencia de todos los nombres de operaciones de comparación.

dis.hasconst

Secuencia de bytecodes que acceden a una constante.

dis.hasfree

Secuencia de bytecodes que acceden a una variable libre (tenga en cuenta que “libre” en este contexto se refiere a nombres en el alcance actual a los que hacen referencia los ámbitos internos o los nombres en los ámbitos externos a los que se hace referencia desde este ámbito. **No** incluye referencias a ámbitos globales o integrados).

dis.hasname

Secuencia de bytecodes que acceden a un atributo por nombre.

dis.hasjrel

Secuencia de bytecodes que tienen un objetivo de salto relativo.

`dis.hasjabs`

Secuencia de bytecodes que tienen un objetivo de salto absoluto.

`dis.haslocal`

Secuencia de códigos de bytes que acceden a una variable local.

`dis.hascompare`

Secuencia de bytecodes de operaciones booleanas.

32.13 `pickletools` — Herramientas para desarrolladores pickle

Código fuente: [Lib/pickletools.py](#)

Este módulo contiene varias constantes relacionadas con los detalles íntimos del módulo `pickle`, algunos comentarios largos sobre la implementación y algunas funciones útiles para analizar pickled data. El contenido de este módulo es útil para los desarrolladores principales de Python que están trabajando en el `pickle`; los usuarios ordinarios del módulo `pickle` probablemente no encontrarán relevante el módulo `pickletools`.

32.13.1 Uso de la línea de comandos

Nuevo en la versión 3.2.

Cuando se invoca desde la línea de comandos, `python -m pickletools` desensamblará el contenido de uno o más archivos pickle. Tenga en cuenta que si desea ver el objeto Python almacenado en el pickle en lugar de los detalles del formato de pickle, es posible que desee utilizar `-m pickle` en su lugar. Sin embargo, cuando el archivo de pickle que desea examinar proviene de una fuente que no es de confianza, `-m pickletools` es una opción más segura porque no ejecuta el código de bytes de pickle.

Por ejemplo, con una tupla `(1, 2)` pickled en el archivo `x.pickle`:

```
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
0: \x80 PROTO      3
2: K    BININT1    1
4: K    BININT1    2
6: \x86 TUPLE2
7: q    BINPUT     0
9: .    STOP
highest protocol among opcodes = 2
```

Opciones de línea de comandos

`-a, --annotate`

Anote cada línea con una breve descripción del código de operación.

`-o, --output=<file>`

Nombre de un archivo donde se debe escribir la salida.

`-l, --indentlevel=<num>`

Número de espacios en blanco por los que se aplica una sangría a un nuevo nivel de MARK.

`-m, --memo`

Cuando se desensamblan varios objetos, conserve la nota entre los ensamblajes.

`-p, --preamble=<preamble>`

Cuando se especifica más de un archivo pickle, imprima un preámbulo determinado antes de cada desensamblado.

32.13.2 Interfaz programática

`pickletools.dis` (*pickle*, *out=None*, *memo=None*, *indentlevel=4*, *annotate=0*)

Produce un desensamblado simbólico del pickle en el objeto similar a un archivo *salida*, de forma predeterminada en `sys.stdout`. *pickle* puede ser una cadena o un objeto similar a un archivo. *memo* puede ser un diccionario Python que se utilizará como nota del pickle; se puede utilizar para realizar ensamblajes de desuso en varios pickles creados por el mismo selector. Los niveles sucesivos, indicados por los códigos de operación MARK en la secuencia, son indentados por espacios *indentlevel*. Si se da un valor distinto de cero a *anotar*, cada código de operación de la salida se anota con una breve descripción. El valor de *anotar* se utiliza como sugerencia para la columna donde debe comenzar la anotación.

Nuevo en la versión 3.2: El argumento *anotar*.

`pickletools.genops` (*pickle*)

Proporciona un *iterator* sobre todos los códigos de operación en un pickle, retornando una secuencia de triples (*opcode*, *arg*, *pos*). *opcode* es una instancia de una clase `OpcodeInfo`; *arg* es el valor descodificado, como un objeto Python, del argumento del código de operación; *pos* es la posición en la que se encuentra este código de operación. *pickle* puede ser una cadena o un objeto similar a un archivo.

`pickletools.optimize` (*picklestring*)

Retorna una nueva cadena pickle equivalente después de eliminar los códigos de operación PUT no utilizados. El pickle optimizado es más corto, toma menos tiempo de transmisión, requiere menos espacio de almacenamiento y se restaura de manera más eficiente.

Los módulos descritos en este capítulo proporcionan varios servicios que están disponibles en todas las versiones de Python. Esta es una descripción general:

33.1 `formatter` — Formateo de salida genérica

Obsoleto desde la versión 3.4: Debido a la falta de uso, el módulo formateador ha quedado obsoleto.

Este módulo admite dos definiciones de interfaz, cada una con múltiples implementaciones: la interfaz *formateador* y la interfaz *escritor* que es requerida por la interfaz formateadora.

Los objetos formateadores transforman un flujo abstracto de eventos de formato en eventos de salida específicos en los objetos del escritor. Los formateadores gestionan varias estructuras de pila para permitir cambiar y restaurar varias propiedades de un objeto escritor; los escritores no necesitan poder manejar cambios relativos ni ningún tipo de operación de «retroceso». Las propiedades específicas del escritor que se pueden controlar a través de los objetos del formateador son la alineación horizontal, la fuente y las sangrías del margen izquierdo. Se proporciona un mecanismo que también permite proporcionar configuraciones de estilo arbitrarias y no exclusivas a un escritor. Las interfaces adicionales facilitan el formateo de eventos que no son reversibles, como la separación de párrafos.

Los objetos escritores encapsulan las interfaces de los dispositivos. Se admiten dispositivos abstractos, como formatos de archivo, así como dispositivos físicos. Todas las implementaciones proporcionadas funcionan con dispositivos abstractos. La interfaz pone a disposición mecanismos para establecer las propiedades que administran los objetos del formateador e insertar datos en la salida.

33.1.1 La interfaz Formateador

Las interfaces para crear formateadores dependen de la clase de formateador específica que se instancia. Las interfaces que se describen a continuación son las interfaces necesarias que todos los formateadores deben admitir una vez inicializados.

Un elemento de datos se define a nivel de módulo:

`formatter.AS_IS`

Valor que se puede usar en la especificación de fuente pasada al método `push_font()` descrito a continuación, o como el nuevo valor a cualquier otro método `push_property()`. Al presionar el valor `AS_IS`, se puede llamar al método correspondiente `pop_property()` sin tener que rastrear si se cambió la propiedad.

Los siguientes atributos están definidos para los objetos de instancia del formateador:

`formatter.writer`

La instancia de escritor con la que interactúa el formateador.

`formatter.end_paragraph (blanklines)`

Cierra todos los párrafos abiertos e inserta al menos *blanklines* antes del siguiente párrafo.

`formatter.add_line_break ()`

Agrega un salto de línea duro si aún no existe uno. Esto no rompe el párrafo lógico.

`formatter.add_hor_rule (*args, **kw)`

Inserta una regla horizontal en la salida. Se inserta una ruptura dura si hay datos en el párrafo actual, pero el párrafo lógico no está roto. Los argumentos y palabras clave se pasan al método del escritor `send_line_break()`.

`formatter.add_flowring_data (data)`

Proporciona datos que deben formatearse con espacios en blanco contraídos. Los espacios en blanco de llamadas anteriores y sucesivas a `add_flowring_data()` también se consideran cuando se realiza el colapso de espacios en blanco. Se espera que los datos que se pasan a este método estén envueltos en palabras por el dispositivo de salida. Tenga en cuenta que el objeto escritor debe realizar cualquier ajuste de texto debido a la necesidad de depender de la información del dispositivo y la fuente.

`formatter.add_literal_data (data)`

Proporciona datos que deben pasarse al escritor sin cambios. Los espacios en blanco, incluidos los caracteres de nueva línea y tabulación, se consideran legales en el valor de *data*.

`formatter.add_label_data (format, counter)`

Inserta una etiqueta que debe colocarse a la izquierda del margen izquierdo actual. Esto debe usarse para construir listas numeradas o con viñetas. Si el valor de *format* es una cadena, se interpreta como una especificación de formato para *counter*, que debe ser un número entero. El resultado de este formato se convierte en el valor de la etiqueta; si *format* no es una cadena, se utiliza directamente como valor de etiqueta. El valor de la etiqueta se pasa como el único argumento del método del escritor `send_label_data()`. La interpretación de valores de etiquetas que no son cadenas depende del escritor asociado.

Las especificaciones de formato son cadenas que, en combinación con un valor de contador, se utilizan para calcular valores de etiqueta. Cada carácter de la cadena de formato se copia al valor de la etiqueta, y algunos caracteres se reconocen para indicar una transformación en el valor del contador. Específicamente, el carácter '1' representa el formateador del valor del contador como un número arábigo, los caracteres 'A' y 'a' representan representaciones alfabéticas del valor del contador en mayúsculas y minúsculas, respectivamente, y 'I' y 'i' representan el valor del contador en números romanos, en mayúsculas y minúsculas. Tenga en cuenta que las transformaciones alfabética y romana requieren que el valor del contador sea mayor que cero.

`formatter.flush_softspace ()`

Envía cualquier espacio en blanco pendiente almacenado en búfer de una llamada anterior a `add_flowring_data()` al objeto escritor asociado. Esto debe llamarse antes de cualquier manipulación directa del objeto de escritura.

`formatter.push_alignment (align)`

Empuja una nueva configuración de alineación en la pila de alineación. Puede ser *AS_IS* si no se desea ningún cambio. Si el valor de alineación se cambia con respecto a la configuración anterior, se llama al método del escritor `new_alignment()` con el valor *align*.

`formatter.pop_alignment ()`

Restaura la alineación anterior.

`formatter.push_font ((size, italic, bold, teletype))`

Cambia algunas o todas las propiedades de fuente del objeto de escritura. Las propiedades que no se establecen en *AS_IS* se establecen en los valores pasados, mientras que otras se mantienen en su configuración actual. Se llama al método del escritor `new_font()` con la especificación de fuente completamente resuelta.

`formatter.pop_font ()`

Restaura la fuente anterior.

`formatter.push_margin(margin)`

Aumenta el número de sangrías del margen izquierdo en uno, asociando la etiqueta lógica *margin* con la nueva sangría. El nivel de margen inicial es 0. Los valores modificados de la etiqueta lógica deben ser valores verdaderos; los valores falsos distintos de `AS_IS` no son suficientes para cambiar el margen.

`formatter.pop_margin()`

Restaura el margen anterior.

`formatter.push_style(*styles)`

Empuja cualquier número de especificaciones de estilo arbitrarias. Todos los estilos se insertan en la pila de estilos en orden. Una tupla que representa la pila completa, incluidos los valores `AS_IS`, se pasa al método `new_styles()` del escritor.

`formatter.pop_style(n=1)`

Muestra las últimas *n* especificaciones de estilo pasadas a `push_style()`. Una tupla que representa la pila revisada, que incluye los valores `AS_IS`, se pasa al método `new_styles()` del escritor.

`formatter.set_spacing(spacing)`

Establece el estilo de espaciado para el escritor.

`formatter.assert_line_data(flag=1)`

Informa al formateador que se han agregado datos al párrafo actual fuera de banda. Esto debe usarse cuando el escritor haya sido manipulado directamente. El argumento *flag* opcional se puede establecer en falso si las manipulaciones del escritor produjeron un salto de línea duro al final de la salida.

33.1.2 Implementaciones del formateador

Este módulo proporciona dos implementaciones de objetos formateadores. La mayoría de las aplicaciones pueden utilizar una de estas clases sin modificación ni subclases.

class `formatter.NullFormatter(writer=None)`

Un formateador que no hace nada. Si se omite *writer*, se crea una instancia `NullWriter`. Ningún método del escritor es llamado por instancias `NullFormatter`. Las implementaciones deben heredar de esta clase si implementan una interfaz de escritor, pero no necesitan heredar ninguna implementación.

class `formatter.AbstractFormatter(writer)`

El formateador estándar. Esta implementación ha demostrado una amplia aplicabilidad para muchos escritores y puede usarse directamente en la mayoría de las circunstancias. Se ha utilizado para implementar un navegador World Wide Web con todas las funciones.

33.1.3 La interfaz Escritor

Las interfaces para crear escritores dependen de la clase de escritor específica que se instancia. Las interfaces que se describen a continuación son las interfaces necesarias que todos los escritores deben admitir una vez inicializadas. Tenga en cuenta que, si bien la mayoría de las aplicaciones pueden usar la clase `AbstractFormatter` como formateador, el escritor generalmente debe ser proporcionado por la aplicación.

`writer.flush()`

Vacía cualquier salida almacenada o eventos de control de dispositivos.

`writer.new_alignment(align)`

Define el estilo de alineación. El valor *align* puede ser cualquier objeto, pero por convención es una cadena o `None`, donde `None` indica que se debe usar la alineación «preferida» del escritor. Los valores de *align* convencionales son `"left"`, `"center"`, `"right"`, y `"justify"`.

`writer.new_font(font)`

Establece el estilo de fuente. El valor de *font* será `None`, lo que indica que se debe usar la fuente predeterminada del dispositivo, o una tupla de la forma (*tamaño*, *cursiva*, *negrita*, *teletipo*). El *tamaño* será una cadena que indica el tamaño de fuente que se debe utilizar; cadenas específicas y su interpretación deben ser definidas por la aplicación. Los valores *cursiva*, *negrita* y *teletipo* son valores booleanos que especifican cuál de esos atributos de fuente debe usarse.

`writer.new_margin (margin, level)`

Establece el nivel de margen en el entero *level* y la etiqueta lógica en *margin*. La interpretación de la etiqueta lógica queda a discreción del escritor; la única restricción sobre el valor de la etiqueta lógica es que no sea un valor falso para valores distintos de cero de *level*.

`writer.new_spacing (spacing)`

Establece el estilo de espaciado en *spacing*.

`writer.new_styles (styles)`

Establece estilos adicionales. El valor de *styles* es una tupla de valores arbitrarios; el valor *AS_IS* debe ignorarse. La tupla *styles* se puede interpretar como un conjunto o como una pila dependiendo de los requisitos de la aplicación y la implementación del escritor.

`writer.send_line_break ()`

Rompe la línea actual.

`writer.send_paragraph (blankline)`

Produce una separación de párrafos de al menos *blankline* líneas en blanco, o su equivalente. El valor de *blankline* será un número entero. Tenga en cuenta que la implementación recibirá una llamada a *send_line_break ()* antes de esta llamada si se necesita un salto de línea; este método no debe incluir terminar la última línea del párrafo. Solo es responsable del espacio vertical entre párrafos.

`writer.send_hor_rule (*args, **kw)`

Muestra una regla horizontal en el dispositivo de salida. Los argumentos de este método son completamente específicos de la aplicación y del escritor, y deben interpretarse con cuidado. La implementación del método puede asumir que ya se ha emitido un salto de línea a través de *send_line_break ()*.

`writer.send_flowling_data (data)`

Datos de caracteres de salida que se pueden ajustar en palabras y volver a fluir según sea necesario. Dentro de cualquier secuencia de llamadas a este método, el escritor puede asumir que se han contraído tramos de múltiples caracteres de espacio en blanco a caracteres de un solo espacio.

`writer.send_literal_data (data)`

Salida de datos de caracteres que ya se formatearon para su visualización. En general, esto debe interpretarse en el sentido de que los saltos de línea indicados por caracteres de nueva línea deben conservarse y no deben introducirse nuevos saltos de línea. Los datos pueden contener caracteres de tabulación y nueva línea incrustados, a diferencia de los datos proporcionados en la interfaz *send_formatted_data ()*.

`writer.send_label_data (data)`

Establece *data* a la izquierda del margen izquierdo actual, si es posible. El valor de *data* no está restringido; el tratamiento de los valores que no son cadenas depende completamente de la aplicación y del escritor. Este método solo se llamará al comienzo de una línea.

33.1.4 Implementaciones del escritor

Este módulo proporciona tres implementaciones de la interfaz de objetos del escritor como ejemplos. La mayoría de las aplicaciones necesitarán derivar nuevas clases de escritor de la clase *NullWriter*.

class `formatter.NullWriter`

Un escritor que solo proporciona la definición de la interfaz; no se toman acciones sobre ningún método. Esta debería ser la clase base para todos los escritores que no necesitan heredar ningún método de implementación.

class `formatter.AbstractWriter`

Un escritor que se puede utilizar para depurar formateadores, pero no mucho más. Cada método simplemente se anuncia a sí mismo imprimiendo su nombre y argumentos en la salida estándar.

class `formatter.DumbWriter (file=None, maxcol=72)`

Clase de escritor simple que escribe la salida en el *file object* pasado como *file* o, si se omite *file*, en la salida estándar. La salida simplemente se ajusta en palabras al número de columnas especificado por *maxcol*. Esta clase es adecuada para reajustar una secuencia de párrafos.

Servicios Específicos para MS Windows

Este capítulo describe los módulos que sólo están disponibles en plataformas MS Windows.

34.1 `msilib` — Leer y escribir archivos *Microsoft Installer*

Código fuente: `Lib/msilib/__init__.py`

El `msilib` soporta la creación de archivos *Microsoft Installer* (`.msi`). Como estos archivos a menudo contienen archivos *cabinet* (`.cab`) embebidos, se expone también una API para crear archivos CAB. El soporte para leer archivos `.cab` todavía no está implementado; el soporte a lectura de base de datos `.msi` es posible.

Este paquete se centra en proveer acceso completo a todas las tablas de un archivo `.msi`, por lo que se puede considerar una API de bajo nivel. Dos aplicaciones principales de este paquete son el comando `bdist_msi` de `distutils`, y la creación del propio paquete instalador de Python (aunque utiliza una versión diferente de `msilib`).

El contenido del paquete se puede dividir en cuatro partes: rutinas CAB de bajo nivel, rutinas MSI de bajo nivel, rutinas MSI de alto nivel, y estructuras de tabla estándar.

`msilib.FCICreate (cabname, files)`

Crea un nuevo archivo CAB llamado *cabname*. *files* debe ser una lista de tuplas, donde cada una contiene el nombre del archivo en disco, y el nombre del archivo dentro del archivo CAB.

Los archivos son añadidos al archivo CAB en el orden en el cual aparecen en la lista. Todos los archivos son añadidos a un solo archivo CAB, utilizando el algoritmo de compresión MSZIP.

Las retrollamadas a Python para los diferentes pasos de la creación de MSI no están actualmente expuestas.

`msilib.UuidCreate ()`

Retorna la representación de un nuevo identificador único como cadena de caracteres. Esto envuelve las funciones `UuidCreate ()` y `UuidToString ()` de la API de Windows.

`msilib.OpenDatabase (path, persist)`

Retorna una nueva base de datos llamando a `MsiOpenDatabase`. *path* es el nombre del archivo MSI; *persist* puede ser una de las constantes `MSIDBOPEN_CREATEDIRECT`, `MSIDBOPEN_CREATE`, `MSIDBOPEN_DIRECT`, `MSIDBOPEN_READONLY` o `MSIDBOPEN_TRANSACT`, y puede incluir la bandera `MSIDBOPEN_PATCHFILE`. El significado de estas banderas se puede consultar en la documentación de Microsoft; dependiendo de las banderas, se abrirá una base de datos existente, o se creará una nueva.

`msilib.CreateRecord(count)`

Retorna un nuevo objeto de registro llamando a `MSICreateRecord()`. *count* es el número de campos del registro.

`msilib.init_database(name, schema, ProductName, ProductCode, ProductVersion, Manufacturer)`

Crea y retorna una nueva base de datos *name*, la inicializa con *schema*, y establece las propiedades *ProductName*, *ProductCode*, *ProductVersion* y *Manufacturer*.

schema debe ser un objeto módulo que contenga los atributos `tables` y `_Validation_records`; normalmente se debería usar `msilib.schema`.

La base de datos contendrá únicamente el esquema y los registros de validación cuando esta función retorne.

`msilib.add_data(database, table, records)`

Añade todos los *records* de la tabla llamada *table* a *database*.

El argumento *table* debe ser una de las tablas predefinidas en el esquema MSI, como 'Feature', 'File', 'Component', 'Dialog', 'Control', etc.

records debería ser una lista de tuplas, donde cada una contenga todos los campos de un registro, acorde al esquema de la tabla. Para los campos opcionales se puede asignar `None`.

Los valores de los campos pueden ser enteros, cadenas de caracteres o instancias de la clase `Binary`.

class `msilib.Binary(filename)`

Representa entradas en la tabla `Binary`; insertar un objeto así utilizando `add_data()` lee el archivo llamado *filename* en la tabla.

`msilib.add_tables(database, module)`

Añade todos los contenidos de la tabla de *module* a *database*. *module* debe contener un atributo `tables`, que liste todas las tablas cuyos contenidos deban ser añadidos, y un atributo por tabla que contenga el propio contenido.

Esto suele utilizarse para instalar las tablas secuenciales.

`msilib.add_stream(database, name, path)`

Añade el archivo *path* en la tabla `_Stream` de *database*, con el nombre del flujo *name*.

`msilib.gen_uuid()`

Retorna un nuevo UUID, en el formato que MSI suele requerir (entre llaves, con todos los dígitos hexadecimales en mayúsculas).

Ver también:

[FCICreate UuidCreate UuidToString](#)

34.1.1 Objetos Database

`Database.OpenView(sql)`

Retorna un objeto de vista, llamando a `MSIDatabaseOpenView()`. *sql* es la sentencia SQL a ejecutar.

`Database.Commit()`

Persiste (**commit**) los cambios pendientes en la transacción actual, llamando a `MSIDatabaseCommit()`.

`Database.GetSummaryInformation(count)`

Retorna un nuevo objeto de información resumida, llamando a `MsiGetSummaryInformation()`. *count* es el número máximo de valores actualizados.

`Database.Close()`

Cierra el objeto de base de datos, mediante `MsiCloseHandle()`.

Nuevo en la versión 3.7.

Ver también:

[MSIDatabaseOpenView](#) [MSIDatabaseCommit](#) [MSIGetSummaryInformation](#) [MsiCloseHandle](#)

34.1.2 Objetos View

`View.Execute(params)`

Ejecuta la consulta SQL de la vista, mediante `MsiViewExecute()`. Si *params* no es `None`, es un registro que describe los valores actuales de los *tokens* del parámetro en la consulta.

`View.GetColumnInfo(kind)`

Retorna un registro que describe las columnas de la vista, llamando a `MsiViewGetColumnInfo()`. *kind* puede ser `MSICOLINFO_NAMES` o `MSICOLINFO_TYPES`.

`View.Fetch()`

Retorna un registro de resultado de la consulta, llamando a `MsiViewFetch()`.

`View.Modify(kind, data)`

Modifica la vista, llamando a `MsiViewModify()`. *kind* puede ser `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` o `MSIMODIFY_VALIDATE_DELETE`.

data debe ser un registro que describa los nuevos datos.

`View.Close()`

Cierra la vista, mediante `MsiViewClose()`.

Ver también:

`MsiViewExecute` `MsiViewGetColumnInfo` `MsiViewFetch` `MsiViewModify` `MsiViewClose`

34.1.3 Objetos Summary Information

`SummaryInformation.GetProperty(field)`

Retorna una propiedad del resumen, mediante `MsiSummaryInfoGetProperty()`. *field* es el nombre de la propiedad, y puede ser una de las constantes: `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` o `PID_SECURITY`.

`SummaryInformation.GetPropertyCount()`

Retorna el número de propiedades del resumen, mediante `MsiSummaryInfoGetPropertyCount()`.

`SummaryInformation.SetProperty(field, value)`

Establece una propiedad mediante `MsiSummaryInfoSetProperty()`. *field* puede tener los mismos valores que en `GetProperty()`; *value* es el nuevo valor de la propiedad. Los tipos de los valores pueden ser enteros o cadenas de caracteres.

`SummaryInformation.Persist()`

Escribe las propiedades modificadas al flujo de información resumida, mediante `MsiSummaryInfoPersist()`.

Ver también:

`MsiSummaryInfoGetProperty` `MsiSummaryInfoGetPropertyCount` `MsiSummaryInfoSetProperty` `MsiSummaryInfoPersist`

34.1.4 Objetos Record

`Record.GetFieldCount()`

Retorna el número de campos del registro, mediante `MsiRecordGetFieldCount()`.

`Record.GetInteger(field)`

Retorna el valor de *field* como entero cuando sea posible. *field* debe ser un entero.

`Record.GetString(field)`

Retorna el valor de *field* como una cadena de caracteres cuando sea posible. *field* debe ser un entero.

`Record.SetString(field, value)`

Establece *field* a *value* mediante `MsiRecordSetString()`. *field* debe ser un entero; *value* una cadena de caracteres.

`Record.SetStream(field, value)`

Establece *field* al contenido del archivo llamado *value*, mediante `MsiRecordSetStream()`. *field* debe ser un entero; *value* una cadena de caracteres.

`Record.SetInteger(field, value)`

Establece *field* a *value* mediante `MsiRecordSetInteger()`. *field* y *value* deben ser enteros.

`Record.ClearData()`

Establece todos los campos del registro a 0, mediante `MsiRecordClearData()`.

Ver también:

[MsiRecordGetFieldCount](#) [MsiRecordSetString](#) [MsiRecordSetStream](#) [MsiRecordSetInteger](#) [MsiRecordClearData](#)

34.1.5 Errores

Todos los *wrappers* sobre funciones MSI lanzan `MSIError`; la cadena de caracteres dentro de la excepción contendrá más detalles.

34.1.6 Objetos CAB

class `msilib.CAB(name)`

La clase `CAB` representa un archivo CAB. Durante la construcción del MSI, los archivos se añadirán simultáneamente a la table `Files`, y a un archivo CAB. Después, cuando todos los archivos sean añadidos, el archivo CAB puede ser sobrescrito, y finalmente añadido al archivo MSI.

name es el nombre del archivo CAB dentro del archivo MSI.

append (*full, file, logical*)

Añade el archivo con la ruta *full* al archivo CAB, bajo el nombre *logical*. Si ya existe algún archivo llamado *logical*, se creará un nuevo archivo.

Retorna el índice del archivo dentro del archivo CAB, y el nuevo nombre del archivo dentro del archivo CAB.

commit (*database*)

Genera un archivo CAB, lo añade como un flujo al archivo MSI, lo añade a la tabla `Media`, y elimina el archivo generado del disco.

34.1.7 Objetos Directory

class `msilib.Directory` (*database, cab, basedir, physical, logical, default*[, *componentflags*])

Crea un nuevo directorio en la tabla *Directory*. Hay un componente actual en cada punto temporal para el directorio, el cual es, o bien explícitamente creado mediante `start_component()`, o bien implícito cuando los archivos se añaden por primera vez. Los archivos son añadidos en el componente actual, y al archivo CAB. Para crear un directorio, un objeto de directorio base tiene que ser especificado (puede ser `None`), la ruta al directorio físico, y un nombre de directorio lógico. *default* especifica el zócalo *DefaultDir* en la table de directorio. *componentflags* especifica las banderas por defecto que obtendrán los nuevos componentes.

start_component (*component=None, feature=None, flags=None, keyfile=None, uuid=None*)

Añade una entrada a la tabla *Component*, y hace este componente el actual componente para este directorio. Si no se especifica ningún nombre de componente, se usará el nombre del directorio. Si no se especifica ninguna *feature*, se usará la característica actual. Si no se especifican *flags*, se usarán las banderas por defecto del directorio. Si no se especifica ningún *keyfile*, el *KeyPath* quedará nulo en la tabla *Component*.

add_file (*file, src=None, version=None, language=None*)

Añade el archivo al componente actual del directorio, inicializando uno nuevo si no existe un componente actual. Por defecto, el nombre del archivo en el origen y la tabla del archivo serán idénticos. Si se especifica el archivo *src*, se interpretará como relativo al directorio actual. Opcionalmente, se pueden especificar una *version* y un *language* para la entrada en la tabla *File*.

glob (*pattern, exclude=None*)

Añade una lista de archivos al componente actual, como se especifica en el patrón *glob*. Se pueden excluir archivos individualmente en la lista *exclude*.

remove_pyc ()

Elimina los archivos `.pyc` al desinstalar.

Ver también:

[Directory Table](#) [File Table](#) [Component Table](#) [FeatureComponents Table](#)

34.1.8 Features

class `msilib.Feature` (*db, id, title, desc, display, level=1, parent=None, directory=None, attributes=0*)

Añade un nuevo registro a la tabla *Feature*, usando los valores *id*, *parent.id*, *title*, *desc*, *display*, *level*, *directory* y *attributes*. El objeto de característica resultante se le puede dar al método `start_component()` de *Directory*.

set_current ()

Establece esta característica como la actual característica de *msilib*. Los nuevos componentes serán añadidos automáticamente a la característica por defecto, a no ser que se especifique explícitamente una característica.

Ver también:

[Feature Table](#)

34.1.9 Clases GUI

msilib dispone de varias clases que envuelven a las tablas GUI en una base de datos MSI. Sin embargo, no se dispone de ninguna interfaz de usuario estándar; se puede usar `bdist_msi` para crear archivos MSI con una interfaz de usuario para instalar paquetes Python.

class `msilib.Control` (*dlg, name*)

Clase base de los controles de diálogo. *dlg* es el objeto de diálogo al que pertenece el control, y *name* es el nombre del control.

event (*event, argument, condition=1, ordering=None*)

Crea una entrada en la tabla `ControlEvent` para este control.

mapping (*event, attribute*)

Crea una entrada en la tabla `EventMapping` para este control.

condition (*action, condition*)

Crea una entrada en la tabla `ControlCondition` para este control.

class `msilib.RadioButtonGroup` (*dlg, name, property*)

Crea un control de botón de selección llamado *name*. *property* es la propiedad del instalador que se establece cuando un botón de selección es seleccionado.

add (*name, x, y, width, height, text, value=None*)

Añade un botón de selección llamado *name* al grupo, en las coordenadas *x, y, width, height*, con la etiqueta *text*. Si *value* es `None`, se establecerá por defecto a *name*.

class `msilib.Dialog` (*db, name, x, y, w, h, attr, title, first, default, cancel*)

Retorna un nuevo objeto `Dialog`. Se creará una entrada en la tabla `Dialog` con las coordenadas, atributos de diálogo y título especificados, así como los nombres del primer control y los controles por defecto y de cancelación.

control (*name, type, x, y, width, height, attributes, property, text, control_next, help*)

Retorna un nuevo objeto `Control`. Se creará una entrada en la tabla `Control` con los parámetros especificados.

Este es un método genérico; para tipos específicos hay métodos especializados disponibles.

text (*name, x, y, width, height, attributes, text*)

Añade y retorna un control `Text`.

bitmap (*name, x, y, width, height, text*)

Añade y retorna un control `Bitmap`.

line (*name, x, y, width, height*)

Añade y retorna un control `Line`.

pushbutton (*name, x, y, width, height, attributes, text, next_control*)

Añade y retorna un control `PushButton`.

radiogroup (*name, x, y, width, height, attributes, property, text, next_control*)

Añade y retorna un control `RadioButtonGroup`.

checkbox (*name, x, y, width, height, attributes, property, text, next_control*)

Añade y retorna un control `CheckBox`.

Ver también:

[Dialog Table Control Table Control Types ControlCondition Table ControlEvent Table EventMapping Table RadioButton Table](#)

34.1.10 Tablas pre-calculadas

`msilib` proporciona algunos subpaquetes que contienen únicamente definiciones de esquema y tabla. Actualmente, estas definiciones están basadas en la versión 2.0 de MSI.

`msilib.schema`

Este es el esquema estándar MSI para MSI 2.0, con la variable *tables* proporcionando una lista de definición de tablas, y *_Validation_records* proporcionando la información para la validación MSI.

`msilib.sequence`

Este módulo alberga los contenidos de la tabla para las tablas de secuencia estándar: *AdminExecuteSequence*, *AdminUISequence*, *AdvtExecuteSequence*, *InstallExecuteSequence*, and *InstallUISequence*.

`msilib.text`

Este módulo contiene definiciones para las tablas *UIText* y *ActionText*, para las acciones estándar del instalador.

34.2 msvcrt — Rutinas útiles del entorno de ejecución MS VC++

Estas funciones dan acceso a ciertas capacidades útiles en plataformas Windows. Algunos módulos de más alto nivel usan estas funciones para crear las implementaciones en Windows de sus servicios. Por ejemplo, el módulo *getpass* usa esto en la implementación de la función *getpass()*.

Más información sobre estas funciones se pueden encontrar en la documentación de la API de la plataforma.

El módulo implementa las variantes tanto de caracteres normales como amplios de la API E/S de la consola (se codifican en más de 8 bits, pudiendo llegar hasta 32). La API normal se ocupa solamente de caracteres ASCII y es de uso limitado a aplicaciones internacionales. La API para caracteres amplios se recomienda usar siempre que sea posible.

Distinto en la versión 3.3: Las operaciones en este módulo lanzan ahora *OSError* donde antes se lanzaba *IOError*.

34.2.1 Operaciones con archivos

msvcrt.locking (*fd, mode, nbytes*)

Bloquea parte de un archivo basado en el descriptor del archivo *fd* del entorno de ejecución C. Lanza una excepción *OSError* si falla. La región que ha sido bloqueada se extiende desde la posición del archivo actual hasta *nbytes* bytes y puede que continúe aún habiendo llegado al final del archivo. *mode* tiene que ser una de las constantes *LK_** que están enumeradas más abajo. Se pueden bloquear varias regiones de un mismo archivo pero no se pueden superponer. Las regiones adyacentes no se combinan; tienen que ser desbloqueadas manualmente.

Lanza un *evento de auditoría* *msvcrt.locking* con los argumentos *fd, mode, nbytes*.

msvcrt.LK_LOCK

msvcrt.LK_RLCK

Bloquea los bytes especificados. Si no se pueden bloquear, el programa lo intenta de nuevo tras 1 segundo. Si, tras 10 intentos, no puede bloquear los bytes, se lanza una excepción *OSError*.

msvcrt.LK_NBLCK

msvcrt.LK_NBLCK

Bloquea los bytes especificados. Si no se pueden bloquear, lanza una excepción *OSError*.

msvcrt.LK_UNLCK

Desbloquea los bytes especificados que han sido previamente bloqueados.

msvcrt.setmode (*fd, flags*)

Establece el modo traducción del final de línea del descriptor de un archivo *fd*. Si se establece como modo texto, *flags* debería ser *os.O_TEXT*; para establecerlo como modo binario, debería ser *os.O_BINARY*.

msvcrt.open_oshandle (*handle, flags*)

Crea un descriptor de archivo en el entorno de ejecución de C desde el manejador de archivo *handle*. El parámetro *flags* debe ser un OR bit a bit de *os.O_APPEND*, *os.O_RDONLY*, y *os.O_TEXT*. El descriptor de archivo retornado se puede utilizar como parámetro para *os.fdopen()* para crear un objeto archivo.

Lanza un *evento de auditoría* *msvcrt.open_oshandle* con los argumentos *handle, flags*.

msvcrt.get_oshandle (*fd*)

Retorna el manejador de archivo para un descriptor de archivo *fd*. Lanza una excepción *OSError* si *fd* no se reconoce.

Lanza un *evento de auditoría* *msvcrt.get_oshandle* con el argumento *fd*.

34.2.2 Consola E/S

`msvcrt.kbhit()`

Retorna `True` si hay una pulsación de tecla está esperando para ser leída.

`msvcrt.getch()`

Lee una pulsación de la tecla y retorna el carácter resultante como una cadena de caracteres de bytes. Nada se muestra en la consola. Esta llamada se bloqueará si una pulsación de la tecla aún no está disponible, pero no esperará a que se presione `Enter`. Si la tecla pulsada era una tecla de función especial, esto retornará `'\000'` o `'\xe0'`; la siguiente llamada retornará el código de la tecla pulsada. La pulsación de la tecla `:kbd:"Control-C"` no se puede leer con esta función.

`msvcrt.getwch()`

Variante de carácter amplio de `getch()`, retornando un valor Unicode.

`msvcrt.getche()`

Similar a la función `getch()`, pero la pulsación de la tecla se imprime si representa un carácter imprimible.

`msvcrt.getwche()`

Variante de carácter amplio de `getche()`, retornando un valor Unicode.

`msvcrt.putch(char)`

Imprime la cadena de caracteres de bytes `char` a la consola sin almacenamiento en buffer.

`msvcrt.putwch(unicode_char)`

Variante de carácter amplio de `putch()`, admitiendo un valor Unicode.

`msvcrt.ungetch(char)`

Provoca que la cadena de caracteres de bytes `char` sea «colocada de nuevo» en el buffer de la consola, será el siguiente carácter que lea la función `getch()` o `getche()`.

`msvcrt.ungetwch(unicode_char)`

Variante de carácter amplio de `ungetch()`, admitiendo un valor Unicode.

34.2.3 Otras funciones

`msvcrt.heapmin()`

Fuerza a la pila `malloc()` a que se limpie y retorne los bloques sin usar al sistema operativo. En el caso de que ocurriera algún fallo, se lanzaría una excepción `OSError`.

34.3 winreg — Acceso al registro de Windows

Estas funciones exponen la API de registro de Windows a Python. En lugar de utilizar un número entero como identificador de registro, se utiliza un *handle object* para garantizar que los identificadores se cierren correctamente, incluso si el programador se niega a cerrarlos explícitamente.

Distinto en la versión 3.3: Varias funciones de este módulo solían lanzar un `WindowsError`, que ahora es un alias de `OSError`.

34.3.1 Funciones

Este módulo ofrece las siguientes funciones:

`winreg.CloseKey(hkey)`

Cierra una clave de registro abierta previamente. El argumento *hkey* especifica una clave abierta previamente.

Nota: Si *hkey* no se cierra con este método (o mediante `hkey.Close()`), se cierra cuando Python destruye el objeto *hkey*.

`winreg.ConnectRegistry(computer_name, key)`

Establece una conexión con un identificador de registro predefinido en otra computadora y retorna un *handle object*.

computer_name es el nombre de la computadora remota, de la forma `r"\\computername"`. Si es `None`, se utiliza la computadora local.

key es el identificador predefinido al que conectarse.

El valor de retorno es el identificador de la llave abierta. Si la función falla, se lanza una excepción `OSError`.

Lanza un *auditing event* `winreg.ConnectRegistry` con argumentos *computer_name*, *key*.

Distinto en la versión 3.3: Ver *above*.

`winreg.CreateKey(key, sub_key)`

Crea o abre la clave especificada, retornando un *handle object*.

key es una clave ya abierta, o una de las predefinidas `HKEY_* constants`.

sub_key es una cadena de caracteres que nombra la clave que este método abre o crea.

Si *key* es una de las claves predefinidas, *sub_key* puede ser `None`. En ese caso, el identificador retornado es el mismo identificador de clave que se pasó a la función.

Si la clave ya existe, esta función abre la clave existente.

El valor de retorno es el identificador de la llave abierta. Si la función falla, se lanza una excepción `OSError`.

Lanza un *auditing event* `winreg.CreateKey` con los argumentos *key*, *sub_key*, *access*.

Lanza un *auditing event* `winreg.OpenKey/result` con el argumento *key*.

Distinto en la versión 3.3: Ver *above*.

`winreg.CreateKeyEx(key, sub_key, reserved=0, access=KEY_WRITE)`

Crea o abre la clave especificada, retornando un *handle object*.

key es una clave ya abierta, o una de las predefinidas `HKEY_* constants`.

sub_key es una cadena de caracteres que nombra la clave que este método abre o crea.

reserved es un número entero reservado y debe ser cero. El valor predeterminado es cero.

access es un número entero que especifica una máscara de acceso que describe el acceso de seguridad deseado para la clave. El valor predeterminado es `KEY_WRITE`. Ver *Access Rights* para otros valores permitidos.

Si *key* es una de las claves predefinidas, *sub_key* puede ser `None`. En ese caso, el identificador retornado es el mismo identificador de clave que se pasó a la función.

Si la clave ya existe, esta función abre la clave existente.

El valor de retorno es el identificador de la llave abierta. Si la función falla, se lanza una excepción `OSError`.

Lanza un *auditing event* `winreg.CreateKey` con los argumentos *key*, *sub_key*, *access*.

Lanza un *auditing event* `winreg.OpenKey/result` con el argumento *key*.

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Ver [above](#).

`winreg.DeleteKey(key, sub_key)`

Elimina la clave especificada.

key es una clave ya abierta, o una de las predefinidas [HKEY_* constants](#).

sub_key es una cadena de caracteres que debe ser una subclave de la clave identificada por el parámetro *key*. Este valor no debe ser `None`, y es posible que la clave no tenga subclaves.

Este método no puede eliminar claves con subclaves.

Si el método tiene éxito, se elimina toda la clave, incluidos todos sus valores. Si el método falla, se lanza una excepción [OSError](#).

Lanza un [auditing event](#) `winreg.DeleteKey` con los argumentos *key*, *sub_key*, *access*.

Distinto en la versión 3.3: Ver [above](#).

`winreg.DeleteKeyEx(key, sub_key, access=KEY_WOW64_64KEY, reserved=0)`

Elimina la clave especificada.

Nota: La función [DeleteKeyEx\(\)](#) se implementa con la función `RegDeleteKeyEx` de la API de Windows, que es específica de las versiones de Windows de 64 bits. Consulte la [RegDeleteKeyEx documentation](#).

key es una clave ya abierta, o una de las predefinidas [HKEY_* constants](#).

sub_key es una cadena de caracteres que debe ser una subclave de la clave identificada por el parámetro *key*. Este valor no debe ser `None`, y es posible que la clave no tenga subclaves.

reserved es un número entero reservado y debe ser cero. El valor predeterminado es cero.

access es un número entero que especifica una máscara de acceso que describe el acceso de seguridad deseado para la clave. El valor predeterminado es [KEY_WOW64_64KEY](#). Ver [Access Rights](#) para otros valores permitidos.

Este método no puede eliminar claves con subclaves.

Si el método tiene éxito, se elimina toda la clave, incluidos todos sus valores. Si el método falla, se lanza una excepción [OSError](#).

En versiones de Windows no compatibles, se lanza [NotImplementedError](#).

Lanza un [auditing event](#) `winreg.DeleteKey` con los argumentos *key*, *sub_key*, *access*.

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Ver [above](#).

`winreg.DeleteValue(key, value)`

Elimina un valor con nombre de una clave de registro.

key es una clave ya abierta, o una de las predefinidas [HKEY_* constants](#).

value es una cadena que identifica el valor a eliminar.

Lanza un [auditing event](#) `winreg.DeleteValue` con argumentos *key*, *value*.

`winreg.EnumKey(key, index)`

Enumera las subclaves de una clave de registro abierta y retorna una cadena de caracteres.

key es una clave ya abierta, o una de las predefinidas [HKEY_* constants](#).

index es un número entero que identifica el índice de la clave a recuperar.

La función recupera el nombre de una subclave cada vez que se llama. Normalmente se llama repetidamente hasta que se lanza una excepción [OSError](#), lo que indica que no hay más valores disponibles.

Lanza un [auditing event](#) `winreg.EnumKey` con argumentos *key*, *index*.

Distinto en la versión 3.3: Ver [above](#).

`winreg.EnumValue(key, index)`

Enumera los valores de una clave de registro abierta y retorna una tupla.

key es una clave ya abierta, o una de las predefinidas [HKEY_* constants](#).

index es un número entero que identifica el índice del valor a recuperar.

La función recupera el nombre de una subclave cada vez que se llama. Normalmente se llama repetidamente, hasta que se lanza una excepción [OSError](#), lo que indica que no hay más valores.

El resultado es una tupla de 3 elementos:

Índice	Significado
0	Una cadena de caracteres que identifica el nombre del valor
1	Un objeto que contiene los datos del valor y cuyo tipo depende del tipo de registro subyacente
2	Un número entero que identifica el tipo de datos de valor (consulte la tabla en los documentos para SetValueEx())

Lanza un *auditing event* `winreg.EnumValue` con argumentos *key*, *index*.

Distinto en la versión 3.3: Ver [above](#).

`winreg.ExpandEnvironmentStrings(str)`

Expande los marcadores de posición de la variable de entorno `%NAME%` en cadenas como [REG_EXPAND_SZ](#):

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

Lanza un *auditing event* `winreg.ExpandEnvironmentStrings` con el argumento *str*.

`winreg.FlushKey(key)`

Escribe todos los atributos de una clave en el registro.

key es una clave ya abierta, o una de las predefinidas [HKEY_* constants](#).

No es necesario llamar a [FlushKey\(\)](#) para cambiar una clave. Los cambios en el registro se descargan en el disco mediante el registro mediante su vaciador diferido. Los cambios en el registro también se vacían en el disco cuando se apaga el sistema. A diferencia de [CloseKey\(\)](#), el método [FlushKey\(\)](#) retorna solo cuando todos los datos se han escrito en el registro. Una aplicación solo debe llamar a [FlushKey\(\)](#) si requiere absoluta certeza de que los cambios de registro están en el disco.

Nota: Si no sabe si se requiere una llamada [FlushKey\(\)](#), probablemente no lo sea.

`winreg.LoadKey(key, sub_key, file_name)`

Crea una subclave bajo la clave especificada y almacena la información de registro de un archivo especificado en esa subclave.

key es un identificador retornado por [ConnectRegistry\(\)](#) o una de las constantes [HKEY_USERS](#) o [HKEY_LOCAL_MACHINE](#).

sub_key es una cadena de caracteres que identifica la subclave a cargar.

file_name es el nombre del archivo desde el que cargar los datos de registro. Este archivo debe haber sido creado con la función [SaveKey\(\)](#). En el sistema de archivos de la tabla de asignación de archivos (FAT), es posible que el nombre del archivo no tenga extensión.

Una llamada a [LoadKey\(\)](#) falla si el proceso de llamada no tiene el privilegio `SE_RESTORE_PRIVILEGE`. Tenga en cuenta que los privilegios son diferentes de los permisos; consulte la [RegLoadKey documentation](#) para obtener más detalles.

Si *key* es un identificador retornado por [ConnectRegistry\(\)](#), entonces la ruta especificada en *file_name* es relativa a la computadora remota.

Lanza un *auditing event* `winreg.LoadKey` con los argumentos `key`, `sub_key`, `file_name`.

`winreg.OpenKey` (`key`, `sub_key`, `reserved=0`, `access=KEY_READ`)

`winreg.OpenKeyEx` (`key`, `sub_key`, `reserved=0`, `access=KEY_READ`)

Abre la clave especificada, retornando a *handle object*.

`key` es una clave ya abierta, o una de las predefinidas *HKEY_* constants*.

`sub_key` es una cadena de caracteres que identifica la `sub_key` para abrir.

`reserved` es un número entero reservado y debe ser cero. El valor predeterminado es cero.

`access` es un número entero que especifica una máscara de acceso que describe el acceso de seguridad deseado para la clave. El valor predeterminado es *KEY_READ*. Ver *Access Rights* para otros valores permitidos.

El resultado es un nuevo identificador para la clave especificada.

Si la función falla, se lanza *OSError*.

Lanza un *auditing event* `winreg.OpenKey` con los argumentos `key`, `sub_key`, `access`.

Lanza un *auditing event* `winreg.OpenKey/result` con el argumento `key`.

Distinto en la versión 3.2: Permite el uso de argumentos con nombre.

Distinto en la versión 3.3: Ver *above*.

`winreg.QueryInfoKey` (`key`)

Retorna información sobre una clave, como una tupla.

`key` es una clave ya abierta, o una de las predefinidas *HKEY_* constants*.

El resultado es una tupla de 3 elementos:

Índice	Significado
0	Un número entero que indica el número de subclaves que tiene esta clave.
1	Un número entero que da el número de valores que tiene esta clave.
2	Un número entero que indica la última vez que se modificó la clave (si está disponible) como cientos de nanosegundos desde el 1 de enero de 1601.

Lanza un *auditing event* `winreg.QueryInfoKey` con el argumento `key`.

`winreg.QueryValue` (`key`, `sub_key`)

Recupera el valor sin nombre de una clave, como una cadena de caracteres.

`key` es una clave ya abierta, o una de las predefinidas *HKEY_* constants*.

`sub_key` es una cadena de caracteres que contiene el nombre de la subclave con la que está asociado el valor. Si este parámetro es `None` o está vacío, la función recupera el valor establecido por el método *SetValue()* para la clave identificada por `key`.

Los valores del registro tienen componentes de nombre, tipo y datos. Este método recupera los datos del primer valor de una clave que tiene un nombre `NULL`. Pero la llamada a la API subyacente no retorna el tipo, así que siempre use *QueryValueEx()* si es posible.

Lanza un *auditing event* `winreg.QueryValue` con los argumentos `key`, `sub_key`, `value_name`.

`winreg.QueryValueEx` (`key`, `value_name`)

Recupera el tipo y los datos de un nombre de valor especificado asociado con una clave de registro abierta.

`key` es una clave ya abierta, o una de las predefinidas *HKEY_* constants*.

`value_name` es una cadena de caracteres que indica el valor a consultar.

El resultado es una tupla de 2 elementos:

Índice	Significado
0	El valor del elemento de registro.
1	Un número entero que proporciona el tipo de registro para este valor (consulte la tabla en docs para <code>SetValueEx()</code>)

Lanza un *auditing event* `winreg.QueryValue` con los argumentos `key`, `sub_key`, `value_name`.

`winreg.SaveKey(key, file_name)`

Guarda la clave especificada y todas sus subclaves en el archivo especificado.

`key` es una clave ya abierta, o una de las predefinidas *HKEY_* constants*.

`file_name` es el nombre del archivo en el que se guardarán los datos del registro. Este archivo no puede existir ya. Si este nombre de archivo incluye una extensión, no se puede usar en sistemas de archivos de tabla de asignación de archivos (FAT) mediante el método `LoadKey()`.

Si `key` representa una clave en una computadora remota, la ruta descrita por `file_name` es relativa a la computadora remota. La persona que llama a este método debe poseer el privilegio de seguridad `SeBackupPrivilege`. Tenga en cuenta que los privilegios son diferentes a los permisos – consulte la documentación sobre conflictos entre derechos de usuario y permisos <<https://msdn.microsoft.com/en-us/library/ms724878%28v=VS.85%29.aspx>>__ para más detalles.

Esta función pasa `NULL` para `security_attributes` a la API.

Lanza un *auditing event* `winreg.SaveKey` con los argumentos `key`, `file_name`.

`winreg.SetValue(key, sub_key, type, value)`

Asocia un valor con una clave especificada.

`key` es una clave ya abierta, o una de las predefinidas *HKEY_* constants*.

`sub_key` es una cadena de caracteres que nombra la subclave con la que está asociado el valor.

`type` es un número entero que especifica el tipo de datos. Actualmente debe ser *REG_SZ*, lo que significa que solo se admiten cadenas de caracteres. Utilice la función: `SetValueEx()` para admitir otros tipos de datos.

`value` es una cadena de caracteres que especifica el nuevo valor.

Si la clave especificada por el parámetro `sub_key` no existe, la función `SetValue` la crea.

Las longitudes de los valores están limitadas por la memoria disponible. Los valores largos (más de 2048 bytes) deben almacenarse como archivos con los nombres de archivo almacenados en el registro de configuración. Esto ayuda a que el registro funcione de manera eficiente.

La clave identificada por el parámetro `key` debe haber sido abierta con acceso *KEY_SET_VALUE*.

Lanza un *auditing event* `winreg.SetValue` con argumentos `key`, `sub_key`, `type`, `value`.

`winreg.SetValueEx(key, value_name, reserved, type, value)`

Almacena datos en el campo de valor de una clave de registro abierta.

`key` es una clave ya abierta, o una de las predefinidas *HKEY_* constants*.

`value_name` es una cadena de caracteres que nombra la subclave con la que está asociado el valor.

`reserved` puede ser cualquier cosa — cero siempre se pasa a la API.

`type` es un número entero que especifica el tipo de datos. Consulte *Value Types* para los tipos disponibles.

`value` es una cadena de caracteres que especifica el nuevo valor.

Este método también puede establecer un valor adicional e información de tipo para la clave especificada. La clave identificada por el parámetro `key` debe haber sido abierta con acceso *KEY_SET_VALUE*.

Para abrir la clave, use los métodos `CreateKey()` o `OpenKey()`.

Las longitudes de los valores están limitadas por la memoria disponible. Los valores largos (más de 2048 bytes) deben almacenarse como archivos con los nombres de archivo almacenados en el registro de configuración. Esto ayuda a que el registro funcione de manera eficiente.

Lanza un *auditing event* `winreg.SetValue` con argumentos `key`, `sub_key`, `type`, `value`.

`winreg.DisableReflectionKey(key)`

Desactiva la reflexión del registro para los procesos de 32 bits que se ejecutan en un sistema operativo de 64 bits.

`key` es una clave ya abierta, o una de las predefinidas *HKEY_* constants*.

Generalmente lanzará *NotImplementedError* si se ejecuta en un sistema operativo de 32 bits.

Si la clave no está en la lista de reflexión, la función tiene éxito pero no tiene ningún efecto. La desactivación de la reflexión de una clave no afecta la reflexión de ninguna subclave.

Lanza un *auditing event* `winreg.DisableReflectionKey` con el argumento `key`.

`winreg.EnableReflectionKey(key)`

Restaura la reflexión del registro para la clave deshabilitada especificada.

`key` es una clave ya abierta, o una de las predefinidas *HKEY_* constants*.

Generalmente lanzará *NotImplementedError* si se ejecuta en un sistema operativo de 32 bits.

La restauración de la reflexión de una clave no afecta la reflexión de ninguna subclave.

Lanza un *auditing event* `winreg.EnableReflectionKey` con el argumento `key`.

`winreg.QueryReflectionKey(key)`

Determina el estado de reflexión para la clave especificada.

`key` es una clave ya abierta, o una de las predefinidas *HKEY_* constants*.

Retorna `True` si la reflexión está deshabilitada.

Generalmente lanzará *NotImplementedError* si se ejecuta en un sistema operativo de 32 bits.

Lanza un *auditing event* `winreg.QueryReflectionKey` con el argumento `key`.

34.3.2 Constantes

Las siguientes constantes están definidas para su uso en muchas funciones `_winreg`.

HKEY_* Constantes

`winreg.HKEY_CLASSES_ROOT`

Las entradas de registro subordinadas a esta clave definen tipos (o clases) de documentos y las propiedades asociadas con esos tipos. Las aplicaciones Shell y COM utilizan la información almacenada en esta clave.

`winreg.HKEY_CURRENT_USER`

Las entradas de registro subordinadas a esta clave definen las preferencias del usuario actual. Estas preferencias incluyen la configuración de variables de entorno, datos sobre grupos de programas, colores, impresoras, conexiones de red y preferencias de la aplicación.

`winreg.HKEY_LOCAL_MACHINE`

Las entradas de registro subordinadas a esta clave definen el estado físico de la computadora, incluidos los datos sobre el tipo de bus, la memoria del sistema y el hardware y software instalados.

`winreg.HKEY_USERS`

Las entradas de registro subordinadas a esta clave definen la configuración de usuario predeterminada para nuevos usuarios en la computadora local y la configuración de usuario para el usuario actual.

`winreg.HKEY_PERFORMANCE_DATA`

Las entradas de registro subordinadas a esta clave le permiten acceder a los datos de rendimiento. Los datos no se almacenan realmente en el registro; las funciones de registro hacen que el sistema recopile los datos de su fuente.

`winreg.HKEY_CURRENT_CONFIG`

Contiene información sobre el perfil de hardware actual del sistema informático local.

`winreg.HKEY_DYN_DATA`

Esta clave no se usa en versiones de Windows posteriores a la 98.

Access Rights

Para más información, ver [Registry Key Security and Access](#).

`winreg.KEY_ALL_ACCESS`

Combina los derechos de acceso `STANDARD_RIGHTS_REQUIRED`, `KEY_QUERY_VALUE`, `KEY_SET_VALUE`, `KEY_CREATE_SUB_KEY`, `KEY_ENUMERATE_SUB_KEYS`, `KEY_NOTIFY`, y `KEY_CREATE_LINK`.

`winreg.KEY_WRITE`

Combina los derechos de acceso `STANDARD_RIGHTS_WRITE`, `KEY_SET_VALUE`, y `KEY_CREATE_SUB_KEY`.

`winreg.KEY_READ`

Combina los valores `STANDARD_RIGHTS_READ`, `KEY_QUERY_VALUE`, `KEY_ENUMERATE_SUB_KEYS`, y `KEY_NOTIFY`.

`winreg.KEY_EXECUTE`

Equivalente a `KEY_READ`.

`winreg.KEY_QUERY_VALUE`

Requerido para consultar los valores de una clave de registro.

`winreg.KEY_SET_VALUE`

Requerido para crear, eliminar o establecer un valor de registro.

`winreg.KEY_CREATE_SUB_KEY`

Necesario para crear una subclave de una clave de registro.

`winreg.KEY_ENUMERATE_SUB_KEYS`

Requerido para enumerar las subclaves de una clave de registro.

`winreg.KEY_NOTIFY`

Requerido para solicitar notificaciones de cambio para una clave de registro o para subclaves de una clave de registro.

`winreg.KEY_CREATE_LINK`

Reservado para uso del sistema.

Específico de 64 bits

Para más información, ver [Accessing an Alternate Registry View](#).

`winreg.KEY_WOW64_64KEY`

Indica que una aplicación en Windows de 64 bits debería funcionar en la vista de registro de 64 bits.

`winreg.KEY_WOW64_32KEY`

Indica que una aplicación en Windows de 64 bits debería funcionar en la vista de registro de 32 bits.

Tipos de valor

Para más información, ver [Registry Value Types](#).

`winreg.REG_BINARY`

Datos binarios en cualquier forma.

`winreg.REG_DWORD`

Número de 32 bits.

`winreg.REG_DWORD_LITTLE_ENDIAN`

Un número de 32 bits en formato little-endian. Equivalente a [REG_DWORD](#).

`winreg.REG_DWORD_BIG_ENDIAN`

Un número de 32 bits en formato big-endian.

`winreg.REG_EXPAND_SZ`

Cadena de caracteres terminada en nulo que contiene referencias a variables de entorno (%PATH%).

`winreg.REG_LINK`

Un enlace simbólico Unicode.

`winreg.REG_MULTI_SZ`

Una secuencia de cadenas de caracteres terminadas en nulo, terminadas por dos caracteres nulos. (Python maneja esta terminación automáticamente).

`winreg.REG_NONE`

Sin tipo de valor definido.

`winreg.REG_QWORD`

Un número de 64 bits.

Nuevo en la versión 3.6.

`winreg.REG_QWORD_LITTLE_ENDIAN`

Un número de 64 bits en formato little-endian. Equivalente a [REG_QWORD](#).

Nuevo en la versión 3.6.

`winreg.REG_RESOURCE_LIST`

Una lista de recursos de controladores de dispositivo.

`winreg.REG_FULL_RESOURCE_DESCRIPTOR`

Una configuración de hardware.

`winreg.REG_RESOURCE_REQUIREMENTS_LIST`

Una lista de recursos de hardware.

`winreg.REG_SZ`

Una cadena de caracteres terminada en nulo.

34.3.3 Objetos de control del registro

Este objeto envuelve un objeto HKEY de Windows y lo cierra automáticamente cuando se destruye. Para garantizar la limpieza, puede llamar al método `Close()` en el objeto, o a la función `CloseKey()`.

Todas las funciones de registro de este módulo retornan uno de estos objetos.

Todas las funciones de registro de este módulo que aceptan un objeto identificador también aceptan un número entero, sin embargo, se recomienda el uso del objeto identificador.

Los objetos de control proporcionan semántica para `__bool__()` – así

```
if handle:
    print("Yes")
```

imprimirá `Yes` si el controlador es válido actualmente (no se ha cerrado o desprendido).

El objeto también admite la semántica de comparación, por lo que los objetos de identificador se compararán con verdadero si ambos hacen referencia al mismo valor de identificador de Windows subyacente.

Los objetos de identificador se pueden convertir a un número entero (por ejemplo, usando la función incorporada `int()` function), en cuyo caso se retorna el valor de identificador de Windows subyacente. También puede usar el método `Detach()` para retornar el identificador de enteros y también desconectar el identificador de Windows del objeto identificador.

`PyHKEY.Close()`

Cierra el identificador de Windows subyacente.

Si el controlador ya está cerrado, no se lanza ningún error.

`PyHKEY.Detach()`

Separa el identificador de Windows del objeto identificador.

El resultado es un número entero que contiene el valor del identificador antes de que se separe. Si el controlador ya está separado o cerrado, esto retornará cero.

Después de llamar a esta función, el identificador se invalida efectivamente, pero el identificador no se cierra. Llamaría a esta función cuando necesite que el identificador Win32 subyacente exista más allá de la vida útil del objeto identificador.

Lanza un *auditing event* `winreg.PyHKEY.Detach` con el argumento `key`.

`PyHKEY.__enter__()`

`PyHKEY.__exit__(*exc_info)`

El objeto HKEY implementa `__enter__()` y `__exit__()` y, por lo tanto, admite el protocolo de contexto para la declaración `with`:

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ... # work with key
```

cerrará automáticamente `key` cuando el control abandone el bloque `with`.

34.4 :mod:"winsound" — Interfaz de reproducción de sonido para Windows

El modulo `winsound` permite acceder a la maquinaria básica de reproducción de sonidos proporcionada por la plataformas Windows. Incluye funciones y varias constantes.

`winsound.Beep(frequency, duration)`

Hace sonar el altavoz del PC. El parámetro *frequency* especifica la frecuencia, en hercio (hz), de la señal de audio y debe estar en el rango de 37 a 32.767 hz. El parámetro *duration* especifica el numero de milisegundo de duración de la señal de audio. Si el sistema no puede hacer sonar el altavoz, se lanza `RuntimeError`.

`winsound.PlaySound(sound, flags)`

Llama a la función responsable de `PlaySound()` desde el API de la plataforma. El parámetro *sound* puede ser un nombre de archivo, un alias de sonido del sistema, datos de audio como un *bytes-like object*, o `None`. Su interpretación depende del valor de *flags*, que puede ser una combinación Ored de las constantes descritas a continuación. Si el parámetro de *sound* es `None`, cualquier sonido de forma de onda que se esté reproduciendo en ese momento se detiene. Si el sistema indica un error, se lanza `RuntimeError`.

`winsound.MessageBeep(type=MB_OK)`

Llama a la función responsable de `MensajeBeep()` de la API de la plataforma. Esto reproduce un sonido como se especifica en el registro. El argumento *type* especifica qué sonido se reproduce; los valores posibles son `-1`, `MB_ICONASTERISK`, `MB_ICONEXCLAMATION`, `MB_ICONHAND`, `MB_ICONQUESTION`, y `MB_OK`, todos descritos a continuación. El valor «1» produce un «simple pitido»; este es el último recurso si un sonido no puede ser reproducido de otra manera. Si el sistema indica un error, se lanza `RuntimeError`.

`winsound.SND_FILENAME`

El parámetro *sound* es el nombre de un archivo WAV. No lo uses con `SND_ALIAS`.

`winsound.SND_ALIAS`

El parámetro *sound* es un nombre de asociación de sonido del registro. Si el registro no contiene tal nombre, reproduce el sonido por defecto del sistema a menos que `SND_NODEFAULT` también se especifique. Si no se registra ningún sonido por defecto, se lanza `RuntimeError`. No lo uses con `SND_FILENAME`.

Todos los sistemas Win32 soportan al menos lo siguiente; la mayoría de los sistemas soportan muchos más:

<code>PlaySound()</code> <i>name</i>	Panel de control correspondiente nombre (<i>name</i>) del sonido
'SystemAsterisk'	Asterisco
'SystemExclamation'	Exclamación
'SystemExit'	Salir de Windows
'SystemHand'	Parada crítica
'SystemQuestion'	Pregunta

Por ejemplo:

```
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`

Reproducir el sonido repetidamente. El flag `SND_ASYNC` también debe ser usada para evitar el bloqueo. No puede ser usada con `SND_MEMORY`.

`winsound.SND_MEMORY`

El parámetro *sound* de `PlaySound()` es una imagen de memoria de un archivo WAV, como un *bytes-like object*.

Nota: Este módulo no admite la reproducción desde una imagen de memoria de forma sincrónica, por lo que una combinación de este indicador y `SND_ASYNC` se lanza `RuntimeError`.

`winsound.SND_PURGE`

Detiene la reproducción de todas las instancias de un sonido específico.

Nota: Esta flag no esta soportado en las plataformas modernas de Windows.

`winsound.SND_ASYNC`

Retorna inmediatamente, permitiendo que los sonidos se reproduzcan asincrónicamente.

`winsound.SND_NODEFAULT`

Si no se puede encontrar el audio especificado, no reproduce el sonido predeterminado del sistema.

`winsound.SND_NOSTOP`

No interrumpe los sonidos que se están reproduciendo.

`winsound.SND_NOWAIT`

Retorna inmediatamente en caso de que el controlador de sonido está ocupado.

Nota: Esta flag no esta soportado en las plataformas modernas de Windows.

`winsound.MB_ICONASTERISK`

Reproduce el sonido `SystemDefault`.

`winsound.MB_ICONEXCLAMATION`

Reproduce el sonido `SystemExclamation`.

`winsound.MB_ICONHAND`

Reproduce el sonido `SystemHand`.

`winsound.MB_ICONQUESTION`

Reproduce el sonido `SystemQuestion`.

`winsound.MB_OK`

Reproduce el sonido `SystemDefault`.

Servicios específicos de Unix

Los módulos descritos en este capítulo proporcionan interfaces para las características exclusivas del sistema operativo Unix o, en algunos casos, para algunas o muchas variantes del mismo. He aquí una visión general:

35.1 `posix` — Las llamadas más comunes al sistema POSIX

Este módulo proporciona acceso a la funcionalidad del sistema operativo que está estandarizada por el estándar C y el estándar POSIX (una interfaz Unix finamente disfrazada).

No importe este módulo directamente. En su lugar, importe el módulo `os`, que proporciona una versión *portable* de esta interfaz. En Unix, el módulo `os` proporciona un superconjunto de la interfaz `posix`. En sistemas operativos que no son Unix, el módulo `posix` no está disponible, pero un subconjunto siempre está disponible a través de la interfaz `os`. Una vez que se importa `os`, no hay penalización de rendimiento en su uso en lugar de `posix`. Además, `os` proporciona algunas funciones adicionales, como llamar automáticamente a `putenv()` cuando se cambia una entrada en `os.environ`.

Los errores se notifican como excepciones; las excepciones habituales se proporcionan para los errores de tipo, mientras que los errores notificados por las llamadas del sistema lanzan `OSError`.

35.1.1 Soporte de archivos grandes

Varios sistemas operativos (incluidos AIX, HP-UX, Irix y Solaris) proporcionan compatibilidad con archivos de más de 2 GiB de un modelo de programación C donde `int` y `long` son valores de 32 bits. Esto se logra normalmente definiendo el tamaño relevante y los tipos de desplazamiento como valores de 64 bits. Tales archivos se conocen a veces como *large files*.

La compatibilidad con archivos grandes está habilitada en Python cuando el tamaño de un `off_t` es mayor que un `long` y `long long` es al menos tan grande como `off_t`. Puede ser necesario configurar y compilar Python con ciertos indicadores del compilador para habilitar este modo. Por ejemplo, está habilitado de forma predeterminada con las versiones recientes de Irix, pero con Solaris 2.6 y 2.7 debe hacer algo como:

```
CFLAGS="-`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \  
./configure
```

En sistemas Linux con capacidad para archivos grandes, esto podría funcionar:

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \  
./configure
```

35.1.2 Contenido notable del módulo

Además de muchas funciones descritas en la documentación del módulo `os`, `posix` define el siguiente elemento de datos:

`posix.environ`

Diccionario que representa el entorno de cadena en el momento en que se inició el intérprete. Las claves y los valores son bytes en Unix y str en Windows. Por ejemplo, `environ[b'HOME']` (`environ['HOME']` en Windows) es el nombre de ruta de acceso de su directorio principal, equivalente a `getenv("HOME")` en C.

La modificación de este diccionario no afecta al entorno de cadena que transmite `execv()`, `popen()` o `system()`; si necesita cambiar el entorno, pase `environ` a `execve()` o agregue asignaciones variables y declaraciones de exportación a la cadena de comandos para `system()` o `popen()`.

Distinto en la versión 3.2: En Unix, las claves y los valores son bytes.

Nota: El módulo `os` proporciona una implementación alternativa de `environ` que actualiza el entorno en la modificación. Tenga en cuenta también que la actualización `os.environ` hará que este diccionario sea obsoleto. El uso de la versión del módulo `os` de esto se recomienda sobre el acceso directo al módulo `posix`.

35.2 pwd — La base de datos de contraseñas

Este módulo proporciona acceso a la base de datos de cuentas de usuario y contraseñas de Unix. Está disponible en todas las versiones de Unix.

Las entradas de la base de datos de contraseñas se reportan como un objeto de tipo tupla, cuyos atributos corresponden a los miembros de la estructura `passwd` (campo Atributo abajo, ver `<pwd.h>`):

Índice	Atributo	Significado
0	<code>pw_name</code>	Nombre de usuario
1	<code>pw_passwd</code>	Contraseña encriptada opcional
2	<code>pw_uid</code>	Identificación numérica de usuario
3	<code>pw_gid</code>	Identificación del grupo numérico
4	<code>pw_gecos</code>	Nombre de usuario o campo de comentarios
5	<code>pw_dir</code>	El directorio <i>home</i> del usuario
6	<code>pw_shell</code>	Intérprete de comandos de usuario

Los elementos `uid` y `gid` son enteros, todos los demás son cadenas. `KeyError` se lanza si la entrada pedida no se encuentra.

Nota: 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.

Define los siguientes elementos:

`pwd.getpwuid (uid)`

Retorna la entrada de la base de datos de contraseñas para el ID de usuario numérico dado.

`pwd.getpwnam (name)`

Retorna la entrada de la base de datos de contraseñas para el nombre de usuario dado.

`pwd.getpwall ()`

Retorna una lista de todas las entradas de la base de datos de contraseñas disponibles, en orden arbitrario.

Ver también:

Módulo `grp` Una interfaz para la base de datos de grupos, similar a esta.

Módulo `spwd` An interface to the shadow password database, similar to this.

35.3 spwd — La base de datos de contraseñas ocultas

Este módulo proporciona acceso a la base de datos de contraseñas ocultas de Unix. Está disponible en varias versiones de Unix.

Debe tener suficientes privilegios para acceder a la base de datos de contraseñas ocultas (esto generalmente significa que debe ser root).

Las entradas de la base de datos de contraseñas ocultas se informan como un objeto similar a una tupla, cuyos atributos corresponden a los miembros de la estructura `spwd` (campo de atributo a continuación, consulte `<shadow.h>`):

Índice	Atributo	Significado
0	<code>sp_namp</code>	Nombre de inicio de sesión
1	<code>sp_pwdp</code>	Contraseña encriptada
2	<code>sp_lstchg</code>	Fecha del último cambio
3	<code>sp_min</code>	Número mínimo de días entre cambios
4	<code>sp_max</code>	Número máximo de días entre cambios
5	<code>sp_warn</code>	Número de días antes de que expire la contraseña para advertir al usuario sobre ello
6	<code>sp_inact</code>	Número de días desde que caduca la contraseña hasta que se deshabilita la cuenta
7	<code>sp_expire</code>	Número de días desde 1970-01-01 cuando expira la cuenta
8	<code>sp_flag</code>	Reservado

Los elementos `sp_namp` y `sp_pwdp` son cadenas, todos los demás son números enteros. Se lanza `KeyError` si no se puede encontrar la entrada solicitada.

Se definen las siguientes funciones:

`spwd.getspnam (name)`

Retorna la entrada de la base de datos de contraseñas ocultas para el nombre de usuario especificado.

Distinto en la versión 3.6: Lanza un `PermissionError` en vez de `KeyError` si el usuario no tiene privilegios.

`spwd.getspall ()`

Retorna una lista de todas las entradas de la base de datos de contraseñas ocultas disponibles, en orden arbitrario.

Ver también:

Módulo `grp` Una interfaz para la base de datos del grupo, similar a esta.

Módulo `pwd` Una interfaz para la base de datos de contraseñas normal, similar a esta.

35.4 grp — La base de datos de grupo

Este módulo proporciona acceso a la base de datos del grupo Unix. Está disponible en todas las versiones de Unix.

Las entradas de base de datos de grupo se notifican como un objeto similar a una tupla, cuyos atributos corresponden a los miembros de la estructura `group` (campo atributo a continuación, véase `<pwd.h>`):

Índice	Atributo	Significado
0	<code>gr_name</code>	el nombre del grupo
1	<code>gr_passwd</code>	El grupo contraseña (encriptado); usualmente vacío
2	<code>gr_gid</code>	el grupo ID numérico
3	<code>gr_mem</code>	todos los nombres de usuario de los miembros del grupo

El *gid* es un número entero, el nombre y la contraseña son cadenas y la lista de miembros es una lista de cadenas. (Tenga en cuenta que la mayoría de los usuarios no se enumeran explícitamente como miembros del grupo en el que se encuentran de acuerdo con la base de datos de contraseñas. Consulte ambas bases de datos para obtener información completa sobre la membresía. También tenga en cuenta que un “gr_name” que comienza con un + o - es probable que sea una referencia de YP/NIS y puede que no sea accesible vía `getgrnam()` o `getgrgid()`.)

Define los siguientes elementos:

`grp.getgrgid(gid)`

Retorna la entrada de la base de datos del grupo para el ID de grupo numérico dado. Se genera `KeyError` si no se puede encontrar la entrada solicitada.

Obsoleto desde la versión 3.6: Desde Python 3.6, la compatibilidad con argumentos no enteros como flotantes o cadenas en `getgrgid()` está obsoleta.

`grp.getgrnam(name)`

Retorna la entrada de la base de datos del grupo para el nombre de grupo dado. Se genera `KeyError` si no se puede encontrar la entrada solicitada.

`grp.getgrall()`

Retorna una lista de todas las entradas de grupo disponibles, en orden arbitrario.

Ver también:

Módulo `pwd` Una interfaz para la base de datos de usuarios, similar a esta.

Módulo `spwd` Una interfaz para la base de datos de contraseñas ocultas, similar a esta.

35.5 crypt — Función para verificar contraseñas Unix

Código fuente: [Lib/crypt.py](#)

Este módulo implementa una interfaz para la rutina `crypt(3)`, el cual es una función hash unidireccional basada en un algoritmo DES modificado; consulte la página del manual de Unix para obtener más detalles. Los posibles usos incluyen el almacenamiento de contraseñas cifradas para que puedas verificar las contraseñas sin almacenar la contraseña real o intentar descifrar contraseñas Unix con un diccionario.

Tenga en cuenta que el comportamiento de este módulo depende en la implementación real de la rutina `crypt(3)` en el sistema en ejecución. Por lo tanto, cualquier extensión disponible en la implementación actual también estará disponible en este módulo.

Disponibilidad: Unix. No disponible en VxWorks.

35.5.1 Métodos de *hashing*

Nuevo en la versión 3.3.

El módulo `crypt` define la lista de métodos de cifrado (no todos los métodos están disponibles en todas las plataformas):

`crypt.METHOD_SHA512`

Un método de formato modular *crypt* con *salt* de 16 caracteres y hash de 86 caracteres basado en la función hash SHA-512. Este es el método más fuerte.

`crypt.METHOD_SHA256`

Otro método de formato modular *crypt* con *salt* de 16 caracteres y hash de 43 caracteres basado en la función hash SHA-256.

`crypt.METHOD_BLOWFISH`

Otro método de formato modular *crypt* con *salt* de 22 caracteres y hash de 31 caracteres basado en el cifrado Blowfish.

Nuevo en la versión 3.7.

`crypt.METHOD_MD5`

Otro método de formato modular *crypt* con *salt* de 8 caracteres y hash de 22 caracteres basado en la función hash MD5.

`crypt.METHOD_CRYPT`

El método tradicional con un *salt* de 2 caracteres y hash de 13 caracteres. Este es el método más débil.

35.5.2 Atributos del módulo

Nuevo en la versión 3.3.

`crypt.methods`

Una lista de algoritmos hash de contraseña disponibles, como objetos `crypt.METHOD_*`. Esta lista está ordenada de la más fuerte a la más débil.

35.5.3 Funciones del módulo

El módulo `crypt` define las siguientes funciones:

`crypt.crypt(word, salt=None)`

word normalmente será la contraseña de un usuario tal como se escribe en un prompt o en una interfaz gráfica. El *salt* opcional es una cadena retornada por `mk salt()`, uno de los valores `crypt.METHOD_*` (aunque no todos pueden estar disponibles en todas las plataformas), o una contraseña completa encriptada que incluye *salt*, como lo retorna esta función. Si no se proporciona *salt*, se utilizará el método más fuerte (como lo retornado por `methods()`).

La verificación de una contraseña generalmente se hace pasando la contraseña de texto plano como *word* y los resultados completos de una llamada anterior a `crypt()`, que debería ser igual a los resultados de esta llamada.

salt (ya sea una cadena aleatoria de 2 o 16 caracteres, posiblemente con el prefijo `$digit$` para indicar el método) que se utilizará para perturbar el algoritmo de cifrado. Los caracteres en *salt* deben estar en el conjunto `[./a-zA-Z0-9]`, con la excepción del formato modular *crypt* que antepone un `$digit$`.

Retorna una contraseña con hash como una cadena, que estará compuesta por caracteres del mismo alfabeto que *salt*.

Dado que algunas extensiones de `crypt(3)` permiten diferentes valores, con diferentes tamaños en *salt*, se recomienda utilizar la contraseña encriptada completa como *salt* al buscar una contraseña.

Distinto en la versión 3.3: Acepta los valores `crypt.METHOD_*` además de las cadenas para *salt*.

`crypt.mksalt` (*method=None*, *, *rounds=None*)

Retorna un *salt* generado aleatoriamente del método especificado. Si no se proporciona ningún método (*method*), se utiliza el método mas fuerte disponible según lo retornado por `methods()`.

El valor de retorno es una cadena adecuada para pasar como argumento *salt* a `crypt()`.

rounds especifica el número de rondas para `METHOD_SHA256`, `METHOD_SHA512` y `METHOD_BLOWFISH`. Para `METHOD_SHA256` y `METHOD_SHA512` debe ser un entero entre 1000 y 999_999_999, el valor predeterminado es 5000. Para `METHOD_BLOWFISH` debe ser una potencia de dos entre 16 (2^4) y 2_147_483_648 (2^{31}), el valor predeterminado es 4096 (2^{12}).

Nuevo en la versión 3.3.

Distinto en la versión 3.7: Se agregó el parámetro *rounds*.

35.5.4 Ejemplos

Un simple ejemplo que ilustra el uso típico (se necesita una operación de comparación de tiempo constante para limitar la exposición a los ataques de tiempo. `hmac.compare_digest()` es adecuado para este propósito):

```
import pwd
import crypt
import getpass
from hmac import compare_digest as compare_hash

def login():
    username = input('Python login: ')
    cryptpasswd = pwd.getpwnam(username)[1]
    if cryptpasswd:
        if cryptpasswd == 'x' or cryptpasswd == '*':
            raise ValueError('no support for shadow passwords')
        cleartext = getpass.getpass()
        return compare_hash(crypt.crypt(cleartext, cryptpasswd), cryptpasswd)
    else:
        return True
```

Para generar un hash de una contraseña utilizando el método más fuerte disponible y compararlo con el original:

```
import crypt
from hmac import compare_digest as compare_hash

hashed = crypt.crypt(plaintext)
if not compare_hash(hashed, crypt.crypt(plaintext, hashed)):
    raise ValueError("hashed version doesn't validate against original")
```

35.6 `termios` —Control tty estilo POSIX

Este módulo proporciona una interfaz para las llamadas POSIX para el control de E/S (entrada/salida) tty. Para obtener una descripción completa de estas llamadas, consulte `termios` (3) Página de manual de Unix. Solo está disponible para aquellas versiones de Unix que admitan el control de E/S tty estilo POSIX `termios` configurado durante la instalación.

Todas las funciones de este módulo toman un descriptor de archivo *fd* como primer argumento. Puede ser un descriptor de archivo entero, como el que retorna `sys.stdin.fileno()`, o un *file object*, como el propio `sys.stdin`.

Este módulo también define todas las constantes necesarias para trabajar con las funciones proporcionadas aquí; tienen el mismo nombre que sus contrapartes en C. Consulte la documentación de su sistema para obtener más información sobre el uso de estas interfaces de control de terminal.

El módulo define las siguientes funciones:

`termios.tcgetattr(fd)`

Retorna una lista que contiene los atributos tty para el descriptor de archivo *fd*, como se muestra a continuación: `[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]` donde *cc* es una lista de los caracteres especiales tty (cada una es una cadena de longitud 1, excepto los elementos con índices VMIN and VTIME, que son números enteros cuando se definen estos campos). La interpretación de las banderas y las velocidades, así como la indexación en el arreglo *cc*, debe realizarse utilizando las constantes simbólicas definidas en el módulo `termios`.

`termios.tcsetattr(fd, when, attributes)`

Establezca los atributos tty para el descriptor de archivo *fd* de los *atributos*, que es una lista como la retornada por `tcgetattr()`. El argumento *when* determina cuándo se cambian los atributos: TCSANOW para cambiar inmediatamente, TCSADRAIN para cambiar después de transmitir toda la salida en cola, o TCSAFLUSH para cambiar después de transmitir toda la salida en cola y descartar toda la entrada en cola.

`termios.tcsendbreak(fd, duration)`

Envíe una pausa en el descriptor de archivo *fd*. Una *duración* cero envía una pausa de 0.25 a 0.5 segundos; una *duración* distinta de cero tiene un significado dependiente del sistema.

`termios.tcdrain(fd)`

Espere hasta que se haya transmitido toda la salida escrita en el descriptor de archivo *fd*.

`termios.tcflush(fd, queue)`

Descartar datos en cola en el descriptor de archivo *fd*. El selector *queue* especifica qué cola: TCIFLUSH para la cola de entrada, TCOFLUSH para la cola de salida, o TCIOFLUSH para ambas colas.

`termios.tcflow(fd, action)`

Suspender o reanudar la entrada o salida en el descriptor de archivo *fd*. El argumento *action* puede ser TCOOFF para suspender la salida, TCOON para reiniciar la salida, TCIOFF para suspender la entrada, o TCION para reiniciar la entrada.

Ver también:

Módulo `tty` Funciones de conveniencia para operaciones comunes de control de terminal.

35.6.1 Ejemplo

Aquí hay una función que solicita una contraseña con el eco desactivado. Tenga en cuenta la técnica utilizando una llamada separada `tcgetattr()` y una sentencia `try ... finally` para asegurarse de que los antiguos atributos tty se restauran exactamente sin importar lo que suceda:

```
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 = input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

35.7 `tty` — Funciones de control de terminal

Código fuente: [Lib/tty.py](#)

El módulo `tty` define funciones para poner la `tty` en los modos `cbreak` y `raw`.

Dado que requiere el módulo `termios`, solamente funciona en Unix.

El módulo `tty` define las siguientes funciones:

`tty.setraw(fd, when=termios.TCSAFLUSH)`

Cambia el modo del descriptor de archivo `fd` a `raw`. Si se omite `when`, el valor por defecto es `termios.TCSAFLUSH`, que se pasa a `termios.tcsetattr()`.

`tty.setcbreak(fd, when=termios.TCSAFLUSH)`

Cambia el modo del descriptor de archivo `fd` a `cbreak`. Si se omite `when`, el valor por defecto es `termios.TCSAFLUSH`, que se pasa a `termios.tcsetattr()`.

Ver también:

Módulo `termios` Interfaz de control de la terminal de bajo nivel.

35.8 `pty` — Utilidades para Pseudo-terminal

Código Fuente: [Lib/pty.py](#)

El módulo `pty` define las operaciones para manejar el concepto de pseudo-terminal: iniciar otro proceso para poder escribir y leer desde su propia terminal mediante programación.

Debido a que el manejo de una pseudo-terminal depende en gran medida de la plataforma, este código es solo para Linux (se supone que el código de Linux funciona para otras plataformas, pero este aún no se ha probado)

El modulo `pty` define las siguientes funciones:

`pty.fork()`

Bifurcación. Conectar en su propia terminal (terminal hijo) una pseudo-terminal. El valor de retorno es `(pid, fd)`. Tener en cuenta que la terminal hijo tiene como valor `pid 0` y `fd` es *invalid*. El valor de retorno del padre es el `pid` del hijo, y `fd` es un descriptor de archivo conectado a la terminal hijo (también a la salida y entrada estándar de la terminal hijo)

`pty.openpty()`

Abre un nuevo par de pseudo-terminales, usando `os.openpty()`, o código de emulación para sistemas genéricos de Unix. Retorna un par de descriptores de archivo (`master, slave`), para el `master` y el `slave` respectivamente.

`pty.spawn(argv[, master_read[, stdin_read]])`

Genera un proceso conectado a su terminal con el io estándar del proceso actual. Esto se usa a frecuentemente para confundir programas que insisten en leer desde la terminal de control. Se espera que el proceso generado detrás de `pty` sea finalizado y cuando lo haga `spawn` se retornará.

Las funciones `master_read` y `stdin_read` se les envía como parámetro un descriptor de archivo y siempre deben retornar una cadena de bytes. A fin de que se obligue a `spawn` a retornar antes que el proceso hijo salga se debe lanzar un `OSError`.

La implementación predeterminada para ambas funciones retornará hasta 1024 bytes cada vez que se llamen. El dato retornado de `master_read` se pasa al descriptor de archivo maestro para leer la salida del proceso hijo, y `stdin_read` pasa el descriptor de archivo 0, para leer desde la entrada del proceso padre.

Retornando una cadena de bytes vacía de cualquier llamado es interpretado como una condición de fin de archivo (EOF), y el llamado no se realizará después de eso. Si `stdin_read` retorna EOF la terminal de control ya

no puede comunicarse con el proceso padre o el proceso hijo. A menos que el proceso hijo se cierre sin ninguna entrada *spawn* se repetirá para siempre. Si *master_read* retorna EOF se produce el mismo comportamiento (al menos en Linux)

Si ambas retrollamadas retornan EOF entonces *spawn* probablemente nunca retorne algo, a menos que *select* entregue un error en su plataforma cuando pasan tres listas vacías. Esto es un error documentado en [issue 26228](#).

Lanza un *evento de auditoria* `pty.spawn` con el argumento `argv`.

Distinto en la versión 3.4: *spawn()* ahora retorna el valor de estado de *os.waitpid()* para los procesos hijos.

35.8.1 Ejemplo

El siguiente programa actúa como el comando de Unix *script(1)*, usando una pseudo-terminal para registrar todas las entradas y salidas de una sesión en «typescript».

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true')
parser.add_argument('-p', dest='use_python', action='store_true')
parser.add_argument('filename', nargs='?', default='typescript')
options = parser.parse_args()

shell = sys.executable if options.use_python else os.environ.get('SHELL', 'sh')
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime()).encode())

    pty.spawn(shell, read)

    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```

35.9 fcntl — Las llamadas a sistema fcntl y ioctl

Este módulo realiza control de archivos y control de E/S en descriptores de ficheros. Es una interfaz para las rutinas de Unix *fcntl()* y *ioctl()*. Para una completa descripción de estas llamadas, ver las páginas del manual de Unix *fcntl(2)* y *ioctl(2)*.

Todas las funciones de este módulo toman un descriptor de fichero *fd* como su primer argumento. Puede ser un descriptor de fichero entero, como el retornado por *sys.stdin.fileno()*, o un objeto *io.IOBase*, como *sys.stdin*, que proporciona un *fileno()* que retornan un descriptor de fichero original.

Distinto en la versión 3.3: Las operaciones en este módulo solían lanzar un `IOError` donde ahora lanzan un `OSError`.

Distinto en la versión 3.8: El módulo `fcntl` ahora contiene las constantes `F_ADD_SEALS`, `F_GET_SEALS`, y `F_SEAL_*` para sellar los descriptores de fichero `os.memfd_create()`.

El módulo define las siguientes funciones:

`fcntl.fcntl(fd, cmd, arg=0)`

Realice la operación `cmd` en el descriptor de fichero `fd` (los objetos de fichero que proporcionan un método `fileno()` también son aceptados). Los valores utilizados para `cmd` dependen del sistema operativo y están disponibles como constantes en el módulo `fcntl`, utilizando los mismos nombres que se utilizan en los archivos de cabecera C relevantes. El argumento `arg` puede ser un valor entero o un objeto `bytes`. Con un valor entero, el valor retorno de esta función es el valor entero retornado por la llamada en C `fcntl()`. Cuando el argumento son bytes representa una estructura binaria, e.g. creada por `struct.pack()`. Los datos binarios se copian en un búfer cuya dirección se pasa a la llamada en C `:fcntl()`. El valor de retorno después de una llamada correcta es el contenido del búfer, convertido en un objeto `bytes`. La longitud del objeto retornado será la misma que la longitud del argumento `arg`. Esta longitud está limitada a 1024 bytes. Si la información retornada en el búfer por el sistema operativo es mayor que 1024 bytes, lo más probable es que se produzca una violación de segmento o a una corrupción de datos más sutil.

Si se produce un error en `fcntl()`, se lanza un `OSError`.

Lanza un `auditing event` `fcntl.fcntl` con argumentos `fd`, `cmd`, `arg`.

`fcntl.ioctl(fd, request, arg=0, mutate_flag=True)`

Esta función es idéntica a la función `fcntl()`, excepto por el manejo de los argumentos que es aún más complicado.

El parámetro `request` se encuentra limitado a valores que encajen en 32-bits. Se pueden encontrar constantes adicionales de interés para usar como argumento `request` en el módulo `termios`, con los mismos nombres que se usan en los archivos de cabecera C relevantes.

El parámetro `arg` puede ser un entero, un objeto que admita una interfaz de búfer de solo lectura (como `bytes`) o un objeto que admita una interfaz de búfer de lectura-escritura (como: clase `bytearray`).

En todos los casos excepto en el último, el comportamiento es el de la función `fcntl()`.

Si se pasa un búfer mutable, el comportamiento estará determinado por el valor del parámetro `mutate_flag`.

Si es falso, la mutabilidad del búfer se ignorará y el comportamiento será como el de un búfer de solo lectura, excepto por el límite de 1024 bytes mencionado arriba, que será evitado – siempre que el búfer que pase sea al menos tan largo como el sistema operativo quiera colocar allí, las cosas deberían funcionar.

Si `mutate_flag` es verdadero (valor predeterminado), entonces el búfer se pasa (en efecto) a la llamada al sistema subyacente `ioctl()`, el código de retorno de éste último se retorna al Python que llama, y el nuevo contenido del búfer refleja la acción de `ioctl()`. Esto es una ligera simplificación, porque si el búfer proporcionado tiene menos de 1024 bytes de longitud, primero se copia en un búfer estático de 1024 bytes de longitud que luego se pasa a `ioctl()` y se copia de nuevo en el búfer proporcionado.

Si `ioctl()` falla, se lanza la excepción `OSError`.

Un ejemplo:

```
>>> 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])
```

Lanza un evento `auditing event` `fcntl.ioctl` con argumentos `fd`, `request`, `arg`.

`fcntl.flock(f, operation)`

Realiza la operación de bloqueo *operation* sobre el descriptor de fichero *fd* (los objetos de fichero que proporcionan un método *fileno()* también son aceptados). Ver el manual de Unix *flock(2)* para más detalles. (En algunos sistemas, esta función es emulada usando `fcntl()`.)

Si `flock()` falla, una excepción *OSError* se lanza.

Lanza un *auditing event* `fcntl.flock` con argumentos *fd*, *operation*.

`fcntl.lockf(f, cmd, len=0, start=0, whence=0)`

Esto es esencialmente un «wrapper» de las llamadas de bloqueo `fcntl()`. * *fd* * es el descriptor de fichero (los objetos de fichero que proporcionan un método *fileno()* también se aceptan) del archivo para bloquear o desbloquear, y *cmd* es uno de los siguientes valores:

- `LOCK_UN` – desbloquear
- `LOCK_SH` – adquirir un bloqueo compartido
- `LOCK_EX` – adquirir un bloqueo exclusivo

Cuando *cmd* es `LOCK_SH` o `LOCK_EX`, también se puede usar OR bit a bit con `LOCK_NB` para evitar el bloqueo en la adquisición de bloqueos. Si se usa `LOCK_NB` y no se puede adquirir el bloqueo, se generará un `LOCK_NB` y la excepción tendrá un atributo *errno* establecido a `EACCES` o `EAGAIN` (según el sistema operativo; para la portabilidad, compruebe ambos valores). En al menos algunos sistemas, `LOCK_EX` solo se puede usar si el descriptor de fichero se refiere a un archivo abierto para escritura.

len es el número de bytes a bloquear, *start* es el byte de «offset» en el cual comienza el bloqueo, relativo a *whence*, y *whence* es como con `io.IOBase.seek()`, específicamente:

- 0 – relativo al comienzo del archivo (`os.SEEK_SET`)
- 1 – relativa a la posición actual del búfer (`os.SEEK_CUR`)
- 2 – relativo al final del archivo (`os.SEEK_END`)

El valor por defecto de **start** es 0, lo que significa que comienza al inicio del archivo. El valor por defecto para *len* es 0 lo que significa bloquear hasta el final del archivo. El valor por defecto para *whence* también es 0.

Lanza un *auditing event* `fcntl.lockf` con argumentos *fd*, *cmd*, *len*, *start*, *whence*.

Ejemplos (todos en un sistema compatible con SVR4):

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhlh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

Tenga en cuenta que en el primer ejemplo, la variable de valor de retorno *rv* contendrá un valor entero; en el segundo ejemplo contendrá un objeto *bytes*. El diseño de la estructura para la variable *lockdata* depende del sistema — por lo tanto, usar la llamada `flock()` puede ser mejor.

Ver también:

Módulo *os* Si los flags de bloqueo `O_SHLOCK` y `O_EXLOCK` están presentes en el módulo *os* (sólo en BSD), la función `os.open()` proporciona una alternativa a las funciones `lockf()` y `flock()`.

35.10 pipes — Interfaz para tuberías de shell

Código fuente: [Lib/pipes.py](#)

El módulo `pipes` define una clase para abstraer el concepto de una *tubería* — una secuencia de conversores de un archivo a otro.

Debido a que el módulo utiliza líneas de comando `/bin/sh`, se requiere una interfaz POSIX o un shell compatible para ejecutar `os.system()` y `os.popen()`.

El módulo `pipes` define la siguiente clase:

```
class pipes.Template
    Una abstracción de una tubería.
```

Ejemplo:

```
>>> 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'
```

35.10.1 Objetos Template

Las instancias `Template` tienen los siguientes métodos:

`Template.reset()`
Recupera el estado inicial del `Template` de una tubería.

`Template.clone()`
Retorna una nueva y equivalente instancia del `Template` de una tubería.

`Template.debug(flag)`
Si *flag* es verdadero, activa la depuración. Si no, la depuración no se activa. Cuando la depuración está activada, los comandos a ejecutar son impresos, y se le añade el comando `set -x` a la shell para ser más verboso.

`Template.append(cmd, kind)`
Añade una nueva acción al final. La variable *cmd* tiene que ser un comando válido de *Bourne Shell*. La variable *kind* está formada por dos letras.

La primera letra puede ser tanto `-` (que significa que el comando lee su entrada de datos estándar), `f` (que significa que el comando lee un fichero desde línea de comandos) o `.` (que significa que el comando no lee ninguna entrada de datos, y por tanto tiene que ser el primero.)

De manera parecida, la segunda letra puede ser tanto `-` (que significa que el comando escribe a la salida de datos estándar), `f` (que significa que el comando escribirá a un fichero de la línea de comandos) o `.` (que significa que el comando no escribe nada, por lo que debe ser el primero)

`Template.prepend(cmd, kind)`
Añade una nueva acción al principio. Ver `append()` para la explicación de los argumentos.

`Template.open(file, mode)`
Retorna un objeto similar a un fichero, abierto a *file*, pero leído o escrito por la tubería. Destacar que solo una de `'r'` o `'w'` puede ser añadida.

`Template.copy(infile, outfile)`
Copia *infile* a *outfile* a través de la tubería.

35.11 resource — Información sobre el uso de recursos

Este módulo proporciona mecanismos básicos para medir y controlar los recursos del sistema utilizados por un programa.

Las constantes simbólicas se utilizan para especificar recursos concretos del sistema y para solicitar información de uso sobre el proceso actual o sus elementos secundarios.

Se genera un `OSError` cuando la llamada al sistema (`syscall`) falla.

exception `resource.error`

Un alias en desuso de `OSError`.

Distinto en la versión 3.3: Tras [PEP 3151](#) esta clase se convirtió en un alias de `OSError`.

35.11.1 Límites de recursos

El uso de recursos se puede limitar usando la función `setrlimit()` que se describe a continuación. Cada recurso está controlado por un par de límites: un límite flexible y un límite duro. El límite flexible es el límite actual, y puede ser reducido o elevado con el tiempo mediante un proceso. El límite flexible nunca puede exceder el límite duro. El límite duro se puede reducir a cualquier valor mayor que el del límite flexible, pero no se puede elevar. (Solo los procesos con el UID efectivo del superusuario pueden aumentar un límite duro.)

Los recursos específicos que se pueden limitar dependen del sistema. Se describen en la página de manual `getrlimit(2)`. Los recursos enumerados a continuación se admiten cuando el sistema operativo subyacente los admite; los recursos que no pueden ser verificados o controlados por el sistema operativo no se definen en este módulo para esas plataformas.

`resource.RLIM_INFINITY`

Constante utilizada para representar el límite de un recurso ilimitado.

`resource.getrlimit(resource)`

Retorna una tupla (`soft`, `hard`) con los límites flexible y duro actuales de `resource`. Genera `ValueError` si se especifica un recurso no válido o `error` si la llamada al sistema subyacente falla inesperadamente.

`resource.setrlimit(resource, limits)`

Establece nuevos límites para el consumo de `resource`. El argumento `limits` debe ser una tupla de dos enteros (`soft`, `hard`) que describe los nuevos límites. Un valor de `RLIM_INFINITY` se puede utilizar para solicitar un límite ilimitado.

Genera `ValueError` si se especifica un recurso no válido, si el nuevo límite flexible excede el límite duro, o si un proceso intenta aumentar el límite duro. Si se especifica un límite de `RLIM_INFINITY` cuando el límite duro o el límite del sistema para ese recurso no son ilimitados, se producirá un `ValueError`. Un proceso con el UID efectivo de superusuario puede solicitar cualquier valor de límite válido, incluso ilimitado, pero se generará un `ValueError` si el límite solicitado excede el límite impuesto por el sistema.

`setrlimit` también puede generar un `error` si falla la llamada al sistema subyacente.

VxWorks solo admite configurar `RLIMIT_NOFILE`.

Genera un `auditing event` `resource.setrlimit` con los argumentos `resource`, `limits`.

`resource.prlimit(pid, resource[, limits])`

Combina `setrlimit()` y `getrlimit()` en una sola función y admite obtener y establecer los límites de recursos de un proceso arbitrario. Si `pid` es 0, entonces la llamada se aplica al proceso actual. `resource` y `limits` tienen el mismo significado que en `setrlimit()`, excepto por que `limits` es opcional.

Cuando no se proporciona `limits` la función retorna el límite de `resource` del proceso `pid`. Cuando se proporciona `limits`, se establece el límite de `resource` del proceso y se retorna el límite de recursos anterior.

Genera `ProcessLookupError` cuando no se encuentra `pid` y `PermissionError` cuando el usuario no tiene `CAP_SYS_RESOURCE` para el proceso.

Genera un *evento de auditoría* `resource.prlimit` con los argumentos `pid`, `resource`, `limits`.

Disponibilidad: Linux 2.6.36 o posterior con glibc 2.13 o posterior.

Nuevo en la versión 3.4.

Estos símbolos definen los recursos cuyo consumo se puede controlar usando las funciones `setrlimit()` y `getrlimit()` que se describen más abajo. Los valores de estos símbolos son exactamente las constantes utilizadas por programas en C.

La página de manual de Unix para `getrlimit(2)` detalla los recursos disponibles. Tenga en cuenta que no todos los sistemas usan el mismo símbolo o el mismo valor para referirse al mismo recurso. Este módulo no pretende enmascarar las diferencias entre plataformas — los símbolos no definidos para una plataforma no estarán disponibles en este módulo en esa plataforma.

`resource.RLIMIT_CORE`

El tamaño máximo (en bytes) de un archivo central que puede crear el proceso actual. Esto podría resultar en la creación de un archivo central parcial si se requiriera uno más grande para contener la imagen del proceso entera.

`resource.RLIMIT_CPU`

La cantidad máxima de tiempo del procesador (en segundos) que puede utilizar un proceso. Si se excede este límite se envía una señal `SIGXCPU` al proceso. (Vea la documentación del módulo `signal` para más información sobre cómo detectar esta señal y hacer algo productivo, p. ej. descargar los archivos abiertos al disco).

`resource.RLIMIT_FSIZE`

El tamaño máximo de un archivo que pueda crear el proceso.

`resource.RLIMIT_DATA`

El tamaño máximo (en bytes) de la memoria *heap* del proceso.

`resource.RLIMIT_STACK`

El tamaño máximo (en bytes) de la pila de llamadas para el proceso actual. Esto afecta únicamente a la pila del hilo principal en un proceso multi-hilo.

`resource.RLIMIT_RSS`

El tamaño máximo del conjunto residente (*RSS*) del que puede disponer el proceso.

`resource.RLIMIT_NPROC`

El número máximo de procesos que puede crear el proceso actual.

`resource.RLIMIT_NOFILE`

El número máximo de descriptores de archivo abierto para el proceso actual.

`resource.RLIMIT_OFILE`

El nombre BDS para `RLIMIT_NOFILE`.

`resource.RLIMIT_MEMLOCK`

El espacio de direcciones máximo que se puede bloquear en la memoria.

`resource.RLIMIT_VMEM`

El área de memoria mapeada más grande que puede ocupar el proceso.

`resource.RLIMIT_AS`

El área máxima (en bytes) de espacio de direcciones que puede tomar el proceso.

`resource.RLIMIT_MSGQUEUE`

El número de bytes que se pueden asignar a las colas de mensajes POSIX.

Disponibilidad: Linux 2.6.8 o posterior.

Nuevo en la versión 3.4.

`resource.RLIMIT_NICE`

El techo del nivel del proceso *nice* (calculado como `20 - rlim_cur`).

Disponibilidad: Linux 2.6.12 o posterior.

Nuevo en la versión 3.4.

`resource.RLIMIT_RTPRIO`

El techo de la prioridad en tiempo real.

Disponibilidad: Linux 2.6.12 o posterior.

Nuevo en la versión 3.4.

`resource.RLIMIT_RTIME`

El límite de tiempo (en microsegundos) en tiempo de CPU que puede dedicar un proceso de programación en tiempo real sin hacer una llamada al sistema de bloqueo.

Disponibilidad: Linux 2.6.25 o posterior.

Nuevo en la versión 3.4.

`resource.RLIMIT_SIGPENDING`

El número de señales que el proceso puede poner en cola.

Disponibilidad: Linux 2.6.8 o posterior.

Nuevo en la versión 3.4.

`resource.RLIMIT_SBSIZE`

El tamaño máximo (en bytes) de uso del búfer del socket para este usuario. Esto limita la cantidad de memoria de red, y por lo tanto la cantidad de mbufs, que este usuario puede retener en todo momento.

Disponibilidad: FreeBSD 9 o posterior.

Nuevo en la versión 3.4.

`resource.RLIMIT_SWAP`

The maximum size (in bytes) of the swap space that may be reserved or used by all of this user id's processes. This limit is enforced only if bit 1 of the `vm.overcommit` sysctl is set. Please see [tuning\(7\)](#) for a complete description of this sysctl.

Disponibilidad: FreeBSD 9 o posterior.

Nuevo en la versión 3.4.

`resource.RLIMIT_NPTS`

El número máximo de pseudo-terminales que puede crear esta ID de usuario.

Disponibilidad: FreeBSD 9 o posterior.

Nuevo en la versión 3.4.

35.11.2 Utilización de recursos

Estas funciones se usan para recuperar la información de utilización de recursos:

`resource.getrusage(who)`

Esta función retorna un objeto que describe los recursos consumidos por el proceso actual o sus elementos secundarios, según como esté especificado en el parámetro *who*. El parámetro *who* debe especificarse usando una de las constantes `RUSAGE_*` descritas más abajo.

Un ejemplo sencillo:

```
from resource import *
import time

# a non CPU-bound task
time.sleep(3)
print(getrusage(RUSAGE_SELF))

# a CPU-bound task
```

(continué en la próxima página)

(proviene de la página anterior)

```
for i in range(10 ** 8):
    _ = 1 + 1
print(getrusage(RUSAGE_SELF))
```

Los campos del valor retornado describen cómo se ha utilizado un recurso específico, p. ej. la cantidad de tiempo dedicada a la ejecución en modo usuario o el número de veces que el proceso ha sido intercambiado desde la memoria principal. Algunos valores dependen del intervalo de tic del reloj, p. ej. la cantidad de memoria que está usando el proceso.

Por compatibilidad con versiones anteriores, el valor retornado es accesible también como una tupla de 16 elementos.

Los campos `ru_utime` y `ru_stime` del valor retornado son valores de coma flotante que representan la cantidad de tiempo dedicada a la ejecución en modo usuario y la cantidad de tiempo dedicada a la ejecución en modo sistema respectivamente. Los valores restantes son enteros. Consulte la página del manual `getrusage(2)` para información detallada sobre estos valores. A continuación se presenta un breve resumen:

Índice	Campo	Recurso
0	<code>ru_utime</code>	tiempo en modo usuario (flotante en segundos)
1	<code>ru_stime</code>	tiempo en modo sistema (flotante en segundos)
2	<code>ru_maxrss</code>	tamaño máximo del conjunto residente
3	<code>ru_ixrss</code>	tamaño de memoria compartida
4	<code>ru_idrss</code>	tamaño de memoria no compartida
5	<code>ru_isrss</code>	tamaño de la pila no compartida
6	<code>ru_minflt</code>	fallos de página que no requieran E/S
7	<code>ru_majflt</code>	fallos de página que requieran E/S
8	<code>ru_nswap</code>	número de intercambios
9	<code>ru_inblock</code>	bloque de operaciones de entrada
10	<code>ru_oublock</code>	bloque de operaciones de salida
11	<code>ru_msgsnd</code>	mensajes enviados
12	<code>ru_msgrcv</code>	mensajes recibidos
13	<code>ru_nsignals</code>	señales recibidas
14	<code>ru_nvcsw</code>	intercambios de contexto voluntarios
15	<code>ru_nivcsw</code>	intercambios de contexto involuntarios

Esta función generará un `ValueError` si el parámetro `who` especificado no es válido. También puede generar una excepción `error` en circunstancias inusuales.

`resource.getpagesize()`

Retorna el número de bytes en una página de sistema. (Esta no es necesariamente del mismo tamaño que la página de hardware).

Los siguientes símbolos `RUSAGE_*` se pasan a la función `getrusage()` para especificar qué información de procesos se debería proporcionar.

`resource.RUSAGE_SELF`

Pasar a `getrusage()` para solicitar recursos consumidos por el proceso de llamada, que es la suma de recursos utilizados por todos los hilos en el proceso.

`resource.RUSAGE_CHILDREN`

Pasar a `getrusage()` para solicitar recursos consumidos por procesos secundarios del proceso de llamada que se han terminado o a los que se les está esperando.

`resource.RUSAGE_BOTH`

Pasar a `getrusage()` para solicitar recursos consumidos por el proceso actual y sus procesos secundarios. Puede que no esté disponible en todos los sistemas.

`resource.RUSAGE_THREAD`

Pasa a `getrusage()` para solicitar recursos consumidos por el hilo actual. Puede que no esté disponible en

todos los sistemas.

Nuevo en la versión 3.2.

35.12 nis — Interfaz a Sun's NIS (Páginas amarillas)

El módulo `nis` da una envoltura alrededor de la biblioteca NIS, útil para la administración central de varios hosts.

Debido a que la NIS sólo existe en sistemas Unix, este módulo sólo está disponible para Unix.

El módulo `nis` define las siguientes funciones:

`nis.match(key, mapname, domain=default_domain)`

Retorna la coincidencia para `key` en el mapa `mapname`, o retorna un error (`nis.error`) si es `none`. Ambas deben ser cadenas, `key` es 8 bits limpios. El valor de retorno es un array arbitrario de bytes (puede contener `NULL` y otros placeres).

Tenga en cuenta que `mapname` se comprueba primero si es un alias de otro nombre.

El argumento `domain` permite anular el dominio NIS utilizado para la búsqueda. Si no se especifica, la búsqueda se realiza en el dominio NIS por defecto.

`nis.cat(mapname, domain=default_domain)`

Retorna un diccionario de mapeo de `key` a `value` de tal manera que `match(key, mapname) == value`. Tenga en cuenta que tanto las claves como los valores del diccionario son arreglos arbitrarios de bytes.

Tenga en cuenta que `mapname` se comprueba primero si es un alias de otro nombre.

El argumento `domain` permite anular el dominio NIS utilizado para la búsqueda. Si no se especifica, la búsqueda se realiza en el dominio NIS por defecto.

`nis.maps(domain=default_domain)`

Retorna una lista de todos los mapas válidos.

El argumento `domain` permite anular el dominio NIS utilizado para la búsqueda. Si no se especifica, la búsqueda se realiza en el dominio NIS por defecto.

`nis.get_default_domain()`

Retorna el dominio NIS por defecto del sistema.

El módulo `nis` define la siguiente excepción:

exception `nis.error`

Un error que se produce cuando una función NIS retorna un código de error.

35.13 syslog — Rutinas de la biblioteca syslog de Unix

Este módulo ofrece una interfaz a las rutinas de la biblioteca `syslog` de Unix. Consulte las páginas del manual de Unix para una descripción detallada de la *facility* (es así como se llama en la documentación a un subsistema de aplicaciones) `syslog`.

Este módulo envuelve la familia de rutinas de `syslog`. En el módulo `logging.handlers` está disponible `SysLogHandler`, una biblioteca en Python puro que puede comunicarse con un servidor `syslog`.

El módulo define las siguientes funciones:

`syslog.syslog(message)`

`syslog.syslog(priority, message)`

Envía la cadena de caracteres *message* al registrador del sistema. Se añade un final de línea si es necesario. Cada mensaje se etiqueta con una prioridad compuesta por una *facility* y un *nivel*. El argumento opcional *priority*, que por defecto es `LOG_INFO`, determina la prioridad del mensaje. Si la *facility* no está codificada en *priority* usando un OR lógico (`LOG_INFO | LOG_USER`), se usará el valor con el que se llama a `openlog()`.

Si `openlog()` no se ha llamado antes de la llamada a `syslog()`, `openlog()` se llamará sin argumentos.

Lanza un *evento de auditoría* `syslog.syslog` con los argumentos `priority, message`.

`syslog.openlog([ident[, logoption[, facility]]])`

Las opciones de registro de las llamadas a `syslog()` pueden establecerse llamando a la función `openlog()`. `syslog()` llamará a `openlog()` sin argumentos si el registro no está abierto actualmente.

El argumento por palabra clave opcional *ident* es una cadena de caracteres que precede a cada mensaje, y por defecto es `sys.argv[0]` retirando los componentes delanteros de la ruta. El argumento nombrado opcional *logoption* (por defecto es 0) es un campo de tipo bit – véanse a continuación los posibles valores que pueden combinarse. El argumento nombrado opcional *facility* (por defecto es `LOG_USER`) establece una *facility* por defecto para mensajes que no tienen una *facility* explícitamente codificada.

Lanza un *evento de auditoría* `syslog.openlog` con los argumentos `ident, logoption, facility`.

Distinto en la versión 3.2: En versiones anteriores, los argumentos nombrados no estaban permitidos, y *ident* era obligatorio. El valor por defecto de *ident* dependía de las librerías del sistema, y a menudo era `python` en lugar del nombre del fichero del programa en Python.

`syslog.closelog()`

Reinicia los valores del módulo `syslog` y llama a la librería del sistema `closelog()`.

Esto provoca que el módulo actúe como lo hacía cuando fue importado inicialmente. Por ejemplo, se llamará a `openlog()` en la primera llamada a `syslog()` (si no se ha llamado a `openlog()` ya), e *ident* y otros parámetros de `openlog()` se reiniciarán a sus valores por defecto.

Lanza un *evento de auditoría* `syslog.closelog` sin argumentos.

`syslog.setlogmask(maskpri)`

Asigna la máscara de prioridad a *maskpri* y retorna el valor anterior de la máscara. Las llamadas a `syslog()` con un nivel de prioridad no asignado en *maskpri* se ignoran. El comportamiento por defecto es registrar todas las prioridades. La función `LOG_MASK(pri)` calcula la máscara para la prioridad individual *pri*. La función `LOG_UPTO(pri)` calcula la máscara para todas las prioridades mayores o iguales que *pri*.

Lanza un *evento de auditoría* `syslog.setlogmask` con argumento *maskpri*.

El módulo define las siguientes constantes:

Niveles de prioridad (de más alto a más bajo): `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`, de `LOG_LOCAL0` a `LOG_LOCAL7`, y, si está definida en `<syslog.h>`, `LOG_AUTHPRIV`.

Opciones de registro: `LOG_PID`, `LOG_CONS`, `LOG_NDELAY`, y, si están definidas en `<syslog.h>`, `LOG_ODELAY`, `LOG_NOWAIT`, y `LOG_PERROR`.

35.13.1 Ejemplos

Ejemplo sencillo

Un conjunto sencillo de ejemplos:

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

Un ejemplo de configuración de varias opciones de registro, que incluirán el identificador del proceso en los mensajes registrados, y escribirán los mensajes a la facility de destino usada para el registro de correo:

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

Módulos Reemplazados

Los módulos descritos en este capítulo se encuentran obsoletos, y sólo se conservan por compatibilidad con versiones anteriores. Han sido reemplazados por otros módulos.

36.1 `optparse` — Analizador sintáctico (parser) para opciones de línea de comandos

Source code: [Lib/optparse.py](#)

Obsoleto desde la versión 3.2: El módulo `optparse` está obsoleto y no será desarrollado de aquí en adelante. El desarrollo continuará en el módulo `argparse`.

`optparse` es una biblioteca más conveniente, flexible y poderosa para analizar opciones de línea de comandos que el antiguo módulo `getopt`. `optparse` usa un estilo más declarativo: creas una instancia de `OptionParser`, le añades las opciones deseadas y realizas el análisis sintáctico de la línea de comandos. `optparse` permite a los usuarios especificar opciones siguiendo la sintaxis convencional de GNU/POSIX, además de generar mensajes de uso y de ayuda automáticamente.

A continuación puedes ver un ejemplo de uso de `optparse` mediante un script simple:

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
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()
```

Con estas pocas líneas de código, los usuarios de tu script ahora pueden hacer un uso «normal» del mismo mediante la línea de comandos, por ejemplo:

```
<yourscript> --file=outfile -q
```

A medida que analiza la línea de comandos, `optparse` establece los atributos del objeto `options` retornado por `parse_args()` basándose en los valores de la línea de comandos proporcionada por el usuario. Cuando `parse_args()` termina de analizar esta línea de comandos, `options.filename` será "outfile" y `options.verbose` será `False`. `optparse` admite opciones largas y cortas, fusionar opciones cortas y asociar opciones con sus argumentos de diversas formas. Por lo tanto, las siguientes líneas de comandos son todas equivalentes al ejemplo previo:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

Additionally, users can run one of the following

```
<yourscript> -h
<yourscript> --help
```

y `optparse` imprimirá un breve resumen de las opciones de tu script:

```
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
```

donde el valor de `yourscript` se determina en tiempo de ejecución (normalmente a partir de `sys.argv[0]`).

36.1.1 Contexto

`optparse` ha sido diseñado explícitamente para fomentar la creación de programas con interfaces de línea de comandos convencionales y sencillas. Con ese fin en mente, solo admite la sintaxis y semántica de línea de comandos más común y convencionalmente usada en Unix. Si no estás familiarizado con estas convenciones, lee esta sección para familiarizarte con ellas.

Terminología

argumento una cadena de caracteres ingresada en la línea de comandos y pasada mediante la shell a `exec1()` o `execv()`. En Python, los argumentos son elementos de `sys.argv[1:]` (dado que `sys.argv[0]` es el propio nombre del programa que se está ejecutando). Las shells de Unix también usan el término «word» (“palabra”) para referirse a ellos.

En ocasiones es deseable proporcionar una lista de argumentos que no sea `sys.argv[1:]`, por lo que deberías considerar un “argumento” como “un elemento de `sys.argv[1:]` o de alguna otra lista proporcionada como sustituto de `sys.argv[1:]`”.

opción un argumento utilizado para proporcionar información adicional con la finalidad de guiar o personalizar la ejecución de un programa. Existen muchas sintaxis diferentes para especificar opciones. La sintaxis tradicional de Unix es un guion (“-”) seguido de una sola letra, por ejemplo `-x` o `-F`. La sintaxis tradicional de Unix permite además fusionar múltiples opciones en un solo argumento, por ejemplo `-x -F` es equivalente a `-xF`. Por otro lado, el proyecto GNU introdujo `--` seguido de una serie de palabras separadas por guiones, por ejemplo `--file` o `--dry-run`. Estas son las dos únicas sintaxis para opciones que el módulo `optparse` proporciona.

Algunas de las otras sintaxis para opciones que el mundo ha visto son:

- un guion seguido de algunas letras, por ejemplo `-pf` (esto *no* es lo mismo que múltiples opciones fusionadas en un solo argumento)
- un guion seguido de una palabra completa, por ejemplo `-file` (esto es técnicamente equivalente a la sintaxis anterior, pero generalmente no se ven ambas en un mismo programa)

- un signo más seguido de una sola letra, unas pocas letras o una palabra, por ejemplo `+f o +rgb`
- una barra seguida de una letra, de unas pocas letras o de una palabra, por ejemplo `/f o /file`

Estas sintaxis para opciones no son compatibles con `optparse` y nunca lo serán. Esto es deliberado: los tres primeros no son estándar en ningún entorno y el último solo tiene sentido si se dirige exclusivamente a VMS, MS-DOS y/o Windows.

argumento de opción un argumento que sigue a una opción, está estrechamente asociado con ella y se consume de la lista de argumentos cuando esa opción es consumida. Con `optparse`, los argumentos de las opciones pueden estar en un argumento separado de su opción:

```
-f foo
--file foo
```

o incluidos en el mismo argumento:

```
-ffoo
--file=foo
```

Normalmente, una opción dada puede aceptar o no un argumento. Mucha gente desea una característica de «argumentos de opción opcionales», lo que implica que algunas opciones aceptarán un argumento si está presente y no lo aceptarán si no lo está. Esto es algo controvertido, porque hace que el análisis sea ambiguo: si `-a` toma un argumento opcional y `-b` es otra opción completamente distinta, ¿cómo interpretamos `-ab`? Debido a esta ambigüedad, `optparse` no es compatible con esta funcionalidad.

argumento posicional es algo que queda en la lista de argumentos después de que las opciones hayan sido analizadas sintácticamente, es decir, después de que las opciones y sus argumentos hayan sido analizados y eliminados de la lista de argumentos.

opción requerida es una opción que debe proporcionarse forzosamente en la línea de comandos. Ten en cuenta que la frase «opción requerida» se contradice a sí misma. `optparse` no te impide implementar opciones requeridas, pero tampoco brinda mucha ayuda para ello.

Por ejemplo, considera esta hipotética línea de comandos:

```
prog -v --report report.txt foo bar
```

`-v` y `--report` son ambas opciones. `report.txt` es un argumento de opción, suponiendo que `--report` toma un argumento. En cambio, `foo` y `bar` son ambos argumentos posicionales.

¿Qué finalidad tienen las opciones?

Las opciones se utilizan para poder proporcionar información adicional con el fin de ajustar o personalizar la ejecución de un programa. Por si aún no ha quedado claro, las opciones suelen ser *opcionales*. Un programa debería poder ejecutarse sin problemas sin ninguna opción. (Elija un programa aleatorio del conjunto de herramientas de Unix o GNU. ¿Puede ejecutarse sin ninguna opción y aún así tener sentido? Las principales excepciones son `find`, `tar` y `dd`—los cuales son todos bichos raros mutantes que han sido apropiadamente criticados por su sintaxis no estándar y por tener interfaces confusas).

Como se ha comentado, mucha gente quiere que sus programas tengan «opciones requeridas». Pero pensemos en ello detenidamente. ¡Si es necesario, entonces *no es opcional*! Si hay una pieza de información absolutamente requerida para que tu programa pueda ejecutarse correctamente no uses opciones, para eso están los argumentos posicionales.

Como ejemplo de un buen diseño de una interfaz de línea de comandos, considera la humilde herramienta `cp` para copiar archivos. No tiene mucho sentido intentar copiar archivos sin proporcionar un destino y al menos una fuente de origen. Por lo tanto, `cp` falla si lo ejecutas sin argumentos. Sin embargo, tiene una sintaxis flexible y útil que no requiere ninguna opción:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

Puedes hacer mucho simplemente con eso. La mayoría de las implementaciones de `cp` proporcionan un montón de opciones para modificar exactamente cómo se copian los archivos: se puede preservar el modo y la fecha de modificación, evitar que se sigan enlaces simbólicos, preguntar antes de sobrescribir el contenido de archivos existentes, etc. Pero nada de esto distrae de la misión principal de `cp`, que consiste en copiar uno o varios archivos en otro directorio.

¿Qué finalidad tienen los argumentos posicionales?

Los argumentos posicionales son adecuados para aquellas piezas de información que tu programa, absolutamente y sin duda alguna, requiere para funcionar.

Una buena interfaz de usuario debería tener la menor cantidad de requisitos absolutos posibles. Si tu programa requiere 17 piezas distintas de información para ejecutarse correctamente, no importa mucho *cómo* obtengas esa información del usuario; la mayoría de ellos se rendirán y se irán antes de ejecutar con éxito el programa. Esto se aplica tanto si la interfaz de usuario es una línea de comandos, un archivo de configuración o una GUI: si hace demasiadas demandas a sus usuarios, la mayoría simplemente se rendirá.

En resumen, trata de minimizar la cantidad de información que los usuarios están absolutamente obligados a proporcionar, utiliza valores predeterminados sensatos siempre que sea posible. Como es natural, también deseas que tus programas sean razonablemente flexibles, para eso están las opciones. De nuevo, no importa si son entradas en un archivo de configuración, widgets en un cuadro de diálogo de «Preferencias» de una GUI u opciones en la línea de comandos; cuantas más opciones implementes, más flexible será tu programa y más complicada se vuelve su implementación. Una excesiva flexibilidad evidentemente también tiene inconvenientes, demasiadas opciones pueden abrumar a los usuarios y hacer que tu código sea mucho más difícil de mantener.

36.1.2 Tutorial

Si bien `optparse` es bastante flexible y potente, también es sencillo de usar en la mayoría de los casos. Esta sección cubre los patrones de código comunes a cualquier programa basado en `optparse`.

En primer lugar, necesitas importar la clase `OptionParser` y luego, al comienzo del programa principal, crear una instancia de ella:

```
from optparse import OptionParser
...
parser = OptionParser()
```

Ahora ya puedes comenzar a definir opciones. La sintaxis básica es:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

Cada opción tiene una o más cadenas de caracteres de opción, como `-f` o `--file` y varios atributos de opción que le dicen a `optparse` qué esperar y qué hacer cuando encuentra esa opción en la línea de comandos.

Normalmente, cada opción tendrá una cadena de opción corta y una cadena de opción larga, por ejemplo:

```
parser.add_option("-f", "--file", ...)
```

Puedes definir tantas cadenas de opción cortas y tantas largas como desees (incluso ninguna), siempre que haya al menos una cadena de opción en total.

Las cadenas de opción pasadas a `OptionParser.add_option()` son en definitiva etiquetas para la opción definida por esa llamada. Simplemente por brevedad, con frecuencia nos referiremos a *encontrar una opción* en la línea de comandos. En realidad, `optparse` encuentra *cadenas de caracteres de opción* y busca opciones en ellas.

Una vez que todas tus opciones estén definidas, indica a `optparse` que analice sintácticamente la línea de comandos de tu programa:

```
(options, args) = parser.parse_args()
```

(Si lo deseas, puedes pasar una lista de argumentos personalizada a `parse_args()`, pero eso rara vez es necesario: por defecto se usa `sys.argv[1:]`.)

`parse_args()` retorna dos valores:

- `options`, un objeto que contiene valores para todas tus opciones. Por ejemplo, si `--file` toma un argumento de una sola cadena de caracteres, entonces `options.file` será el nombre del archivo proporcionado por el usuario o `None` si el usuario no proporcionó esa opción en la línea de comandos
- `args`, la lista de argumentos posicionales que quedan después de analizar las opciones

Este tutorial solo cubre los cuatro atributos de opción más importantes: *action*, *type*, *dest* (destino) y *help*. De todos ellos, *action* es el fundamental.

Comprendiendo las acciones de opción

Las acciones le dicen a *optparse* qué hacer cuando encuentra una determinada opción en la línea de comandos. Hay un conjunto fijo de acciones codificadas en *optparse*. Agregar nuevas acciones es un tema avanzado cubierto en la sección *Extendiendo el módulo optparse*. La mayoría de las acciones indican a *optparse* que almacene un valor en alguna variable, por ejemplo, tomar una cadena de caracteres de la línea de comandos y almacenarla en un atributo del objeto `options`.

Si no se especifica una acción para la opción, *optparse* usa por defecto `store`.

La acción store

La acción para opciones más común es `store`, que le dice a *optparse* que tome el siguiente argumento (o el resto del argumento actual), se asegure de que sea del tipo correcto y lo guarde en el destino elegido.

Por ejemplo:

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

Ahora creamos una línea de comandos simulada y le pedimos a *optparse* que la analice:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

Cuando *optparse* se encuentra con la cadena de opción `-f`, consume el siguiente argumento, `foo.txt` y lo almacena en `options.filename`. Por lo tanto, después de la llamada a `parse_args()`, `options.filename` será `"foo.txt"`.

Algunos de los otros tipos de opción admitidos por *optparse* son `int` y `float`. Aquí hay una opción que espera un argumento entero:

```
parser.add_option("-n", type="int", dest="num")
```

Ten en cuenta que esta opción no tiene una cadena de opción larga, lo cual es perfectamente aceptable. Además, no hay ninguna acción explícita, ya que el valor predeterminado es `store`.

Analicemos otra línea de comandos simulada. En esta ocasión, vamos a proporcionar el argumento de la opción pegado junto a la misma, sin separación entre ambos: dado que `-n42` (un argumento) es equivalente a `-n 42` (dos argumentos), el código:

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

imprimirá `42`.

Si no se especifica un tipo, *optparse* asume `string`. Esto, combinado con el hecho de que la acción predeterminada es `store`, implica que nuestro primer ejemplo puede implementarse de forma mucho más concisa:

```
parser.add_option("-f", "--file", dest="filename")
```

Si no se proporciona un destino, *optparse* infiere un destino por defecto adecuado a partir de las cadenas de opción proporcionadas: si la primera cadena de opción larga es `--foo-bar`, entonces el destino predeterminado es `foo_bar`. Si no hay cadenas de opción largas, *optparse* mira la primera cadena de opción corta: por ejemplo, el destino predeterminado para `-f` es `f`.

optparse también incluye el tipo incorporado `complex`. La adición de nuevos tipos se cubre en la sección *Ex-tendiendo el módulo optparse*.

Manejo de opciones booleanas (flags)

Las opciones flags—establecen una variable en verdadero o falso cuando se encuentra una opción en particular— son bastante comunes. *optparse* las admite con dos acciones diferenciadas, `store_true` y `store_false`. Por ejemplo, puedes tener un flag `verbose` que se activa con `-v` y se desactiva con `-q`:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Aquí tenemos dos opciones diferentes con el mismo destino, lo cual es totalmente correcto. Solo significa que debes tener un poco de cuidado al establecer los valores predeterminados, lo veremos a continuación.

Cuando *optparse* encuentra `-v` en la línea de comandos, establece `options.verbose` en `True`; cuando encuentra `-q`, `options.verbose` se establece en `False`.

Otras acciones

Algunas de las otras acciones soportadas por *optparse* son:

"store_const" almacena un valor constante

"append" agrega el argumento de esta opción a una lista

"count" incrementa un contador en uno

"callback" llama a una función específica

Estas acciones se tratan en la sección *Guía de referencia* y en la sección *Retrollamadas de opción*.

Valores por defecto

Todos los ejemplos anteriores implican establecer alguna variable (el «destino») cuando aparecen ciertas opciones en la línea de comandos. ¿Qué pasa si esas opciones nunca se encuentran? Dado que no proporcionamos ningún valor predeterminado, todas las variables están establecidas en `None`. Por lo general, esto está bien, pero a veces se desea tener más control. *optparse* permite proporcionar un valor predeterminado para cada destino, que es asignado antes de analizar la línea de comandos.

Primero, considera el ejemplo `verbose/quiet` anterior. Si queremos que *optparse* establezca `verbose` en `True` a menos que encuentre `-q`, entonces podemos hacer lo siguiente:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Dado que los valores predeterminados se aplican a *destination* en lugar de a cualquier opción en particular y que estas dos opciones tienen el mismo destino, lo anterior es exactamente equivalente a:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Considera lo siguiente:


```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Nuevamente, el valor predeterminado para `verbose` será `True`: el último valor predeterminado proporcionado para cualquier destino en particular es el único que se tendrá en cuenta.

Una forma más clara de especificar valores predeterminados es el método `set_defaults()` de `OptionParser`, al que puedes llamar en cualquier momento antes de llamar a `parse_args()`:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

Como vimos antes, el último valor especificado para un destino de opción dado es el que cuenta. Para mayor claridad, intenta utilizar un método u otro para establecer valores predeterminados, no ambos.

Generando ayuda

La capacidad de `optparse` para generar ayuda y texto de uso automáticamente es útil para crear interfaces de línea de comandos fáciles de usar. Todo lo que tienes que hacer es proporcionar un valor `help` para cada opción y, opcionalmente, un breve mensaje de uso para el programa en general. A continuación hay un `OptionParser` al que se le han añadido múltiples opciones fáciles de usar (documentadas):

```
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]")
```

Si `optparse` encuentra `-h` o `--help` en la línea de comandos, o si simplemente se llama al método `parser.print_help()`, se imprime lo siguiente en la salida estándar:

```
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]
```

(Si la salida de ayuda se activa mediante una opción de ayuda, `optparse` termina la ejecución después de imprimir el texto de ayuda.)

Estamos haciendo muchas cosas aquí con el fin de ayudar a que `optparse` genere el mejor mensaje de ayuda posible:

- el script define su propio mensaje de uso:

```
usage = "usage: %prog [options] arg1 arg2"
```

`optparse` expande `% prog` en la cadena de uso reemplazándolo por el nombre del programa actual, es decir, `os.path.basename (sys.argv [0])`. A continuación, la cadena expandida se imprime antes de la ayuda detallada de la opción.

Si no proporcionas una cadena de uso, `optparse` usa un valor predeterminado anodino pero apropiado: `"Usage: %prog [options]"`, lo cual está bien si tu script no toma ningún argumento posicional.

- cada opción simplemente define una cadena de ayuda y no se preocupa por el ajuste de línea. `optparse` se encarga de ajustar las líneas y hacer que la salida de ayuda se vea bien.
- las opciones que toman un valor indican este hecho en su mensaje de ayuda generado automáticamente, por ejemplo, para la opción «*mode*»:

```
-m MODE, --mode=MODE
```

Aquí, a «*MODE*» se le denomina una metavariable: representa el argumento que se espera que el usuario proporcione a `-m/--mode`. De forma predeterminada, `optparse` convierte el nombre de la variable de destino a mayúsculas y lo usa para la metavariable. En ocasiones, eso no es lo que se desea, por ejemplo, la opción `--filename` establece explícitamente `metavar="FILE"`, lo que da como resultado la siguiente descripción de opción generada automáticamente:

```
-f FILE, --filename=FILE
```

Sin embargo, esto es importante para algo más que para ahorrar espacio: el texto de ayuda escrito manualmente utiliza la metavariable `FILE` para indicarle al usuario que hay una conexión entre la sintaxis semiformal `-f FILE` y la descripción semántica informal «escribir la salida en `FILE`». Esta es una manera simple pero efectiva de hacer que tu texto de ayuda sea mucho más claro y útil para los usuarios finales.

- las opciones que tienen un valor predeterminado pueden incluir `%default` en la cadena de ayuda, en cuyo caso, `optparse` lo reemplazará por el resultado de aplicar `str()` al valor por defecto de esa opción. Si una opción no tiene un valor por defecto (o el valor por defecto es `None`), `%default` se reemplazará por `none`.

Agrupando opciones

Cuando se trabaja con muchas opciones, suele ser conveniente agruparlas para obtener una mejor salida de ayuda. La clase `OptionParser` puede contener varios grupos de opciones, cada uno de los cuales puede contener múltiples opciones.

Podemos obtener un grupo de opciones usando la clase `OptionGroup`:

```
class optparse.OptionGroup (parser, title, description=None)
    donde
```

- `parser` es la instancia de `OptionParser` en la que se insertará el grupo
- `title` es el título dado al grupo
- `description`, opcional, es la descripción larga del grupo

la clase `OptionGroup` hereda de `OptionContainer` (al igual que `OptionParser`), por lo que el método `add_option()` se puede usar para agregar una opción al grupo.

Una vez que se han declarado todas las opciones, usando el método `add_option_group()` de la clase `OptionParser` el grupo se agrega al analizador sintáctico previamente definido.

Agregar un `OptionGroup` a un analizador es fácil, continuando con el analizador definido en la sección anterior:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk. "
                    "It is believed that some of them bite.")
```

(continué en la próxima página)

(proviene de la página anterior)

```
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

Esto daría como resultado la siguiente salida de ayuda:

```
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.
```

Un ejemplo un poco más completo podría implicar el uso de más de un grupo, ampliando el ejemplo anterior:

```
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)
```

lo que da como resultado la siguiente salida:

```
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
```

Otro método interesante, particularmente cuando se trabaja programáticamente con grupos de opciones, es:

`OptionParser.get_option_group(opt_str)`

Retorna el `OptionGroup` al que pertenece la cadena de opción corta o larga *opt_str* (por ejemplo, `'-o'` o `'--option'`). Si no existe dicho `OptionGroup`, el método retorna `None`.

Imprimir una cadena de caracteres con la versión del programa

De forma similar a lo que ocurre con la cadena de uso abreviada, `optparse` también puede imprimir una cadena de versión para tu programa. Debes proporcionar la cadena mediante el argumento `version` de `OptionParser`:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

`%prog` se expande al igual que en `usage`. Aparte de eso, `version` puede contener cualquier cosa que desees. Cuando se proporciona, `optparse` agrega automáticamente una opción `--version` al analizador. Si encuentra esta opción en la línea de comandos, expande la cadena `version` (reemplazando `%prog`), la imprime en la salida estándar y termina la ejecución.

Por ejemplo, si tu script se llama `/usr/bin/foo`:

```
$ /usr/bin/foo --version
foo 1.0
```

Para imprimir y obtener la cadena de caracteres `version`, se pueden utilizar cualquiera de los siguientes métodos:

`OptionParser.print_version(file=None)`

Imprime el mensaje con la versión del programa actual (`self.version`) en *file* (por defecto, la salida estándar). Al igual que con `print_usage()`, cualquier aparición de `%prog` en `self.version` se reemplaza con el nombre del programa actual. El método no hace nada si `self.version` está vacío o no está definido.

`OptionParser.get_version()`

Igual que `print_version()`, pero retorna la cadena de versión en lugar de imprimirla.

Cómo maneja los errores el módulo optparse

A grandes rasgos, hay dos clases de errores de los que `optparse` tiene que preocuparse: errores del programador y errores del usuario. Los errores del programador suelen ser el resultado de llamadas erróneas a `OptionParser.add_option()`, como cadenas de opción no válidas, atributos de opción desconocidos, atributos de opción que faltan, etc. Se tratan de la forma habitual: generan una excepción (ya sea `optparse.OptionError` o `TypeError`) y hacen que el programa se bloquee.

Manejar los errores del usuario es mucho más importante, ya que está garantizado que sucederán sin importar cuán estable sea el código. `optparse` puede detectar automáticamente algunos errores del usuario, como argumentos de opción incorrectos (pasar `-n 4x` donde `-n` toma un argumento entero) o la falta de argumentos (`-n` al final de la línea de comandos, donde `-n` toma un argumento de cualquier tipo). Además de esto, puedes llamar a `OptionParser.error()` para establecer una condición de error definida por la propia aplicación:

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

En cualquier caso, `optparse` maneja el error de la misma manera: imprime el mensaje de uso del programa junto a un mensaje de error en la salida de errores estándar y termina la ejecución con el estado de error 2.

Considera el primero de los dos ejemplos anteriores, donde el usuario pasa `4x` a una opción que toma un número entero:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

O, en el caso en el que el usuario definitivamente no pase ningún valor:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

Los mensajes de error generados por *optparse* tienen siempre cuidado de mencionar la opción involucrada en el error. Asegúrate de hacer lo mismo cuando llames a `OptionParser.error()` desde el código de tu aplicación.

Si el comportamiento del manejo de errores predeterminado de *optparse* no se adapta a tus necesidades, deberás subclassificar `OptionParser` y redefinir su método `exit()` y/o `error()`.

Reuniendo todas las piezas

Así es como los scripts basados en *optparse* usualmente se ven:

```
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")
    ...
    (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()
```

36.1.3 Guía de referencia

Creando el analizador sintáctico (parser)

El primer paso para poder usar *optparse* es crear una instancia de `OptionParser`.

class `optparse.OptionParser(...)`

El constructor de la clase `OptionParser` no tiene argumentos obligatorios, pero sí varios argumentos opcionales por palabra clave. Siempre deben pasarse como argumentos por palabras clave, es decir, no se debe confiar nunca en el orden en que se declaran los argumentos.

usage (por defecto: `"%prog [options]"`) El resumen de uso a imprimir cuando el programa se ejecuta incorrectamente o con una opción de ayuda. Cuando *optparse* imprime la cadena de uso, reemplaza `%prog` con `os.path.basename(sys.argv[0])` (o con `prog` si se proporcionó eso argumento por palabra clave). Para suprimir un mensaje de uso, se debe pasar el valor especial `optparse.SUPPRESS_USAGE`.

option_list (por defecto: []) Una lista de objetos `Option` con los que poblar el analizador. Las opciones en `option_list` se agregan después de cualquier opción en `standard_option_list` (un atributo de clase que puede ser establecido por las subclases de `OptionParser`), pero antes de cualquier versión u opción de ayuda. Obsoleto: usar en su lugar el método `add_option()`, una vez creado el analizador.

option_class (por defecto: `optparse.Option`) Clase usada por el método `add_option()` para añadir opciones al analizador.

version (por defecto: `None`) Una cadena de caracteres con la versión del programa a imprimir cuando el usuario proporciona una opción de versión. Si se proporciona un valor verdadero para `version`, `optparse` agrega automáticamente una opción de versión con `--version` como única cadena de opción. La subcadena `%prog` se expande de la misma manera que en `usage`.

conflict_handler (por defecto: `"error"`) Especifica qué hacer cuando se agregan al analizador opciones con cadenas de opción en conflicto entre sí. Ver sección *Conflictos entre opciones*.

description (por defecto: `None`) Un párrafo de texto que ofrece una breve descripción del programa. `optparse` reformatea este párrafo para que se ajuste al ancho actual de la terminal y lo imprime cuando el usuario solicita ayuda (después de `usage`, pero antes de la lista de opciones).

formatter (por defecto: una nueva instancia de la clase `IndentedHelpFormatter`) Una instancia de la clase `optparse.HelpFormatter` que será usada para imprimir el texto de ayuda. El módulo `optparse` proporciona dos clases concretas para este propósito: `IndentedHelpFormatter` y `TitledHelpFormatter`.

add_help_option (por defecto: `True`) Si es verdadero, `optparse` agregará al analizador una opción de ayuda (con las cadenas de opción `-h` y `--help`).

prog La cadena de caracteres a usar como substituta de `os.path.basename(sys.argv[0])` cuando se expanda `%prog` en `usage` y en `version`.

epilog (por defecto: `None`) Un párrafo con texto de ayuda que se imprimirá después de la opción de ayuda.

Completando el analizador con opciones

Hay varias formas de agregar las opciones al analizador. La forma preferida es mediante el método `OptionParser.add_option()`, tal como se muestra en la sección del *Tutorial*. El método `add_option()` se puede llamar de dos formas diferentes:

- pasándole una instancia de `Option` (como la que retorna `make_option()`)
- pasándole cualquier combinación de argumentos posicionales y por palabra clave que sean aceptables para `make_option()` (es decir, para el constructor de la clase `Option`), lo que creará la instancia `Option` automáticamente

La otra alternativa es pasar una lista de instancias de `Option` previamente construidas al constructor `OptionParser`, como en el siguiente ejemplo:

```
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()` es una función fábrica para crear instancias `Option`. Actualmente es simplemente un alias para el constructor `Option`, pero es posible que una futura versión de `optparse` pueda dividir `Option` en varias clases, en cuyo caso, `make_option()` elegirá la clase correcta a instanciar. Esta es la razón por la que no se debe instanciar `Option` directamente.)

Definiendo las opciones

Cada instancia de `Option` representa un conjunto de cadenas de opción sinónimas para la línea de comandos, por ejemplo `-f` y `--file`. Se puede especificar cualquier número de cadenas de opción cortas o largas, pero se debe proporcionar al menos una cadena de opción en total.

La forma canónica de crear una instancia de `Option` es mediante el método `add_option()` de la clase `OptionParser`.

```
OptionParser.add_option(option)
```

```
OptionParser.add_option(*opt_str, attr=value, ...)
```

Para definir una opción con solo una cadena de opción corta:

```
parser.add_option("-f", attr=value, ...)
```

Y para definir una opción con solo una cadena de opción larga:

```
parser.add_option("--foo", attr=value, ...)
```

Los argumentos por palabra clave definen atributos para el nuevo objeto `Option`. El atributo de opción más importante es `action`, que determina en gran medida qué otros atributos son apropiados u obligatorios. Si se pasan atributos de opción no apropiados, o no se pasan los requeridos, `optparse` lanza una excepción `OptionError` explicando el error.

La acción (*action*) de una opción determina que hace el módulo `optparse` cuando encuentra dicha opción en la línea de comandos. Las acciones de opción estándares codificadas en `optparse` son:

"store" almacena el argumento de esta opción (por defecto)

"store_const" almacena un valor constante

"store_true" almacena `True`

"store_false" almacena `False`

"append" agrega el argumento de esta opción a una lista

"append_const" agrega un valor constante a una lista

"count" incrementa un contador en uno

"callback" llama a una función específica

"help" imprime un mensaje de uso que incluye todas las opciones y la documentación correspondiente

(Si no se proporciona una acción, el valor predeterminado es `"store"`. Para esta acción en concreto, también se pueden proporcionar los atributos de opción `type` y `dest`. Consultar *Acciones de opción estándares* para más información.)

Como se puede observar, la mayoría de las acciones implican almacenar o actualizar un valor en algún lugar. `optparse` siempre crea un objeto especial para esto, convencionalmente llamado `options` (que resulta ser una instancia de `optparse.Values`). Los argumentos de opción (y algunos otros valores) se almacenan como atributos de este objeto, de acuerdo con el atributo de opción `dest` (destino) establecido.

Por ejemplo, cuando se llama a:

```
parser.parse_args()
```

una de las primeras cosas que hace el módulo `optparse` es crear el objeto `options`:

```
options = Values()
```

Si una de las opciones de este analizador es definida con:

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

y la línea de comandos que se analiza incluye cualquiera de las siguientes variantes:

```
-ffoo
-f foo
--file=foo
--file foo
```

entonces el módulo `optparse`, al encontrar esta opción, hará algo equivalente a:

```
options.filename = "foo"
```

Los atributos de opción `type` y `dest` son casi tan importantes como `action`, pero `action` es el único de ellos que es apropiado para *todas* las opciones.

Atributos de opción

Los siguientes atributos de opción pueden pasarse como argumentos por palabra clave al método `OptionParser.add_option()`. Si se pasa un atributo de opción que no es apropiado para una opción en particular, o no se pasa un atributo de opción obligatorio, `optparse` lanza una excepción `OptionError`.

`Option.action`

(por defecto: "store")

Determina el comportamiento del módulo `optparse` cuando esta opción se encuentra en la línea de comandos. Las opciones disponibles están documentadas [aquí](#).

`Option.type`

(por defecto: "string")

El tipo de argumento esperado para esta opción (por ejemplo, "string" o "int"). Los tipos de opción disponibles están documentados [aquí](#).

`Option.dest`

(por defecto: derivado de las cadenas de opción)

Si la acción asociada a la opción implica escribir o modificar un valor en algún lugar, este atributo le dice a `optparse` dónde escribirlo: `dest` es el nombre de un atributo del objeto `options` que `optparse` construye a medida que analiza la línea de comandos.

`Option.default`

El valor que se utilizará como destino de esta opción si la opción no aparece en la línea de comandos. Ver también `OptionParser.set_defaults()`.

`Option.nargs`

(por defecto: 1)

Cuántos argumentos de tipo `type` deben consumirse cuando se encuentre esta opción. Si es mayor a 1, `optparse` almacenará una tupla con los valores de los argumentos en `dest`.

`Option.const`

Para acciones que almacenan un valor constante, el valor constante a almacenar.

`Option.choices`

Para opciones de tipo "choice", la lista con las cadenas que el usuario puede elegir.

`Option.callback`

Para las opciones con la acción "callback" asignada, el objeto invocable a llamar cuando se encuentra esta opción. Consultar la sección [Retrollamadas de opción](#) para obtener más detalles sobre los argumentos que son pasados al objeto invocable.

`Option.callback_args`

`Option.callback_kwargs`

Argumentos posicionales y por palabra clave adicionales para ser pasados a `callback` después de los cuatro argumentos pasados a la retrollamada estándar.

Option.help

Texto de ayuda a imprimir para esta opción cuando se enumeran todas las opciones disponibles, después de que el usuario proporcione una opción *help* (como `--help`). Si no se proporciona ningún texto de ayuda, la opción seguirá apareciendo, solo que sin texto de ayuda. Para ocultar esta opción completamente, se debe asignar al atributo el valor especial `optparse.SUPPRESS_HELP`.

Option.metavar

(por defecto: derivado de las cadenas de opción)

Reemplazo para los argumentos de opción que se utilizará al imprimir el texto de ayuda. Consultar la sección *Tutorial* para ver un ejemplo.

Acciones de opción estándares

Las diversas acciones de opción tienen requisitos y efectos ligeramente diferentes. La mayoría de las acciones tienen varios atributos de opción relacionados que puedes especificar para dirigir el comportamiento del módulo *optparse*. Además, algunas de ellas tienen atributos requeridos, que se deben especificar para cualquier opción que utilice esa acción.

- "store" [atributos relacionados: *type*, *dest*, *nargs*, *choices*]

La opción debe ir seguida de un argumento, que se convierte en un valor de acuerdo a *type* y se almacena en *dest*. Si *nargs* es mayor que 1, se consumirán varios argumentos de la línea de comandos. Todos ellos se convertirán de acuerdo a *type* y se almacenarán en *dest* como una tupla. Consultar la sección *Tipos de opción estándares* para más información.

Si se proporciona *choices* (una lista o tupla de cadenas de caracteres), el tipo por defecto es "choice".

Si no se proporciona *type*, el tipo por defecto es "string".

Si *dest* no se proporciona, *optparse* infiere un destino de la primera cadena de opción larga (por ejemplo, `--foo-bar` implica que se usará `foo_bar` como destino). Si no hay cadenas de opción largas, *optparse* obtiene el destino a partir de la primera cadena de opción corta (por ejemplo, `-f` conlleva que se usará `f`).

Ejemplo:

```
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3, dest="point")
```

Mientras analiza la línea de comandos:

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

optparse establecerá:

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- "store_const" [atributo requerido: *const*; atributo relacionado: *dest*]

El valor *const* es almacenado en *dest*.

Ejemplo:

```
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")
```

Si se encuentra `--noisy`, *optparse* establecerá:

```
options.verbose = 2
```

- "store_true" [atributo relacionado: *dest*]

Un caso especial de "store_const" que almacena True en *dest*.

- "store_false" [atributo relacionado: *dest*]

Como "store_true", pero almacena False.

Ejemplo:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [atributos relacionados: *type*, *dest*, *nargs*, *choices*]

La opción debe ir seguida de un argumento, que se añade a la lista de *dest*. Si no se proporciona un valor predeterminado para *dest*, se crea automáticamente una lista vacía cuando *optparse* encuentra por primera vez esta opción en la línea de comandos. Si *nargs* es mayor que 1, se consumen varios argumentos y se agrega una tupla de longitud *nargs* a *dest*.

Los valores por defecto para los atributos *type* y *dest* son los mismos que para la acción "store".

Ejemplo:

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

Si se encuentra `-t3` en la línea de comandos, *optparse* procede de forma equivalente a:

```
options.tracks = []
options.tracks.append(int("3"))
```

Si, un poco más adelante, se encuentra `--tracks=4`, procede así:

```
options.tracks.append(int("4"))
```

La acción `append` llama al método `append` con el valor actual de la opción. Esto significa que cualquier valor por defecto especificado debe tener un método `append`. También significa que si existe un valor por defecto, los elementos por defecto estarán presentes en el valor analizado para la opción, con todos los valores de la línea de comandos agregados a la lista a continuación de ellos:

```
>>> parser.add_option("--files", action="append", default=['~/mypkg/defaults', '~/.mypkg/defaults'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/mypkg/defaults', 'overrides.mypkg']
```

- "append_const" [atributo requerido: *const*; atributo relacionado: *dest*]

Igual que "store_const", pero el valor *const* se agrega a *dest*. Como ocurre con "append", *dest* por defecto es None y se crea automáticamente una lista vacía la primera vez que se encuentra la opción en la línea de comandos.

- "count" [atributo relacionado: *dest*]

Incrementa el entero almacenado en *dest*. Si no se proporciona un valor por defecto, *dest* se establece en cero antes de incrementarse por primera vez.

Ejemplo:

```
parser.add_option("-v", action="count", dest="verbosity")
```

La primera vez que se encuentra `-v` en la línea de comandos, *optparse* procede de forma equivalente a:

```
options.verbosity = 0
options.verbosity += 1
```

Cada aparición posterior de `-v` da como resultado:

```
options.verbosity += 1
```

- "callback" [atributo requerido: `callback`; atributos relacionados: `type`, `nargs`, `callback_args`, `callback_kwargs`]

Llama a la función especificada por `callback`, que es llamada como:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

Ver sección *Retrollamadas de opción* para más detalles.

- "help"

Imprime un mensaje de ayuda completo para todas las opciones presentes en el analizador de opciones actual. El mensaje de ayuda se construye a partir de la cadena `usage`, pasada al constructor de `OptionParser`, y la cadena `help`, pasada a cada opción.

Si no se proporciona una cadena `help` para una opción, dicha opción seguirá apareciendo en el mensaje de ayuda. Para omitir una opción por completo, debe usarse el valor especial `optparse.SUPPRESS_HELP`.

El módulo `optparse` agrega automáticamente una opción `help` a todos los `OptionParsers`, por lo que normalmente no es necesario crear una.

Ejemplo:

```
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)
```

Si `optparse` encuentra `-h` o `--help` en la línea de comandos, imprimirá un mensaje de ayuda en la salida estándar como el siguiente (asumiendo que `sys.argv[0]` es `"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
```

Después de imprimir el mensaje de ayuda, `optparse` termina su proceso con `sys.exit(0)`.

- "version"

Imprime el número de versión proporcionado a `OptionParser` en la salida estándar y termina la ejecución. El número de versión en realidad es formateado e impreso por el método `print_version()` de `OptionParser`. Generalmente solo es relevante si el argumento `version` se proporciona al constructor de `OptionParser`. Al igual que en el caso de las opciones `help`, rara vez será necesario crear opciones de `version`, ya que `optparse` las agrega automáticamente cuando es necesario.

Tipos de opción estándares

El módulo `optparse` proporciona cinco tipos de opción incorporados: `"string"`, `"int"`, `"choice"`, `"float"` y `"complex"`. Consultar la sección [Extendiendo el módulo `optparse`](#) si se necesitan agregar nuevos tipos de opción.

Los argumentos de las opciones en la cadena ingresada no se verifican ni se convierten de ninguna manera: el texto de la línea de comandos se almacena en el destino (o se pasa a la retrollamada) tal cual.

Los argumentos enteros (tipo `"int"`) se analizan de la siguiente manera:

- si el número comienza con `0x`, se analiza como un número hexadecimal
- si el número comienza con `0`, se analiza como un número octal
- si el número comienza con `0b`, se analiza como un número binario
- en cualquier otro caso, el número se analiza como un número decimal

La conversión se realiza llamando a `int()` con la base apropiada (2, 8, 10 o 16). Si esto falla, también lo hará `optparse`, aunque mostrando un mensaje de error más útil para el usuario.

Los argumentos de las opciones de tipo `"float"` y `"complex"` se convierten directamente usando `float()` y `complex()` respectivamente, con un manejo de errores similar.

Las opciones de tipo `"choice"` son un subtipo de las opciones `"string"`. El atributo de opción `choices` (que es una secuencia de cadenas) define el conjunto de argumentos de opción permitidos. Posteriormente, `optparse.check_choice()` comparará los argumentos de las opciones proporcionadas por el usuario con esta lista maestra y generará una excepción `OptionValueError` si se proporciona una cadena no válida.

Analizando los argumentos

El objetivo primario de crear y agregar opciones a un `OptionParser` es llamar a su método `parse_args()`:

```
(options, args) = parser.parse_args(args=None, values=None)
```

donde los parámetros de entrada son:

args la lista de argumentos a procesar (por defecto: `sys.argv [1:]`)

values un objeto de la clase `optparse.Values` para almacenar en él los argumentos de las opciones. Por defecto es una nueva instancia de la clase `Values`. Si se proporciona un objeto previamente creado, los valores predeterminados de la opción no se inicializarán en el mismo

y los valores de retorno son:

options el mismo objeto que se pasó como `values`, o la instancia `optparse.Values` creada por `optparse`

args los argumentos posicionales que quedan en la línea de comandos después de que se hayan procesado todas las opciones

El uso más habitual es no proporcionar ningún argumento por palabra clave. Si se proporciona `values`, dicho argumento será modificado mediante llamadas repetidas a `setattr()` (aproximadamente una por cada argumento de opción a almacenar en un destino de opción) y finalmente será retornado por el método `parse_args()`.

Si el método `parse_args()` encuentra algún error en la lista de argumentos, llama al método `error()` de `OptionParser` con un mensaje de error apropiado para el usuario final. Esto causa que el proceso termine con un estado de salida de 2 (el estado de salida tradicional en Unix para errores en la línea de comandos).

Consultar y manipular el analizador de opciones

El comportamiento predeterminado del analizador de opciones se puede personalizar ligeramente. También se puede indagar en el analizador de opciones y ver qué hay en él. `OptionParser` proporciona varios métodos para ayudar con éstos propósitos:

`OptionParser.disable_interspersed_args()`

Configura el análisis para que se detenga en lo primero que encuentre que no sea una opción. Por ejemplo, si `-a` y `-b` son opciones simples que no toman argumentos, *optparse* normalmente acepta esta sintaxis:

```
prog -a arg1 -b arg2
```

y la trata de forma equivalente a:

```
prog -a -b arg1 arg2
```

Para deshabilitar esta funcionalidad, se debe llamar al método `disable_interspersed_args()`. Esto restaura la sintaxis tradicional usada en Unix, donde el análisis de opción se detiene con el primer argumento que no es una opción.

Se debe usar este método si se dispone de un procesador de comandos que ejecuta otro comando con sus propias opciones y se desea asegurarse de que estas opciones no se confunden entre sí. Lo que puede ocurrir si, por ejemplo, cada comando tiene un conjunto diferente de opciones.

`OptionParser.enable_interspersed_args()`

Configura el análisis para que no se detenga si encuentra un argumento que no sea una opción, lo que permite intercalar modificadores con argumentos de línea de comandos. Este es el comportamiento por defecto.

`OptionParser.get_option(opt_str)`

Retorna la instancia de `Option` con la cadena de opción `opt_str`, o `None` si ninguna opción tiene esa cadena de opción.

`OptionParser.has_option(opt_str)`

Retorna `True` si `OptionParser` tiene una opción con la cadena de opción `opt_str` (por ejemplo, `-q` o `--verbose`).

`OptionParser.remove_option(opt_str)`

Si *OptionParser* tiene una opción correspondiente a `opt_str`, esa opción es eliminada. Si esa opción proporcionó cualquier otra cadena de opción, todas esas cadenas de opción quedan invalidadas. Si `opt_str` no aparece en ninguna opción que pertenezca a este *OptionParser*, se lanza una excepción `ValueError`.

Conflictos entre opciones

Si no se tiene cuidado, es fácil definir opciones con cadenas de opción en conflicto entre sí:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(Esto es particularmente cierto si se ha definido una subclase propia de `OptionParser` con algunas opciones estándar.)

Cada vez que se agrega una opción, *optparse* comprueba si existen conflictos con las opciones ya existentes. Si encuentra alguno, invoca el mecanismo de manejo de conflictos actualmente establecido. Se puede establecer el mecanismo de manejo de conflictos desde el propio constructor:

```
parser = OptionParser(..., conflict_handler=handler)
```

o mediante una llamada separada:

```
parser.set_conflict_handler(handler)
```

Los administradores de conflictos disponibles son:

"error" (por defecto) se asume que los conflictos entre opciones son un error de programación y, por tanto, generarán una excepción `OptionConflictError`

"resolve" resuelve conflictos de opciones de forma inteligente (ver más abajo)

Como ejemplo, vamos a definir un `OptionParser` que resuelva conflictos de manera inteligente y agregaremos algunas opciones conflictivas:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

En este punto, `optparse` detecta que una opción agregada anteriormente ya está usando la cadena de opción `-n`. Dado que `conflict_handler` es `"resolve"`, resuelve la situación eliminando `-n` de la lista de cadenas de opción de la opción previa. Ahora `--dry-run` es la única forma que el usuario tiene para poder activar esa opción. Si el usuario solicita ayuda, el mensaje de ayuda reflejará la nueva situación:

```
Options:
  --dry-run      do no harm
  ...
  -n, --noisy    be noisy
```

Es posible reducir las cadenas de opción para una opción agregada previamente hasta que no quede ninguna, de forma que el usuario no tenga forma de invocar esa opción desde la línea de comandos. En ese caso, `optparse` eliminará esa opción por completo, por lo que no aparecerá en el texto de ayuda ni en ningún otro lugar. Continuando con nuestro analizador de opciones previo:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

En este punto, la opción original `-n/--dry-run` ya no es accesible, por lo que `optparse` la elimina, dejando el siguiente texto de ayuda:

```
Options:
  ...
  -n, --noisy    be noisy
  --dry-run      new dry-run option
```

Limpieza

Las instancias de `OptionParser` tienen varias referencias cíclicas. Esto no debería ser un problema para el recolector de basura de Python, pero es posible que se desee romper las referencias cíclicas explícitamente llamando al método `destroy()` de la instancia `OptionParser` una vez que se haya terminado. Esto es particularmente útil en aplicaciones de larga ejecución en las que `OptionParser` puede terminar accediendo a grafos de objetos considerablemente grandes.

Otros métodos

`OptionParser` admite varios métodos públicos más:

`OptionParser.set_usage(usage)`

Establece la cadena de caracteres de uso de acuerdo a las reglas descritas anteriormente para el argumento por palabra clave `usage` del constructor. Si se pasa `None` se establece la cadena de uso por defecto. Para suprimir el mensaje de uso totalmente, se debe pasar el valor especial `optparse.SUPPRESS_USAGE`.

`OptionParser.print_usage(file=None)`

Imprime el mensaje de uso del programa actual (`self.usage`) en `file` (que por defecto es la salida estándar). Cualquier aparición de la cadena de caracteres `%prog` en `self.usage` es reemplazada con el nombre del programa actual. No hace nada si `self.usage` está vacío o no ha sido definido.

`OptionParser.get_usage()`

Igual que `print_usage()` pero retorna la cadena de uso en vez de imprimirla.

`OptionParser.set_defaults(dest=value, ...)`

Establece valores por defecto para varios destinos de opción a la vez. Usar el método `set_defaults()` es la forma preferida de establecer valores por defecto para las opciones, ya que varias opciones pueden compartir el mismo destino. Por ejemplo, si varias opciones de `mode` tienen el mismo destino, cualquiera de ellas puede establecer el valor por defecto, pero el último establecido es el que finalmente queda establecido:

```
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
```

Para evitar esta confusión, usa el método `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")
```

36.1.4 Retrollamadas de opción

Cuando las acciones y tipos incorporados de `optparse` no son suficientes para tus necesidades, tienes dos opciones: extender `optparse` o definir una opción con una retrollamada asociada. Extender `optparse` es más general, pero algo exagerado para muchos casos simples. Es frecuente que una simple retrollamada sea todo lo que necesitas.

Hay dos pasos a seguir para definir una opción con retrollamada:

- definir la opción en sí usando la acción `"callback"`
- escribir la retrollamada. Esta es una función (o método) que toma al menos cuatro argumentos, los cuales son descritos a continuación

Definición de una opción con retrollamada

Generalmente, la forma más sencilla de definir una opción con retrollamada es mediante el método `OptionParser.add_option()`. Aparte de `action`, el único atributo de opción que debes especificar es `callback`, que es la función a llamar:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

`callback` es una función (u otro objeto invocable), lo que implica que `my_callback()` debe haber sido definida antes de la creación de esta opción con retrollamada. En este caso simple, `optparse` ni siquiera sabe si `-c` toma algún argumento, lo que generalmente significa que la opción no toma argumentos—la mera presencia de `-c` en la línea de comandos es todo lo que necesita saber. Sin embargo, en algunas circunstancias, es posible que se desee que la retrollamada consuma una cantidad arbitraria de argumentos de la propia línea de comandos. Es aquí donde escribir retrollamadas se vuelve complicado; se tratará más adelante en esta sección.

`optparse` siempre pasa cuatro argumentos concretos a la retrollamada. El método solo pasará argumentos adicionales si se especifica explícitamente a través de `callback_args` y `callback_kwargs`. Por lo tanto, la firma mínima de la retrollamada es:

```
def my_callback(option, opt, value, parser):
```

Los cuatro argumentos para la retrollamada se describen a continuación.

Hay varios atributos de opción adicionales que se pueden proporcionar cuando se define una opción con retrollamada:

type tiene su significado habitual: al igual que en las acciones "store" o "append", indica a *optparse* que debe consumir un argumento y convertirlo a *type*. Sin embargo, en lugar de almacenar el valor (o valores) convertido en algún lugar, *optparse* lo pasa a la retrollamada.

nargs también tiene su significado habitual: si se proporciona y es mayor que 1, *optparse* consumirá *nargs* argumentos de la línea de comandos, cada uno de los cuales debe ser convertible a *type*. Hecho esto, se pasa una tupla con los valores convertidos a la retrollamada.

callback_args una tupla con los argumentos posicionales adicionales para pasar a la retrollamada

callback_kwargs un diccionario con los argumentos por palabra clave para pasar a la retrollamada

Cómo son invocadas las retrollamadas

Todas las retrollamadas son invocadas de la siguiente forma:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

donde

option es la instancia de *Option* que invoca a la retrollamada

opt_str es la cadena de opción encontrada en la línea de comandos que activa la retrollamada. (Si se utilizó una opción larga abreviada, *opt_str* será la cadena de opción canónica completa— por ejemplo, si el usuario ingresa `--foo` en la línea de comandos como una abreviatura de `--foobar`, entonces *opt_str* será `--foobar`.)

value es el argumento de esta opción encontrado en la línea de comandos. *optparse* solo esperará un argumento si *type* está establecido. El tipo de *value* será el tipo implícito en el tipo de la propia opción. Si *type* es *None* para esta opción (no se espera ningún argumento), entonces *value* será también *None*. Si *nargs* es mayor que 1, *value* será una tupla con los valores del tipo apropiado.

parser es la instancia de *OptionParser* que controla todo. Su utilidad principal radica en que permite acceder a otros datos de interés a través de sus atributos de instancia:

parser.largs la lista actual de argumentos que sobran, es decir, argumentos que se han consumido pero que no son opciones ni argumentos de opción. Siéntete libre de modificar *parser.largs*, por ejemplo, agregando más argumentos. (Esta lista se convertirá en *args*, el segundo valor de retorno del método *parse_args()*.)

parser.rargs la lista actual de argumentos restantes, es decir, los argumentos que quedan a continuación de *opt_str* y *value* (si corresponde), una vez eliminados ambos. Siéntete libre de modificar *parser.rargs*, por ejemplo, consumiendo más argumentos.

parser.values el objeto donde los valores de las opciones son almacenados por defecto (una instancia de *optparse.OptionValues*). Esto permite que las retrollamadas utilicen el mismo mecanismo que el resto de *optparse* para almacenar valores de las opciones; no es necesario perder el tiempo con globales o clausuras. También se puede acceder a los valores de las opciones o modificarlos si ya se encuentran en la línea de comandos.

args es una tupla de argumentos posicionales arbitrarios suministrados a través del atributo de opción *callback_args*.

kwargs es un diccionario con argumentos por palabra clave arbitrarios proporcionados por *callback_kwargs*.

Lanzando errores en una retrollamada

La retrollamada debería lanzar una excepción `OptionValueError` si hay algún problema con la opción o su(s) argumento(s). Esto permite a `optparse` detectar el problema y finalizar el programa, imprimiendo el mensaje de error que hayas proporcionado en la salida de error estándar. El mensaje debe ser claro, conciso, preciso y mencionar la opción que causa la excepción. De lo contrario, el usuario tendrá dificultades para descubrir qué hizo mal.

Ejemplo de retrollamada 1: una retrollamada trivial

Aquí hay un ejemplo de una opción con retrollamada que no tiene argumentos y simplemente registra que se encontró la opción en la línea de comandos:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Ciertamente, se puede hacer lo mismo simplemente con la acción `"store_true"`.

Ejemplo de retrollamada 2: comprobar el orden de las opciones

Aquí tenemos un ejemplo un poco más interesante: registra el hecho de que se ha encontrado `-a`, pero lanza un error si viene después de `-b` en la línea de comandos

```
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")
```

Ejemplo de retrollamada 3: comprobar el orden de las opciones (generalizado)

Si deseas reutilizar esta misma retrollamada para varias opciones similares (establecer un flag, pero lanzar un error si ya se ha encontrado `-b`), necesitas un poco más de trabajo: tanto el mensaje de error como el flag que estableces deben generalizarse:

```
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')
```

Ejemplo de retrollamada 4: comprobar una condición arbitraria

Por supuesto, puedes poner cualquier condición aquí, no estás limitado a verificar los valores de las opciones previamente definidas. Por ejemplo, si tienes opciones que no deberían llamarse cuando hay luna llena, todo lo que tienes que hacer es lo siguiente:

```
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")
```

(La definición de `is_moon_full()` se deja como ejercicio para el lector).

Ejemplo de retrollamada 5: argumentos fijos

Las cosas se ponen un poco más interesantes cuando se definen opciones con retrollamada que toman un número fijo de argumentos. Especificar que una opción con retrollamada toma argumentos es similar a definir una opción "store" o "append": si se define `type`, entonces la opción toma un argumento que debe poder convertirse a ese tipo; si además se define `nargs`, entonces la opción toma `nargs` argumentos.

Aquí hay un ejemplo que simplemente emula la acción "store" estándar:

```
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")
```

Ten en cuenta que `optparse` se encarga de consumir 3 argumentos y convertirlos a números enteros por ti; todo lo que tienes que hacer es almacenarlos. (En cualquier caso, obviamente no necesitas hacer uso de una retrollamada para este ejemplo).

Ejemplo de retrollamada 6: argumentos variables

Las cosas se complican si quieres que una opción pueda tomar un número variable de argumentos. En este caso, si que deberás escribir una retrollamada, ya que el módulo `optparse` no proporciona ninguna capacidad incorporada para ello. Además, tienes que lidiar con ciertos entresijos del análisis convencional de la línea de comandos de Unix, que `optparse` normalmente maneja por ti. En concreto, las retrollamadas deben implementar las reglas convencionales para los argumentos `--` y `-` desnudos:

- tanto `--` como `-` pueden ser argumentos de opción
- `--` desnudo (si no es el argumento de alguna opción): detener el procesamiento de la línea de comandos y descartar el `--`
- `-` desnudo (si no es el argumento de alguna opción): detener el procesamiento de la línea de comandos pero mantener el `-` (añadiéndolo a `parser.largs`)

Si deseas que una opción tenga un número variable de argumentos, hay varios problemas sutiles y complicados de los que deberás preocuparte. La implementación exacta que elijas dependerá de los sacrificios que estés dispuesto a hacer en tu aplicación (razón por la cual, el módulo `optparse` no admite directamente este tipo de cosas).

En cualquier caso, aquí hay un intento de una retrollamada para una opción con un número de argumentos variable:

```
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)
```

36.1.5 Extendiendo el módulo `optparse`

Dado que los dos factores principales que controlan como el módulo `optparse` interpreta las opciones de la línea de comandos son la acción y el tipo de cada opción, los objetivos más probables de extensión son agregar nuevas acciones y nuevos tipos.

Agregando nuevos tipos

Para añadir nuevos tipos, necesitas definir tu propia subclase de la clase `Option` de `optparse`. Esta clase tiene un par de atributos que definen los tipos de `optparse`: `TYPES` y `TYPE_CHECKER`.

`Option.TYPES`

Una tupla con nombres de tipos. En tu subclase, simplemente define una nueva tupla `TYPES` que se base en la estándar.

`Option.TYPE_CHECKER`

Un diccionario que asigna nombres de tipos a funciones de verificación de tipo. Una función de verificación de tipo tiene la siguiente firma:

```
def check_mytype(option, opt, value)
```

donde `option` es una instancia de `Option`, `opt` es una cadena de opción (por ejemplo, `-f`) y `value` es la cadena de caracteres de la línea de comandos que debe comprobarse y convertirse al tipo deseado. `check_mytype()` debería retornar un objeto del tipo hipotético `mytype`. El valor retornado por una función de verificación de tipo terminará formando parte de la instancia de `OptionValues` retornada por el método `OptionParser.parse_args()` o será pasada a una retrollamada como parámetro `value`.

Tu función de verificación de tipo debería lanzar una excepción `OptionValueError` si encuentra algún problema. `OptionValueError` toma una cadena de caracteres como único argumento, que es pasada tal cual al método `error()` de la clase `OptionParser`, que a su vez antepone a la misma el nombre del programa y la cadena `"error: "` e imprime todo en la salida de error estándar antes de finalizar el proceso.

Aquí hay un ejemplo absurdo que demuestra cómo agregar un tipo de opción "complex" para analizar números complejos al estilo Python en la línea de comandos. (Esto es aún más absurdo de lo que en principio parece, porque en la versión 1.3 de *optparse* se agregó soporte incorporado para números complejos, pero no importa).

Primero, las importaciones necesarias:

```
from copy import copy
from optparse import Option, OptionValueError
```

En primer lugar, debes definir tu verificador de tipo, ya que se hace referencia a él más adelante (en el atributo de clase *TYPE_CHECKER* de tu subclase de *Option*):

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

Finalmente, la subclase de *Option*:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(Si no hacemos una copia (mediante *copy()*), de *Option.TYPE_CHECKER*, terminaríamos modificando el atributo *TYPE_CHECKER* de la clase *Option* de *optparse*. Tratándose de Python, nada te impide hacer eso, excepto los buenos modales y el sentido común.)

¡Y eso es todo! Ahora puedes escribir un script que use tu nuevo tipo de opción como cualquier otro script basado en *optparse*, excepto que debes indicarle a tu *OptionParser* que use *MyOption* en lugar de *Option*:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternativamente, puedes crear tu propia lista de opciones y pasarla a *OptionParser*; si no usas *add_option()* de la manera anterior, no necesitas decirle a *OptionParser* qué clase de opción usar:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

Agregando nuevas acciones

Agregar nuevas acciones es un poco más complicado, dado que hay que comprender que *optparse* establece un par de categorías para las acciones:

Acciones «store» acciones que dan como resultado que *optparse* almacene un valor en un atributo de la instancia actual de *OptionValues*. Estas opciones requieren un atributo *dest* para que pueda ser proporcionado al constructor de *Option*.

Acciones «typed» acciones que toman un valor de la línea de comandos, esperando que sea de cierto tipo; o mejor dicho, una cadena de caracteres que se pueda convertir a un determinado tipo. Estas opciones requieren un atributo *type* para el constructor de *Option*.

Ambas categorías se solapan entre sí: algunas acciones *store* predeterminadas son "store", "store_const", "append" y "count", mientras que las acciones «typed» predeterminadas son "store", "append" y "callback".

Cuando agregas una acción, debes categorizarla enumerándola en al menos uno de los siguientes atributos de clase de la clase *Option* (todos ellos son listas de cadenas de caracteres):

Option.ACTIONS

Todas las acciones deben aparecer en ACTIONS.

Option.STORE_ACTIONS

Las acciones «store» también se enumeran aquí.

Option.TYPED_ACTIONS

Las acciones «typed» también se enumeran aquí.

Option.ALWAYS_TYPED_ACTIONS

Las acciones que siempre toman un tipo (es decir, aquellas cuyas opciones siempre toman un valor) también deben enumerarse aquí. La única consecuencia de esto es que *optparse* asigna el tipo predeterminado, *string*, a las opciones cuya acción se enumera en *ALWAYS_TYPED_ACTIONS* pero que no tienen asignado un tipo explícito.

Para implementar realmente tu nueva acción, debes redefinir el método `take_action()` de *Option*, implementando un nuevo método que reconozca tu acción.

Por ejemplo, agreguemos una nueva acción "extend". Esta es similar a la acción estándar "append", pero en lugar de tomar un solo valor de la línea de comandos y agregarlo a una lista existente, "extend" tomará múltiples valores en una sola cadena de caracteres delimitada por comas y amplía una lista previamente existente con ellos. Es decir, si `--names` es una opción "extend" de tipo "string", la línea de comandos

```
--names=foo,bar --names blah --names ding,dong
```

daría como resultado una lista como la siguiente:

```
["foo", "bar", "blah", "ding", "dong"]
```

De nuevo, definimos una subclase de *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:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

Detalles a tener en cuenta:

- "extend" espera un valor en la línea de comandos y también almacena ese valor en algún lugar, por lo que lo agregamos tanto a *STORE_ACTIONS* como a *TYPED_ACTIONS*.
- para asegurarnos de que *optparse* asigna el tipo predeterminado "string" a las acciones "extend", agregamos también la acción "extend" a *ALWAYS_TYPED_ACTIONS*.
- el método *MyOption.take_action()* solo se encarga de implementar esta nueva acción y posteriormente le pasa el control al método *Option.take_action()* para las acciones estándar de *optparse*.
- *values* es una instancia de la clase *optparse.parser.Values*, que proporciona el útil método *ensure_value()*. El método *ensure_value()* es en esencia lo mismo que *getattr()* pero con un mecanismo de seguridad agregado. Es llamado como:

```
values.ensure_value(attr, value)
```

Si el atributo *attr* de *values* no existe o es *None*, entonces *ensure_value()* primero lo establece en *value* y luego retorna el atributo actualizado. Esto es muy útil para acciones como "extend", "append" y

"count", dado que todas ellas acumulan datos en una variable y esperan que esa variable sea de cierto tipo (una lista para las dos primeras, un número entero para la última). Usar el método `ensure_value()` significa que los scripts que usan tu acción no tienen que preocuparse por establecer un valor predeterminado para los destinos de opción en cuestión; simplemente pueden dejar el valor predeterminado como `None` y `secure_value()` se encargará de que todo esté correcto cuando sea necesario.

36.2 `imp` — Acceda a import internamente

Código fuente: `Lib/imp.py`

Obsoleto desde la versión 3.4: The `imp` module is deprecated in favor of `importlib`.

Este módulo proporciona una interfaz a los mecanismos utilizados para implementar la sentencia `import`. Define las siguientes constantes y funciones:

`imp.get_magic()`

retorna el valor de cadena mágica utilizado para reconocer archivos de código compilados por bytes (`:file:".pyc"` archivos). (Este valor puede ser diferente para cada versión de Python.)

Obsoleto desde la versión 3.4: Utilice `importlib.util.MAGIC_NUMBER` en su lugar.

`imp.get_suffixes()`

retorna una lista de tuplas de 3 elementos, cada una de las que describe un tipo determinado de módulo. Cada triple tiene la forma `(suffix, mode, type)`, donde `suffix` es una cadena que se anexará al nombre del módulo para formar el nombre de archivo para buscar, `mode` es la cadena de modo para pasar a la función incorporada `open()` para abrir el archivo (esto puede ser `'r'` para archivos de texto o `'rb'` para archivos binarios), y `type`, que tiene uno de los valores `PY_SOURCE`, `PY_COMPILED`, o `C_EXTENSION`, que se describe a continuación.

Obsoleto desde la versión 3.3: Utilice las constantes definidas en `importlib.machinery` en su lugar.

`imp.find_module(name[, path])`

Intente encontrar el módulo `name`. Si se omite `path` o `None`, se busca la lista de nombres de directorio dados por `sys.path`, pero primero se buscan algunos lugares especiales: la función intenta encontrar un módulo integrado con el nombre dado (`C_BUILTIN`), a continuación, un módulo congelado (`PY_FROZEN`), y en algunos sistemas algunos otros lugares se buscan también (en Windows, se ve en el registro que puede apuntar a un archivo específico).

De lo contrario, `path` debe ser una lista de nombres de directorio; cada directorio se busca archivos con cualquiera de los sufijos retornados por `get_suffixes()` arriba. Los nombres no válidos de la lista se omiten silenciosamente (pero todos los elementos de lista deben ser cadenas).

Si la búsqueda se realiza correctamente, el valor retornado es una tupla de 3 elementos (`file`, `pathname`, `description`):

`file` es un abierto *file object* posicionado al principio, `pathname` es el nombre de ruta del archivo encontrado, y `description` es una tupla de 3 elementos tal como se encuentra en la lista retornada por `get_suffixes()` que describe el tipo de módulo encontrado.

Si el módulo no vive en un archivo, el archivo `file` retornado es `None`, `pathname` es la cadena vacía y la tupla `description` contiene cadenas vacías para su sufijo y modo; el tipo de módulo se indica como se indica entre paréntesis arriba. Si la búsqueda no se realiza correctamente, se provoca `ImportError`. Otras excepciones indican problemas con los argumentos o el entorno.

Si el módulo es un paquete, `file` es `None`, `pathname` es la ruta de acceso del paquete y el último elemento de la tupla `description` es `PKG_DIRECTORY`.

Esta función no controla los nombres de módulo jerárquico (nombres que contienen puntos). Para encontrar `P.M`, es decir, submódulo `M` del paquete `P`, use `find_module()` y `load_module()` para buscar y cargar el paquete `P`, y luego use `find_module()` con el argumento `path` establecido en `P.__path__`. Cuando `P` tenga un nombre punteado, aplique esta receta de forma recursiva.

Obsoleto desde la versión 3.3: Utilice `importlib.util.find_spec()` en su lugar a menos que se requiera compatibilidad con Python 3.3, en cuyo caso use `importlib.find_loader()`. Para ver el uso del caso anterior, consulte la sección [Examples](#) de la documentación `importlib`.

`imp.load_module(name, file, pathname, description)`

Cargue un módulo que fue encontrado anteriormente por `find_module()` (o por una búsqueda realizada de otro modo que produce resultados compatibles). Esta función hace más que importar el módulo: si el módulo ya estaba importado, se recargará el módulo! El argumento `name` indica el nombre completo del módulo (incluido el nombre del paquete, si se trata de un submódulo de un paquete). El argumento `file` es un archivo abierto y `pathname` es el nombre de archivo correspondiente; pueden ser `None` y `' '`, respectivamente, cuando el módulo es un paquete o no se carga desde un archivo. El argumento `description` es una tupla, como sería retornada por `get_suffixes()`, que describe qué tipo de módulo se debe cargar.

Si la carga se realiza correctamente, el valor retornado es el objeto modulo; de lo contrario, se produce una excepción (normalmente `ImportError`).

Importante: el autor de la llamada es responsable de cerrar el argumento `file`, si no era `None`, incluso cuando se genera una excepción. Esto se hace mejor usando una declaración `try ... finally`.

Obsoleto desde la versión 3.3: Si se utiliza anteriormente junto con `imp.find_module()` considere usar `importlib.import_module()`, de lo contrario utilice el cargador retornado por el reemplazo que eligió para `imp.find_module()`. Si llamó a `imp.load_module()` y funciones relacionadas directamente con argumentos de ruta de archivo, utilice una combinación de `importlib.util.spec_from_file_location()` y `importlib.util.module_from_spec()`. Consulte la sección [Examples](#) de la documentación `importlib` para obtener más información sobre los distintos enfoques.

`imp.new_module(name)`

retorna un nuevo objeto de módulo vacío denominado `name`. Este objeto es *not* insertado en `sys.modules`.

Obsoleto desde la versión 3.4: Utilice `importlib.util.module_from_spec()` en su lugar.

`imp.reload(module)`

Vuelva a cargar un *módulo* importado anteriormente. El argumento debe ser un objeto `module`, por lo que debe haberse importado correctamente antes. Esto es útil si ha editado el archivo de origen del módulo utilizando un editor externo y desea probar la nueva versión sin salir del intérprete de Python. El valor retornado es el objeto `module` (el mismo que el argumento `module`).

Cuando se ejecuta `reload(module)`:

- El código de los módulos de Python se vuelve a compilar y se vuelve a ejecutar el código de nivel de módulo, definiendo un nuevo conjunto de objetos que están enlazados a nombres en el diccionario del módulo. La función `init` de los módulos de extensión no se llama una segunda vez.
- Al igual que con todos los demás objetos de Python, los objetos antiguos solo se recuperan después de que sus recuentos de referencia caigan a cero.
- Los nombres del espacio de nombres del módulo se actualizan para que apunten a cualquier objeto nuevo o modificado.
- Otras referencias a los objetos antiguos (como nombres externos al módulo) no se rebotan para hacer referencia a los nuevos objetos y deben actualizarse en cada espacio de nombres donde se producen si se desea.

Hay una serie de otras advertencias:

Cuando se vuelve a cargar un módulo, se conserva su diccionario (que contiene las variables globales del módulo). Las redefiniciones de nombres invalidarán las definiciones antiguas, por lo que esto generalmente no es un problema. Si la nueva versión de un módulo no define un nombre previamente definido, la definición anterior permanece. Esta característica se puede utilizar en beneficio del módulo si mantiene una tabla global o caché de objetos — con una instrucción `try` puede probar la presencia de la tabla y omitir su inicialización si se desea:

```
try:
    cache
```

(continúe en la próxima página)

(proviene de la página anterior)

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

Es legal, aunque generalmente no es muy útil para recargar módulos incorporados o cargados dinámicamente, excepto para `sys`, `__main__` y `builtins`. En muchos casos, sin embargo, los módulos de extensión no están diseñados para inicializarse más de una vez y pueden fallar de forma arbitraria cuando se vuelven a cargar.

Si un módulo importa objetos de otro módulo utilizando `from... import ...`, llamando a `reload()` para el otro módulo no redefine los objetos importados de él — de una manera alrededor de esto es volver a ejecutar la sentencia `from`, otra es usar `import` y nombres calificados (`module.*name*`) en su lugar.

Si un módulo crea instancias de una clase, volver a cargar el módulo que define la clase no afecta a las definiciones de método de las instancias — siguen utilizando la definición de clase antigua. Lo mismo es cierto para las clases derivadas.

Distinto en la versión 3.3: Se basa en que tanto `__name__` como `__loader__` que se definen en el módulo que se está recargando en lugar de simplemente `__name__`.

Obsoleto desde la versión 3.4: Utilice `importlib.reload()` en su lugar.

Las siguientes funciones son comodidades para controlar las rutas de acceso de archivo **PEP 3147** compiladas en bytes.

Nuevo en la versión 3.2.

`imp.cache_from_source(path, debug_override=None)`

retorna la ruta de acceso **PEP 3147** al archivo compilado por bytes asociado con la ruta `path` de origen. Por ejemplo, si `path` es `/foo/bar/baz.py` el valor retornado sería `/foo/bar/__pycache__/baz.cpython-32.pyc` para Python 3.2. La cadena `cpython-32` proviene de la etiqueta mágica actual (consulte `get_tag()`; si `sys.implementation.cache_tag` no está definida entonces `NotImplementedError` se generará). Al pasar `True` o `False` para `debug_override` puede reemplazar el valor del sistema para `__debug__`, lo que da lugar a un código de bytes optimizado.

ruta no necesita existir.

Distinto en la versión 3.3: Lanza `NotImplementedError` cuando no se define `sys.implementation.cache_tag`.

Obsoleto desde la versión 3.4: Utilice `importlib.util.source_from_cache()` en su lugar.

Distinto en la versión 3.5: El parámetro `debug_override` ya no crea un archivo `.pyo`.

`imp.source_from_cache(path)`

Dada la ruta `path` a un nombre de archivo **PEP 3147**, retorne la ruta de acceso del archivo de código fuente asociada. Por ejemplo, si `path` es `/foo/bar/__pycache__/baz.cpython-32.pyc` la ruta retornada sería `/foo/bar/baz.py`. *path* no necesita existir, sin embargo, si no se ajusta al formato **PEP 3147**, se genera un `ValueError`. Si `sys.implementation.cache_tag` no está definido, se genera `NotImplementedError`.

Distinto en la versión 3.3: Provoca `NotImplementedError` cuando no se define `sys.implementation.cache_tag`.

Obsoleto desde la versión 3.4: Utilice `importlib.util.source_from_cache()` en su lugar.

`imp.get_tag()`

retorna la cadena de etiqueta mágica **PEP 3147** que coincida con esta versión del número mágico de Python, retornada por `get_magic()`.

Obsoleto desde la versión 3.4: Utilice `sys.implementation.cache_tag` directamente a partir de Python 3.3.

Las siguientes funciones ayudan a interactuar con el mecanismo de bloqueo interno del sistema de importación. La semántica de bloqueo de las importaciones es un detalle de implementación que puede variar de una versión a una. Sin embargo, Python garantiza que las importaciones circulares funcionen sin interbloqueos.

`imp.lock_held()`

retorna `True` si el bloqueo de importación global se mantiene actualmente, de lo contrario `False`. En plataformas sin subprocesos, siempre retorna `False`.

En plataformas con subprocesos, un subproceso que ejecuta una importación primero contiene un bloqueo de importación global y, a continuación, configura un bloqueo por módulo para el resto de la importación. Esto impide que otros subprocesos importen el mismo módulo hasta que se complete la importación original, lo que impide que otros subprocesos vean objetos de módulo incompletos contruidos por el subproceso original. Se hace una excepción para las importaciones circulares, que por construcción tienen que exponer un objeto de módulo incompleto en algún momento.

Distinto en la versión 3.3: El esquema de bloqueo ha cambiado a bloqueos por módulo en su mayor parte. Se mantiene un bloqueo de importación global para algunas tareas críticas, como la inicialización de los bloqueos por módulo.

Obsoleto desde la versión 3.4.

`imp.acquire_lock()`

Adquiera el bloqueo de importación global del intérprete para el subproceso actual. Este bloqueo debe ser utilizado por los ganchos de importación para garantizar la seguridad de subprocesos al importar módulos.

Una vez que un subproceso ha adquirido el bloqueo de importación, el mismo subproceso puede adquirirlo de nuevo sin bloquear; el subproceso debe liberarlo una vez por cada vez que lo ha adquirido.

En plataformas sin subprocesos, esta función no hace nada.

Distinto en la versión 3.3: El esquema de bloqueo ha cambiado a bloqueos por módulo en su mayor parte. Se mantiene un bloqueo de importación global para algunas tareas críticas, como la inicialización de los bloqueos por módulo.

Obsoleto desde la versión 3.4.

`imp.release_lock()`

Suelte el bloqueo de importación global del intérprete. En plataformas sin subprocesos, esta función no hace nada.

Distinto en la versión 3.3: El esquema de bloqueo ha cambiado a bloqueos por módulo en su mayor parte. Se mantiene un bloqueo de importación global para algunas tareas críticas, como la inicialización de los bloqueos por módulo.

Obsoleto desde la versión 3.4.

Las siguientes constantes con valores enteros, definidas en este módulo, se utilizan para indicar el resultado de búsqueda de `find_module()`.

`imp.PY_SOURCE`

El módulo se encontró como un archivo de origen.

Obsoleto desde la versión 3.3.

`imp.PY_COMPILED`

El módulo se encontró como un archivo de objeto de código compilado.

Obsoleto desde la versión 3.3.

`imp.C_EXTENSION`

El módulo se encontró como biblioteca compartida cargable dinámicamente.

Obsoleto desde la versión 3.3.

`imp.PKG_DIRECTORY`

El módulo se encontró como un directorio de paquetes.

Obsoleto desde la versión 3.3.

`imp.C_BUILTIN`

El módulo fue encontrado como un módulo incorporado.

Obsoleto desde la versión 3.3.

`imp.PY_FROZEN`

El módulo fue encontrado como un módulo congelado.

Obsoleto desde la versión 3.3.

class `imp.NullImporter` (*path_string*)

El tipo `NullImporter` es un enlace de importación [PEP 302](#) que controla cadenas de ruta de acceso que no son de directorio al no encontrar ningún módulo. Llamar a este tipo con un directorio existente o una cadena vacía genera `ImportError`. De lo contrario, se retorna una instancia `NullImporter`.

Las instancias solo tienen un método:

find_module (*fullname* [, *path*])

Este método siempre retorna `None`, lo que indica que no se pudo encontrar el módulo solicitado.

Distinto en la versión 3.3: `None` se inserta en `sys.path_importer_cache` en lugar de una instancia de `NullImporter`.

Obsoleto desde la versión 3.4: Inserte `None` en `sys.path_importer_cache` en su lugar.

36.2.1 Ejemplos

La siguiente función emula lo que era la instrucción de importación estándar hasta Python 1.4 (sin nombres de módulo jerárquico). (Esta *implementación* no funcionaría en esa versión, ya que `find_module()` se ha ampliado y `load_module()` se ha añadido en 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()
```

Módulos no documentados

Listado de módulos que actualmente no están documentados, pero deberían estarlo. Se aceptan contribuciones mediante correo electrónico (docs@python.org).

La idea y contenidos originales de este apartado se obtuvieron de una publicación de Fredrik Lundh; los contenidos específicos de este apartado han sido sustancialmente revisados.

37.1 Módulos específicos por plataforma

Estos módulos se utilizan para implementar el módulo `os.path`, por lo que no están documentados. No hay una gran necesidad de documentarlos.

`ntpath` — Implementación de `os.path` en plataformas Win32 y Win64.

`posixpath` — Implementación de `os.path` en POSIX.

>>> El prompt en el shell interactivo de Python por omisión. Frecuentemente vistos en ejemplos de código que pueden ser ejecutados interactivamente en el intérprete.

. . . Puede referirse a:

- El prompt en el shell interactivo de Python por omisión cuando se ingresa código para un bloque indentado de código, y cuando se encuentra entre dos delimitadores que emparejan (paréntesis, corchetes, llaves o comillas triples), o después de especificar un decorador.
- La constante incorporada *Ellipsis*.

2to3 Una herramienta que intenta convertir código de Python 2.x a Python 3.x arreglando la mayoría de las incompatibilidades que pueden ser detectadas analizando el código y recorriendo el árbol de análisis sintáctico.

2to3 está disponible en la biblioteca estándar como *lib2to3*; un punto de entrada independiente es provisto como `Tools/scripts/2to3`. Vea *2to3 - Traducción de código Python 2 a 3*.

clase base abstracta Las clases base abstractas (ABC, por sus siglas en inglés *Abstract Base Class*) complementan al *duck-typing* brindando un forma de definir interfaces con técnicas como *hasattr()* que serían confusas o sutilmente erróneas (por ejemplo con magic methods). Las ABC introduce subclases virtuales, las cuales son clases que no heredan desde una clase pero aún así son reconocidas por *isinstance()* y *issubclass()*; vea la documentación del módulo *abc*. Python viene con muchas ABC incorporadas para las estructuras de datos (en el módulo *collections.abc*), números (en el módulo *numbers*), flujos de datos (en el módulo *io*), buscadores y cargadores de importaciones (en el módulo *importlib.abc*). Puede crear sus propios ABCs con el módulo *abc*.

anotación Una etiqueta asociada a una variable, atributo de clase, parámetro de función o valor de retorno, usado por convención como un *type hint*.

Las anotaciones de variables no pueden ser accedidas en tiempo de ejecución, pero las anotaciones de variables globales, atributos de clase, y funciones son almacenadas en el atributo especial `__annotations__` de módulos, clases y funciones, respectivamente.

Vea *variable annotation*, *function annotation*, **PEP 484** y **PEP 526**, los cuales describen esta funcionalidad.

argumento Un valor pasado a una *function* (o *method*) cuando se llama a la función. Hay dos clases de argumentos:

- *argumento nombrado*: es un argumento precedido por un identificador (por ejemplo, `nombre=`) en una llamada a una función o pasado como valor en un diccionario precedido por `**`. Por ejemplo 3 y 5 son argumentos nombrados en las llamadas a *complex()*:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *argumento posicional* son aquellos que no son nombrados. Los argumentos posicionales deben aparecer al principio de una lista de argumentos o ser pasados como elementos de un *iterable* precedido por `*`. Por ejemplo, 3 y 5 son argumentos posicionales en las siguientes llamadas:

```
complex(3, 5)
complex(*(3, 5))
```

Los argumentos son asignados a las variables locales en el cuerpo de la función. Vea en la sección *calls* las reglas que rigen estas asignaciones. Sintácticamente, cualquier expresión puede ser usada para representar un argumento; el valor evaluado es asignado a la variable local.

Vea también el *parameter* en el glosario, la pregunta frecuente la diferencia entre argumentos y parámetros, y [PEP 362](#).

administrador asincrónico de contexto Un objeto que controla el entorno visible en una sentencia `async with` al definir los métodos `__aenter__()` y `__aexit__()`. Introducido por [PEP 492](#).

generador asincrónico Una función que retorna un *asynchronous generator iterator*. Es similar a una función corrutina definida con `async def` excepto que contiene expresiones `yield` para producir series de variables usadas en un ciclo `async for`.

Usualmente se refiere a una función generadora asincrónica, pero puede referirse a un *iterador generador asincrónico* en ciertos contextos. En aquellos casos en los que el significado no está claro, usar los términos completos evita la ambigüedad.

Una función generadora asincrónica puede contener expresiones `await` así como sentencias `async for`, y `async with`.

iterador generador asincrónico Un objeto creado por una función *asynchronous generator*.

Este es un *asynchronous iterator* el cual cuando es llamado usa el método `__anext__()` retornando un objeto a la espera (*awaitable*) el cual ejecutará el cuerpo de la función generadora asincrónica hasta la siguiente expresión `yield`.

Cada `yield` suspende temporalmente el procesamiento, recordando el estado local de ejecución (incluyendo a las variables locales y las sentencias `try` pendientes). Cuando el *iterador del generador asincrónico* vuelve efectivamente con otro objeto a la espera (*awaitable*) retornado por el método `__anext__()`, retoma donde lo dejó. Vea [PEP 492](#) y [PEP 525](#).

iterable asincrónico Un objeto, que puede ser usado en una sentencia `async for`. Debe retornar un *asynchronous iterator* de su método `__aiter__()`. Introducido por [PEP 492](#).

iterador asincrónico Un objeto que implementa los métodos `__aiter__()` y `__anext__()`. `__anext__()` debe retornar un objeto *awaitable*. `async for` resuelve los esperables retornados por un método de iterador asincrónico `__anext__()` hasta que lanza una excepción *StopAsyncIteration*. Introducido por [PEP 492](#).

atributo Un valor asociado a un objeto que es referenciado por el nombre usado en expresiones de punto. Por ejemplo, si un objeto *o* tiene un atributo *a* sería referenciado como *o.a*.

a la espera Es un objeto a la espera (*awaitable*) que puede ser usado en una expresión `await`. Puede ser una *coroutine* o un objeto con un método `__await__()`. Vea también [PEP 492](#).

BDFL Sigla de *Benevolent Dictator For Life*, benevolente dictador vitalicio, es decir [Guido van Rossum](#), el creador de Python.

archivo binario Un *file object* capaz de leer y escribir *objetos tipo binarios*. Ejemplos de archivos binarios son los abiertos en modo binario (`'rb'`, `'wb'` o `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, e instancias de *io.BytesIO* y de *gzip.GzipFile*.

Vea también *text file* para un objeto archivo capaz de leer y escribir objetos *str*.

objetos tipo binarios Un objeto que soporta `bufferobjects` y puede exportar un búfer *C-contiguous*. Esto incluye todas los objetos `bytes`, `bytearray`, y `array.array`, así como muchos objetos comunes `memoryview`. Los objetos tipo binarios pueden ser usados para varias operaciones que usan datos binarios; éstas incluyen compresión, salvar a archivos binarios, y enviarlos a través de un socket.

Algunas operaciones necesitan que los datos binarios sean mutables. La documentación frecuentemente se refiere a éstos como «objetos tipo binario de lectura y escritura». Ejemplos de objetos de búfer mutables incluyen a `bytearray` y `memoryview` de la `bytearray`. Otras operaciones que requieren datos binarios almacenados en objetos inmutables («objetos tipo binario de sólo lectura»); ejemplos de éstos incluyen `bytes` y `memoryview` del objeto `bytes`.

bytecode El código fuente Python es compilado en *bytecode*, la representación interna de un programa python en el intérprete CPython. El *bytecode* también es guardado en caché en los archivos `.pyc` de tal forma que ejecutar el mismo archivo es más fácil la segunda vez (la recompilación desde el código fuente a *bytecode* puede ser evitada). Este «lenguaje intermedio» deberá correr en una *virtual machine* que ejecute el código de máquina correspondiente a cada *bytecode*. Note que los *bytecodes* no tienen como requisito trabajar en las diversas máquina virtuales de Python, ni de ser estable entre versiones Python.

Una lista de las instrucciones en *bytecode* está disponible en la documentación de *el módulo dis*.

callback A subroutine function which is passed as an argument to be executed at some point in the future.

class Una plantilla para crear objetos definidos por el usuario. Las definiciones de clase normalmente contienen definiciones de métodos que operan una instancia de la clase.

variable de clase Una variable definida en una clase y prevista para ser modificada sólo a nivel de clase (es decir, no en una instancia de la clase).

coerción La conversión implícita de una instancia de un tipo en otra durante una operación que involucra dos argumentos del mismo tipo. Por ejemplo, `int(3.15)` convierte el número de punto flotante al entero 3, pero en `3 + 4.5`, cada argumento es de un tipo diferente (uno entero, otro flotante), y ambos deben ser convertidos al mismo tipo antes de que puedan ser sumados o emitiría un `TypeError`. Sin coerción, todos los argumentos, incluso de tipos compatibles, deberían ser normalizados al mismo tipo por el programador, por ejemplo `float(3)+4.5` en lugar de `3+4.5`.

número complejo Una extensión del sistema familiar de número reales en el cual los números son expresados como la suma de una parte real y una parte imaginaria. Los números imaginarios son múltiplos de la unidad imaginaria (la raíz cuadrada de -1), usualmente escrita como *i* en matemáticas o *j* en ingeniería. Python tiene soporte incorporado para números complejos, los cuales son escritos con la notación mencionada al final.; la parte imaginaria es escrita con un sufijo *j*, por ejemplo, `3+1j`. Para tener acceso a los equivalentes complejos del módulo `math` module, use `cmath`. El uso de números complejos es matemática bastante avanzada. Si no le parecen necesarios, puede ignorarlos sin inconvenientes.

administrador de contextos Un objeto que controla el entorno en la sentencia `with` definiendo los métodos `__enter__()` y `__exit__()`. Vea **PEP 343**.

variable de contexto Una variable que puede tener diferentes valores dependiendo del contexto. Esto es similar a un almacenamiento de hilo local *Thread-Local Storage* en el cual cada hilo de ejecución puede tener valores diferentes para una variable. Sin embargo, con las variables de contexto, podría haber varios contextos en un hilo de ejecución y el uso principal de las variables de contexto es mantener registro de las variables en tareas concurrentes asíncronas. Vea *contextvars*.

contiguo Un búfer es considerado contiguo con precisión si es *C-contiguo* o *Fortran contiguo*. Los búferes cero dimensionales con C y Fortran contiguos. En los arreglos unidimensionales, los ítems deben ser dispuestos en memoria uno siguiente al otro, ordenados por índices que comienzan en cero. En arreglos unidimensionales C-contiguos, el último índice varía más velozmente en el orden de las direcciones de memoria. Sin embargo, en arreglos Fortran contiguos, el primer índice vería más rápidamente.

corrutina Las corrutinas son una forma más generalizadas de las subrutinas. A las subrutinas se ingresa por un punto y se sale por otro punto. Las corrutinas pueden ser iniciadas, finalizadas y reanudadas en muchos puntos diferentes. Pueden ser implementadas con la sentencia `async def`. Vea además **PEP 492**.

función corrutina Un función que retorna un objeto *coroutine*. Una función corrutina puede ser definida con la sentencia `async def`, y puede contener las palabras claves `await`, `async for`, y `async with`. Las mismas son introducidas en **PEP 492**.

CPython La implementación canónica del lenguaje de programación Python, como se distribuye en python.org. El término «CPython» es usado cuando es necesario distinguir esta implementación de otras como *Jython* o *IronPython*.

decorador Una función que retorna otra función, usualmente aplicada como una función de transformación empleando la sintaxis `@envoltorio`. Ejemplos comunes de decoradores son `classmethod()` y `staticmethod()`.

La sintaxis del decorador es meramente azúcar sintáctico, las definiciones de las siguientes dos funciones son semánticamente equivalentes:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

El mismo concepto existe para clases, pero son menos usadas. Vea la documentación de `function definitions` y `class definitions` para mayor detalle sobre decoradores.

descriptor Cualquier objeto que define los métodos `__get__()`, `__set__()`, o `__delete__()`. Cuando un atributo de clase es un descriptor, su conducta enlazada especial es disparada durante la búsqueda del atributo. Normalmente, usando `a.b` para consultar, establecer o borrar un atributo busca el objeto llamado `b` en el diccionario de clase de `a`, pero si `b` es un descriptor, el respectivo método descriptor es llamado. Entender descriptors es clave para lograr una comprensión profunda de Python porque son la base de muchas de las capacidades incluyendo funciones, métodos, propiedades, métodos de clase, métodos estáticos, y referencia a súper clases.

Para mayor información sobre los métodos de los descriptors vea `descriptors`.

diccionario Un arreglo asociativo, con claves arbitrarias que son asociadas a valores. Las claves pueden ser cualquier objeto con los métodos `__hash__()` y `__eq__()`. Son llamadas `hash` en Perl.

dictionary comprehension A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`. See `comprehensions`.

vista de diccionario Los objetos retornados por los métodos `dict.keys()`, `dict.values()`, y `dict.items()` son llamados vistas de diccionarios. Proveen una vista dinámica de las entradas de un diccionario, lo que significa que cuando el diccionario cambia, la vista refleja éstos cambios. Para forzar a la vista de diccionario a convertirse en una lista completa, use `list(dictview)`. Vea *Objetos tipos vista de diccionario*.

docstring Una cadena de caracteres literal que aparece como la primera expresión en una clase, función o módulo. Aunque es ignorada cuando se ejecuta, es reconocida por el compilador y puesta en el atributo `__doc__` de la clase, función o módulo comprendida. Como está disponible mediante introspección, es el lugar canónico para ubicar la documentación del objeto.

tipado de pato Un estilo de programación que no revisa el tipo del objeto para determinar si tiene la interfaz correcta; en vez de ello, el método o atributo es simplemente llamado o usado («Si se ve como un pato y grazna como un pato, debe ser un pato»). Enfatizando las interfaces en vez de hacerlo con los tipos específicos, un código bien diseñado pues tener mayor flexibilidad permitiendo la sustitución polimórfica. El tipado de pato *duck-typing* evita usar pruebas llamando a `type()` o `isinstance()`. (Nota: si embargo, el tipado de pato puede ser complementado con *abstract base classes*. En su lugar, generalmente pregunta con `hasattr()` o *EAFP*).

EAFP Del inglés *Easier to ask for forgiveness than permission*, es más fácil pedir perdón que pedir permiso. Este estilo de codificación común en Python asume la existencia de claves o atributos válidos y atrapa las excepciones si esta suposición resulta falsa. Este estilo rápido y limpio está caracterizado por muchas sentencias `try` y `except`. Esta técnica contrasta con estilo *LBYL* usual en otros lenguajes como C.

expresión Una construcción sintáctica que puede ser evaluada, hasta dar un valor. En otras palabras, una expresión es una acumulación de elementos de expresión tales como literales, nombres, accesos a atributos, operadores o

llamadas a funciones, todos ellos retornando valor. A diferencia de otros lenguajes, no toda la sintaxis del lenguaje son expresiones. También hay *statements* que no pueden ser usadas como expresiones, como la `while`. Las asignaciones también son sentencias, no expresiones.

módulo de extensión Un módulo escrito en C o C++, usando la API para C de Python para interactuar con el núcleo y el código del usuario.

f-string Son llamadas *f-strings* las cadenas literales que usan el prefijo `'f'` o `'F'`, que es una abreviatura para formatted string literals. Vea también [PEP 498](#).

objeto archivo Un objeto que expone una API orientada a archivos (con métodos como `read()` o `write()`) al objeto subyacente. Dependiendo de la forma en la que fue creado, un objeto archivo, puede mediar el acceso a un archivo real en el disco u otro tipo de dispositivo de almacenamiento o de comunicación (por ejemplo, entrada/salida estándar, búfer de memoria, sockets, pipes, etc.). Los objetos archivo son también denominados *objetos tipo archivo* o *flujos*.

Existen tres categorías de objetos archivo: crudos *raw* [archivos binarios](#), con búfer [archivos binarios](#) y [archivos de texto](#). Sus interfaces son definidas en el módulo `io`. La forma canónica de crear objetos archivo es usando la función `open()`.

objetos tipo archivo Un sinónimo de *file object*.

buscador Un objeto que trata de encontrar el *loader* para el módulo que está siendo importado.

Desde la versión 3.3 de Python, existen dos tipos de buscadores: *meta buscadores de ruta* para usar con `sys.meta_path`, y *buscadores de entradas de rutas* para usar con `sys.path_hooks`.

Vea [PEP 302](#), [PEP 420](#) y [PEP 451](#) para mayores detalles.

división entera Una división matemática que se redondea hacia el entero menor más cercano. El operador de la división entera es `//`. Por ejemplo, la expresión `11 // 4` evalúa 2 a diferencia del 2.75 retornado por la verdadera división de números flotantes. Note que `(-11) // 4` es -3 porque es -2.75 redondeado *para abajo*. Ver [PEP 238](#).

función Una serie de sentencias que retornan un valor al que las llama. También se le puede pasar cero o más *argumentos* los cuales pueden ser usados en la ejecución de la misma. Vea también [parameter](#), [method](#), y la sección *function*.

anotación de función Una *annotation* del parámetro de una función o un valor de retorno.

Las anotaciones de funciones son usadas frecuentemente para *indicadores de tipo*, por ejemplo, se espera que una función tome dos argumentos de clase `int` y también se espera que retorne dos valores `int`:

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

La sintaxis de las anotaciones de funciones son explicadas en la sección *function*.

Vea [variable annotation](#) y [PEP 484](#), que describen esta funcionalidad.

__future__ Un pseudo-módulo que los programadores pueden usar para habilitar nuevas capacidades del lenguaje que no son compatibles con el intérprete actual.

Al importar el módulo `__future__` y evaluar sus variables, puede verse cuándo las nuevas capacidades fueron agregadas por primera vez al lenguaje y cuando se quedaron establecidas por defecto:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

recolección de basura El proceso de liberar la memoria de lo que ya no está en uso. Python realiza recolección de basura (*garbage collection*) llevando la cuenta de las referencias, y el recogedor de basura cíclico es capaz de detectar y romper las referencias cíclicas. El recogedor de basura puede ser controlado mediante el módulo `gc`.

generador Una función que retorna un *generator iterator*. Luce como una función normal excepto que contiene la expresión `yield` para producir series de valores utilizables en un bucle `for` o que pueden ser obtenidas una por una con la función `next()`.

Usualmente se refiere a una función generadora, pero puede referirse a un *iterador generador* en ciertos contextos. En aquellos casos en los que el significado no está claro, usar los términos completos evita la ambigüedad.

iterador generador Un objeto creado por una función *generator*.

Cada `yield` suspende temporalmente el procesamiento, recordando el estado de ejecución local (incluyendo las variables locales y las sentencias `try` pendientes). Cuando el «iterador generado» vuelve, retoma donde ha dejado, a diferencia de lo que ocurre con las funciones que comienzan nuevamente con cada invocación.

expresión generadora Una expresión que retorna un iterador. Luce como una expresión normal seguida por la cláusula `for` definiendo así una variable de bucle, un rango y una cláusula opcional `if`. La expresión combinada genera valores para la función contenedora:

```
>>> sum(i*i for i in range(10))           # sum of squares 0, 1, 4, ... 81
285
```

función genérica Una función compuesta de muchas funciones que implementan la misma operación para diferentes tipos. Qué implementación deberá ser usada durante la llamada a la misma es determinado por el algoritmo de despacho.

Vea también la entrada de glosario *single dispatch*, el decorador `functools singledispatch()`, y **PEP 443**.

GIL Vea *global interpreter lock*.

bloqueo global del intérprete Mecanismo empleado por el intérprete *CPython* para asegurar que sólo un hilo ejecute el *bytecode* Python por vez. Esto simplifica la implementación de CPython haciendo que el modelo de objetos (incluyendo algunos críticos como *dict*) están implícitamente a salvo de acceso concurrente. Bloqueando el intérprete completo se simplifica hacerlo multi-hilos, a costa de mucho del paralelismo ofrecido por las máquinas con múltiples procesadores.

Sin embargo, algunos módulos de extensión, tanto estándar como de terceros, están diseñados para liberar el GIL cuando se realizan tareas computacionalmente intensivas como la compresión o el *hashing*. Además, el GIL siempre es liberado cuando se hace entrada/salida.

Esfuerzos previos hechos para crear un intérprete «sin hilos» (uno que bloquee los datos compartidos con una granularidad mucho más fina) no han sido exitosos debido a que el rendimiento sufrió para el caso más común de un solo procesador. Se cree que superar este problema de rendimiento haría la implementación mucho más compleja y por tanto, más costosa de mantener.

hash-based pyc Un archivo cache de *bytecode* que usa el *hash* en vez de usar el tiempo de la última modificación del archivo fuente correspondiente para determinar su validez. Vea *pyc-invalidation*.

hashable Un objeto es *hashable* si tiene un valor de hash que nunca cambiará durante su tiempo de vida (necesita un método `__hash__()`), y puede ser comparado con otro objeto (necesita el método `__eq__()`). Los objetos hashables que se comparan iguales deben tener el mismo número hash.

Ser *hashable* hace a un objeto utilizable como clave de un diccionario y miembro de un set, porque éstas estructuras de datos usan los valores de hash internamente.

La mayoría de los objetos inmutables incorporados en Python son *hashables*; los contenedores mutables (como las listas o los diccionarios) no lo son; los contenedores inmutables (como tuplas y conjuntos *frozensets*) son *hashables* si sus elementos son *hashables*. Los objetos que son instancias de clases definidas por el usuario son *hashables* por defecto. Todos se comparan como desiguales (excepto consigo mismos), y su valor de hash está derivado de su función `id()`.

IDLE El entorno integrado de desarrollo de Python, o *Integrated Development Environment for Python*. IDLE es un editor básico y un entorno de intérprete que se incluye con la distribución estándar de Python.

inmutable Un objeto con un valor fijo. Los objetos inmutables son números, cadenas y tuplas. Éstos objetos no pueden ser alterados. Un nuevo objeto debe ser creado si un valor diferente ha de ser guardado. Juegan un

rol importante en lugares donde es necesario un valor de hash constante, por ejemplo como claves de un diccionario.

ruta de importación Una lista de las ubicaciones (o *entradas de ruta*) que son revisadas por *path based finder* al importar módulos. Durante la importación, ésta lista de localizaciones usualmente viene de `sys.path`, pero para los subpaquetes también puede incluir al atributo `__path__` del paquete padre.

importar El proceso mediante el cual el código Python dentro de un módulo se hace alcanzable desde otro código Python en otro módulo.

importador Un objeto que buscan y lee un módulo; un objeto que es tanto *finder* como *loader*.

interactivo Python tiene un intérprete interactivo, lo que significa que puede ingresar sentencias y expresiones en el prompt del intérprete, ejecutarlos de inmediato y ver sus resultados. Sólo ejecute `python` sin argumentos (podría seleccionarlo desde el menú principal de su computadora). Es una forma muy potente de probar nuevas ideas o inspeccionar módulos y paquetes (recuerde `help(x)`).

interpretado Python es un lenguaje interpretado, a diferencia de uno compilado, a pesar de que la distinción puede ser difusa debido al compilador a *bytecode*. Esto significa que los archivos fuente pueden ser corridos directamente, sin crear explícitamente un ejecutable que es corrido luego. Los lenguajes interpretados típicamente tienen ciclos de desarrollo y depuración más cortos que los compilados, sin embargo sus programas suelen correr más lentamente. Vea también *interactive*.

apagado del intérprete Cuando se le solicita apagarse, el intérprete Python ingresa a un fase especial en la cual gradualmente libera todos los recursos reservados, como módulos y varias estructuras internas críticas. También hace varias llamadas al *recolector de basura*. Esto puede disparar la ejecución de código de destructores definidos por el usuario o *weakref callbacks*. El código ejecutado durante la fase de apagado puede encontrar varias excepciones debido a que los recursos que necesita pueden no funcionar más (ejemplos comunes son los módulos de bibliotecas o los artefactos de advertencias *warnings machinery*).

La principal razón para el apagado del intérprete es que el módulo `__main__` o el script que estaba corriendo termine su ejecución.

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*, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *Sequence* semantics.

Los iterables pueden ser usados en el bucle `for` y en muchos otros sitios donde una secuencia es necesaria (`zip()`, `map()`, ...). Cuando un objeto iterable es pasado como argumento a la función incorporada `iter()`, retorna un iterador para el objeto. Este iterador pasa así el conjunto de valores. Cuando se usan iterables, normalmente no es necesario llamar a la función `iter()` o tratar con los objetos iteradores usted mismo. La sentencia `for` lo hace automáticamente por usted, creando un variable temporal sin nombre para mantener el iterador mientras dura el bucle. Vea también *iterator*, *sequence*, y *generator*.

iterador Un objeto que representa un flujo de datos. Llamadas repetidas al método `__next__()` del iterador (o al pasar la función incorporada `next()`) retorna ítems sucesivos del flujo. Cuando no hay más datos disponibles, una excepción *StopIteration* es disparada. En este momento, el objeto iterador está exhausto y cualquier llamada posterior al método `__next__()` sólo dispara otra vez *StopIteration*. Los iteradores necesitan tener un método `__iter__()` que retorna el objeto iterador mismo así cada iterador es también un iterable y puede ser usado en casi todos los lugares donde los iterables son aceptados. Una excepción importante es el código que intenta múltiples pases de iteración. Un objeto contenedor (como la *list*) produce un nuevo iterador cada vez que pasa a una función `iter()` o se usa en un bucle `for`. Intentar ésto con un iterador simplemente retornaría el mismo objeto iterador exhausto usado en previas iteraciones, haciéndolo aparecer como un contenedor vacío.

Puede encontrar más información en *Tipos de iteradores*.

función clave Una función clave o una función de colación es un invocable que retorna un valor usado para el ordenamiento o clasificación. Por ejemplo, `locale.strxfrm()` es usada para producir claves de ordenamiento que se adaptan a las convenciones específicas de ordenamiento de un *locale*.

Cierta cantidad de herramientas de Python aceptan funciones clave para controlar como los elementos son ordenados o agrupados. Incluyendo a `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, y `itertools.groupby()`.

Hay varias formas de crear una función clave. Por ejemplo, el método `str.lower()` puede servir como función clave para ordenamientos que no distingan mayúsculas de minúsculas. Como alternativa, una función clave puede ser realizada con una expresión lambda como `lambda r: (r[0], r[2])`. También, el módulo `operator` provee tres constructores de funciones clave: `attrgetter()`, `itemgetter()`, y `methodcaller()`. Vea en [Sorting HOW TO](#) ejemplos de cómo crear y usar funciones clave.

argumento nombrado Vea [argument](#).

lambda Una función anónima de una línea consistente en un sola [expression](#) que es evaluada cuando la función es llamada. La sintaxis para crear una función lambda es `lambda [parameters]: expression`

LBYL Del inglés *Look before you leap*, «mira antes de saltar». Es un estilo de codificación que prueba explícitamente las condiciones previas antes de hacer llamadas o búsquedas. Este estilo contrasta con la manera [EAFP](#) y está caracterizado por la presencia de muchas sentencias `if`.

En entornos multi-hilos, el método LBYL tiene el riesgo de introducir condiciones de carrera entre los hilos que están «mirando» y los que están «saltando». Por ejemplo, el código, `if key in mapping: return mapping[key]` puede fallar si otro hilo remueve `key` de `mapping` después del test, pero antes de retornar el valor. Este problema puede ser resuelto usando bloqueos o empleando el método EAFP.

lista Es una [sequence](#) Python incorporada. A pesar de su nombre es más similar a un arreglo en otros lenguajes que a una lista enlazada porque el acceso a los elementos es $O(1)$.

comprensión de listas Una forma compacta de procesar todos o parte de los elementos en una secuencia y retornar una lista como resultado. `result = ['{:04x}'.format(x) for x in range(256) if x % 2 == 0]` genera una lista de cadenas conteniendo números hexadecimales (0x..) entre 0 y 255. La cláusula `if` es opcional. Si es omitida, todos los elementos en `range(256)` son procesados.

cargador Un objeto que carga un módulo. Debe definir el método llamado `load_module()`. Un cargador es normalmente retornados por un [finder](#). Vea [PEP 302](#) para detalles y `importlib.abc.Loader` para una [abstract base class](#).

método mágico Una manera informal de llamar a un [special method](#).

mapeado Un objeto contenedor que permite recupero de claves arbitrarias y que implementa los métodos especificados en la [Mapping](#) o [MutableMapping abstract base classes](#). Por ejemplo, `dict`, `collections.defaultdict`, `collections.OrderedDict` y `collections.Counter`.

meta buscadores de ruta Un [finder](#) retornado por una búsqueda de `sys.meta_path`. Los meta buscadores de ruta están relacionados a [buscadores de entradas de rutas](#), pero son algo diferente.

Vea en `importlib.abc.MetaPathFinder` los métodos que los meta buscadores de ruta implementan.

metaclass La clase de una clase. Las definiciones de clases crean nombres de clase, un diccionario de clase, y una lista de clases base. Las metaclasses son responsables de tomar estos tres argumentos y crear la clase. La mayoría de los objetos de un lenguaje de programación orientado a objetos provienen de una implementación por defecto. Lo que hace a Python especial que es posible crear metaclasses a medida. La mayoría de los usuario nunca necesitarán esta herramienta, pero cuando la necesidad surge, las metaclasses pueden brindar soluciones poderosas y elegantes. Han sido usadas para *loggear* acceso de atributos, agregar seguridad a hilos, rastrear la creación de objetos, implementar *singletons*, y muchas otras tareas.

Más información hallará en metaclasses.

método Una función que es definida dentro del cuerpo de una clase. Si es llamada como un atributo de una instancia de otra clase, el método tomará el objeto instanciado como su primer [argument](#) (el cual es usualmente denominado *self*). Vea [function](#) y [nested scope](#).

orden de resolución de métodos Orden de resolución de métodos es el orden en el cual una clase base es buscada por un miembro durante la búsqueda. Mire en [The Python 2.3 Method Resolution Order](#) los detalles del algoritmo usado por el intérprete Python desde la versión 2.3.

módulo Un objeto que sirve como unidad de organización del código Python. Los módulos tienen espacios de nombres conteniendo objetos Python arbitrarios. Los módulos son cargados en Python por el proceso de [importing](#).

Vea también [package](#).

especificador de módulo Un espacio de nombres que contiene la información relacionada a la importación usada al leer un módulo. Una instancia de `importlib.machinery.ModuleSpec`.

MRO Vea [method resolution order](#).

mutable Los objetos mutables pueden cambiar su valor pero mantener su `id()`. Vea también [immutable](#).

tupla nombrada La denominación «tupla nombrada» se aplica a cualquier tipo o clase que hereda de una tupla y cuyos elementos indexables son también accesibles usando atributos nombrados. Este tipo o clase puede tener además otras capacidades.

Varios tipos incorporados son tuplas nombradas, incluyendo los valores retornados por `time.localtime()` y `os.stat()`. Otro ejemplo es `sys.float_info`:

```
>>> sys.float_info[1]           # indexed access
1024
>>> sys.float_info.max_exp      # named field access
1024
>>> isinstance(sys.float_info, tuple) # kind of tuple
True
```

Algunas tuplas nombradas con tipos incorporados (como en los ejemplo precedentes). También puede ser creada con una definición regular de clase que hereda de la clase `tuple` y que define campos nombrados. Una clase como esta puede ser hecha personalmente o puede ser creada con la función factoría `collections.namedtuple()`. Esta última técnica automáticamente brinda métodos adicionales que pueden no estar presentes en las tuplas nombradas personalizadas o incorporadas.

espacio de nombres El lugar donde la variable es almacenada. Los espacios de nombres son implementados como diccionarios. Hay espacio de nombre local, global, e incorporado así como espacios de nombres anidados en objetos (en métodos). Los espacios de nombres soportan modularidad previniendo conflictos de nombramiento. Por ejemplo, las funciones `builtins.open` y `os.open()` se distinguen por su espacio de nombres. Los espacios de nombres también ayuda a la legibilidad y mantenibilidad dejando claro qué módulo implementa una función. Por ejemplo, escribiendo `random.seed()` o `itertools.islice()` queda claro que éstas funciones están implementadas en los módulos `random` y `itertools`, respectivamente.

paquete de espacios de nombres Un [PEP 420 package](#) que sirve sólo para contener subpaquetes. Los paquetes de espacios de nombres pueden no tener representación física, y específicamente se diferencian de los [regular package](#) porque no tienen un archivo `__init__.py`.

Vea también [module](#).

alcances anidados La habilidad de referirse a una variable dentro de una definición encerrada. Por ejemplo, una función definida dentro de otra función puede referir a variables en la función externa. Note que los alcances anidados por defecto sólo funcionan para referencia y no para asignación. Las variables locales leen y escriben sólo en el alcance más interno. De manera semejante, las variables globales pueden leer y escribir en el espacio de nombres global. Con `nonlocal` se puede escribir en alcances exteriores.

clase de nuevo estilo Vieja denominación usada para el estilo de clases ahora empleado en todos los objetos de clase. En versiones más tempranas de Python, sólo las nuevas clases podían usar capacidades nuevas y versátiles de Python como `__slots__`, descriptores, propiedades, `__getattr__()`, métodos de clase y métodos estáticos.

objeto Cualquier dato con estado (atributo o valor) y comportamiento definido (métodos). También es la más básica clase base para cualquier [new-style class](#).

paquete Un [module](#) Python que puede contener submódulos o recursivamente, subpaquetes. Técnicamente, un paquete es un módulo Python con un atributo `__path__`.

Vea también [regular package](#) y [namespace package](#).

parámetro Una entidad nombrada en una definición de una [function](#) (o método) que especifica un [argument](#) (o en algunos casos, varios argumentos) que la función puede aceptar. Existen cinco tipos de argumentos:

- *posicional o nombrado*: especifica un argumento que puede ser pasado tanto como [posicional](#) o como [nombrado](#). Este es el tipo por defecto de parámetro, como `foo` y `bar` en el siguiente ejemplo:


```
def func(foo, bar=None): ...
```

- *sólo posicional*: especifica un argumento que puede ser pasado sólo por posición. Los parámetros sólo posicionales pueden ser definidos incluyendo un carácter / en la lista de parámetros de la función después de ellos, como *posonly1* y *posonly2* en el ejemplo que sigue:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *sólo nombrado*: especifica un argumento que sólo puede ser pasado por nombre. Los parámetros sólo por nombre pueden ser definidos incluyendo un parámetro posicional de una sola variable o un simple * antes de ellos en la lista de parámetros en la definición de la función, como *kw_only1* y *kw_only2* en el ejemplo siguiente:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *variable posicional*: especifica una secuencia arbitraria de argumentos posicionales que pueden ser brindados (además de cualquier argumento posicional aceptado por otros parámetros). Este parámetro puede ser definido anteponiendo al nombre del parámetro *, como a *args* en el siguiente ejemplo:

```
def func(*args, **kwargs): ...
```

- *variable nombrado*: especifica que arbitrariamente muchos argumentos nombrados pueden ser brindados (además de cualquier argumento nombrado ya aceptado por cualquier otro parámetro). Este parámetro puede ser definido anteponiendo al nombre del parámetro con **, como *kwargs* en el ejemplo precedente.

Los parámetros puede especificar tanto argumentos opcionales como requeridos, así como valores por defecto para algunos argumentos opcionales.

Vea también el glosario de *argument*, la pregunta respondida en la diferencia entre argumentos y parámetros, la clase *inspect.Parameter*, la sección *function*, y [PEP 362](#).

entrada de ruta Una ubicación única en el *import path* que el *path based finder* consulta para encontrar los módulos a importar.

buscador de entradas de ruta Un *finder* retornado por un invocable en *sys.path_hooks* (esto es, un *path entry hook*) que sabe cómo localizar módulos dada una *path entry*.

Vea en *importlib.abc.PathEntryFinder* los métodos que los buscadores de entradas de ruta implementan.

gancho a entrada de ruta Un invocable en la lista *sys.path_hook* que retorna un *path entry finder* si éste sabe cómo encontrar módulos en un *path entry* específico.

buscador basado en ruta Uno de los *meta buscadores de ruta* por defecto que busca un *import path* para los módulos.

objeto tipo ruta Un objeto que representa una ruta del sistema de archivos. Un objeto tipo ruta puede ser tanto una *str* como un *bytes* representando una ruta, o un objeto que implementa el protocolo *os.PathLike*. Un objeto que soporta el protocolo *os.PathLike* puede ser convertido a ruta del sistema de archivo de clase *str* o *bytes* usando la función *os.fspath()*; *os.fsdecode()* *os.fsencode()* pueden emplearse para garantizar que retorne respectivamente *str* o *bytes*. Introducido por [PEP 519](#).

PEP Propuesta de mejora de Python, del inglés *Python Enhancement Proposal*. Un PEP es un documento de diseño que brinda información a la comunidad Python, o describe una nueva capacidad para Python, sus procesos o entorno. Los PEPs deberían dar una especificación técnica concisa y una fundamentación para las capacidades propuestas.

Los PEPs tienen como propósito ser los mecanismos primarios para proponer nuevas y mayores capacidad, para recoger la opinión de la comunidad sobre un tema, y para documentar las decisiones de diseño que se han hecho en Python. El autor del PEP es el responsable de lograr consenso con la comunidad y documentar las opiniones disidentes.

Vea [PEP 1](#).

porción Un conjunto de archivos en un único directorio (posiblemente guardado en un archivo comprimido *zip*) que contribuye a un espacio de nombres de paquete, como está definido en [PEP 420](#).

argumento posicional Vea [argument](#).

API provisoria Una API provisoria es aquella que deliberadamente fue excluida de las garantías de compatibilidad hacia atrás de la biblioteca estándar. Aunque no se esperan cambios fundamentales en dichas interfaces, como están marcadas como provisionales, los cambios incompatibles hacia atrás (incluso remover la misma interfaz) podrían ocurrir si los desarrolladores principales lo estiman. Estos cambios no se hacen gratuitamente – solo ocurrirán si fallas fundamentales y serias son descubiertas que no fueron vistas antes de la inclusión de la API.

Incluso para APIs provisionales, los cambios incompatibles hacia atrás son vistos como una «solución de último recurso» - se intentará todo para encontrar una solución compatible hacia atrás para los problemas identificados.

Este proceso permite que la biblioteca estándar continúe evolucionando con el tiempo, sin bloquearse por errores de diseño problemáticos por períodos extensos de tiempo. Vea [PEP 411](#) para más detalles.

paquete provisorio Vea [provisional API](#).

Python 3000 Apodo para la fecha de lanzamiento de Python 3.x (acuñada en un tiempo cuando llegar a la versión 3 era algo distante en el futuro.) También se lo abrevió como *Py3k*.

Pythónico Una idea o pieza de código que sigue ajustadamente la convenciones idiomáticas comunes del lenguaje Python, en vez de implementar código usando conceptos comunes a otros lenguajes. Por ejemplo, una convención común en Python es hacer bucles sobre todos los elementos de un iterable con la sentencia `for`. Muchos otros lenguajes no tienen este tipo de construcción, así que los que no están familiarizados con Python podrían usar contadores numéricos:

```
for i in range(len(food)) :
    print (food[i])
```

En contraste, un método Pythónico más limpio:

```
for piece in food:
    print (piece)
```

nombre calificado Un nombre con puntos mostrando la ruta desde el alcance global del módulo a la clase, función o método definido en dicho módulo, como se define en [PEP 3155](#). Para las funciones o clases de más alto nivel, el nombre calificado es el igual al nombre del objeto:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Cuando es usado para referirse a los módulos, *nombre completamente calificado* significa la ruta con puntos completo al módulo, incluyendo cualquier paquete padre, por ejemplo, *email.mime.text*:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

contador de referencias El número de referencias a un objeto. Cuando el contador de referencias de un objeto cae hasta cero, éste es desalojable. En conteo de referencias no suele ser visible en el código de Python, pero es un elemento clave para la implementación de *CPython*. El módulo *sys* define la *getrefcount()* que los programadores pueden emplear para retornar el conteo de referencias de un objeto en particular.

paquete regular Un *package* tradicional, como aquellos con un directorio conteniendo el archivo `__init__.py`.

Vea también *namespace package*.

__slots__ Es una declaración dentro de una clase que ahorra memoria predeclamando espacio para las atributos de la instancia y eliminando diccionarios de la instancia. Aunque es popular, esta técnica es algo dificultosa de lograr correctamente y es mejor reservarla para los casos raros en los que existen grandes cantidades de instancias en aplicaciones con uso crítico de memoria.

secuencia Un *iterable* que logra un acceso eficiente a los elementos usando índices enteros a través del método especial `__getitem__()` y que define un método `__len__()` que retorna la longitud de la secuencia. Algunas de las secuencias incorporadas son *list*, *str*, *tuple*, y *bytes*. Observe que *dict* también soporta `__getitem__()` y `__len__()`, pero es considerada un mapeo más que una secuencia porque las búsquedas son por claves arbitraria *immutable* y no por enteros.

The *collections.abc.Sequence* abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using *register()*.

set comprehension A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See comprehensions.

despacho único Una forma de despacho de una *generic function* donde la implementación es elegida a partir del tipo de un sólo argumento.

rebanada Un objeto que contiene una porción de una *sequence*. Una rebanada es creada usando la notación de suscripto, `[]` con dos puntos entre los números cuando se ponen varios, como en `nombre_variable[1:3:5]`. La notación con corchete (suscripto) usa internamente objetos *slice*.

método especial Un método que es llamado implícitamente por Python cuando ejecuta ciertas operaciones en un tipo, como la adición. Estos métodos tienen nombres que comienzan y terminan con doble barra baja. Los métodos especiales están documentados en *specialnames*.

sentencia Una sentencia es parte de un conjunto (un «bloque» de código). Una sentencia tanto es una *expression* como alguna de las varias sintaxis usando una palabra clave, como *if*, *while* o *for*.

codificación de texto Un códec que codifica las cadenas Unicode a bytes.

archivo de texto Un *file object* capaz de leer y escribir objetos *str*. Frecuentemente, un archivo de texto también accede a un flujo de datos binario y maneja automáticamente el *text encoding*. Ejemplos de archivos de texto que son abiertos en modo texto (`'r'` o `'w'`), *sys.stdin*, *sys.stdout*, y las instancias de *io.StringIO*.

Vea también *binary file* por objeto de archivos capaces de leer y escribir *objeto tipo binario*.

cadena con triple comilla Una cadena que está enmarcada por tres instancias de comillas («») o apostrofes ("). Aunque no brindan ninguna funcionalidad que no está disponible usando cadenas con comillas simple, son útiles por varias razones. Permiten incluir comillas simples o dobles sin escapar dentro de las cadenas y pueden abarcar múltiples líneas sin el uso de caracteres de continuación, haciéndolas particularmente útiles para escribir docstrings.

tipo El tipo de un objeto Python determina qué tipo de objeto es; cada objeto tiene un tipo. El tipo de un objeto puede ser accedido por su atributo `__class__` o puede ser conseguido usando `type(obj)`.

alias de tipos Un sinónimo para un tipo, creado al asignar un tipo a un identificador.

Los alias de tipos son útiles para simplificar los *indicadores de tipo*. Por ejemplo:

```
from typing import List, Tuple

def remove_gray_shades(
    colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

podría ser más legible así:


```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

Vea *typing* y **PEP 484**, que describen esta funcionalidad.

indicador de tipo Una *annotation* que especifica el tipo esperado para una variable, un atributo de clase, un parámetro para una función o un valor de retorno.

Los indicadores de tipo son opcionales y no son obligados por Python pero son útiles para las herramientas de análisis de tipos estático, y ayuda a las IDE en el completado del código y la refactorización.

Los indicadores de tipo de las variables globales, atributos de clase, y funciones, no de variables locales, pueden ser accedidos usando *typing.get_type_hints()*.

Vea *typing* y **PEP 484**, que describen esta funcionalidad.

saltos de líneas universales Una manera de interpretar flujos de texto en la cual son reconocidos como finales de línea todas siguientes formas: la convención de Unix para fin de línea `'\n'`, la convención de Windows `'\r\n'`, y la vieja convención de Macintosh `'\r'`. Vea **PEP 278** y **PEP 3116**, además de *bytes.splitlines()* para usos adicionales.

anotación de variable Una *annotation* de una variable o un atributo de clase.

Cuando se anota una variable o un atributo de clase, la asignación es opcional:

```
class C:
    field: 'annotation'
```

Las anotaciones de variables son frecuentemente usadas para *type hints*: por ejemplo, se espera que esta variable tenga valores de clase *int*:

```
count: int = 0
```

La sintaxis de la anotación de variables está explicada en la sección *annassign*.

Vea *function annotation*, **PEP 484** y **PEP 526**, los cuales describen esta funcionalidad.

entorno virtual Un entorno cooperativamente aislado de ejecución que permite a los usuarios de Python y a las aplicaciones instalar y actualizar paquetes de distribución de Python sin interferir con el comportamiento de otras aplicaciones de Python en el mismo sistema.

Vea también *venv*.

máquina virtual Una computadora definida enteramente por software. La máquina virtual de Python ejecuta el *bytecode* generado por el compilador de *bytecode*.

Zen de Python Un listado de los principios de diseño y la filosofía de Python que son útiles para entender y usar el lenguaje. El listado puede encontrarse ingresando `«import this»` en la consola interactiva.

Acerca de estos documentos

Estos documentos son generados por [reStructuredText](#) desarrollado por [Sphinx](#), un procesador de documentos específicamente escrito para la documentación de Python.

El desarrollo de la documentación y su cadena de herramientas es un esfuerzo enteramente voluntario, al igual que Python. Si tu quieres contribuir, por favor revisa la página [reporting-bugs](#) para más información de cómo hacerlo. Los nuevos voluntarios son siempre bienvenidos!

Agradecemos a:

- Fred L. Drake, Jr., el creador original de la documentación del conjunto de herramientas de Python y escritor de gran parte del contenido;
- el proyecto [Docutils](#) para creación de reStructuredText y el juego de Utilidades de Documentación;
- Fredrik Lundh por su proyecto [Referencia Alternativa de Python](#) para la cual Sphinx tuvo muchas ideas.

B.1 Contribuidores de la documentación de Python

Muchas personas han contribuido para el lenguaje de Python, la librería estándar de Python, y la documentación de Python. Revisa [Misc/ACKS](#) la distribución de Python para una lista parcial de contribuidores.

Es solamente con la aportación y contribuciones de la comunidad de Python que Python tiene tan fantástica documentación – Muchas gracias!

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 Python-Labs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see <https://www.zope.org/>). 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

Nota: 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

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenses and Acknowledgements for Incorporated Software* for an incomplete list of these licenses.

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.8.20

1. This LICENSE AGREEMENT is between the Python Software Foundation,
→ ("PSF"), and
the Individual or Organization ("Licensee") accessing and otherwise
→ using Python
3.8.20 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 3.8.20 alone or in any derivative
version, provided, however, that PSF's License Agreement and PSF's
→ notice of
copyright, i.e., "Copyright © 2001–2023 Python Software Foundation; All
→ Rights
Reserved" are retained in Python 3.8.20 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 3.8.20 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
3.8.20.
4. PSF is making Python 3.8.20 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 3.8.20 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.8.20
FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A
→ RESULT OF

MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.8.20, 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 3.8.20, 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

(continué en la próxima página)

(proviene de la página anterior)

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 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

(continué en la próxima página)

(proviene de la página anterior)

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 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.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.8.20 DOCUMENTATION

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, 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,
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
```

(continué en la próxima página)

(proviene de la página anterior)

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 Asynchronous socket services

The *asyncchat* 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.3.4 Cookie management

The *http.cookies* 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

(continué en la próxima página)

(proviene de la página anterior)

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.3.5 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.3.6 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

(continué en la próxima página)

(proviene de la página anterior)

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.3.7 XML Remote Procedure Calls

The `xmlrpc.client` 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 MERCHANTABILITY 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 ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.8 test_epoll

The `test_epoll` module 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

(continué en la próxima página)

(proviene de la página anterior)

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.3.9 Select queue

The `select` module 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 OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski's implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

<MIT License>

Copyright (c) 2013 Marek Majkowski <marek@popcount.org>

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.

</MIT License>

(continué en la próxima página)

(proviene de la página anterior)

```
Original location:
    https://github.com/majek/csiphash/

Solution inspired by code from:
    Samuel Neves (supercop/crypto_auth/siphhash24/little)
    djb (supercop/crypto_auth/siphhash24/little2)
    Jean-Philippe Aumasson (https://131002.net/siphhash/siphhash24.c)
```

C.3.11 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.3.12 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
 * are met:
```

(continué en la próxima página)

(proviene de la página anterior)

```

*
* 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.
*
* 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,

```

(continué en la próxima página)

(proviene de la página anterior)

```

* 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 routines 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.3.13 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,

```

(continúe en la próxima página)

(proviene de la página anterior)

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.3.14 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.3.15 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) 1995-2011 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:

(continué en la próxima página)

(proviene de la página anterior)

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
jloup@gzip.org

Mark Adler
madler@alumni.caltech.edu

C.3.16 cfuhash

The implementation of the hash table used by the `tracemalloc` is based on the cfuhash project:

Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * 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.
- * Neither the name of the author 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 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, 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.3.17 libmpdec

The `_decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2016 Stefan Krah. 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  
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF  
SUCH DAMAGE.
```

C.3.18 W3C C14N test suite

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at <https://www.w3.org/TR/xml-c14n2-testcases/> and is distributed under the 3-clause BSD license:

Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang), All Rights Reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of works must retain the original copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work 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, 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.

APÉNDICE D

Copyright

Python and this documentation is:

Copyright © 2001-2023 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.

Bibliografía

- [Frie09] Friedl, Jeffrey. *Mastering Regular Expressions*. 3a ed., O'Reilly Media, 2009. La tercera edición del libro ya no abarca a Python en absoluto, pero la primera edición cubría la escritura de buenos patrones de expresiones regulares con gran detalle.
- [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>.*

—
__future__, 1753
__main__, 1714
_dummy_thread, 864
_thread, 862

a

abc, 1741
aifc, 1350
argparse, 632
array, 243
ast, 1817
asynchat, 1021
asyncio, 865
asyncore, 1017
atexit, 1746
audioop, 1347

b

base64, 1127
bdb, 1633
binascii, 1130
binhex, 1130
bisect, 241
builtins, 1714
bz2, 476

c

calendar, 212
cgi, 1202
cgitb, 1208
chunk, 1358
cmath, 295
cmd, 1418
code, 1777
codecs, 158
codeop, 1779
collections, 217
collections.abc, 233
colorsys, 1359
compileall, 1838
concurrent.futures, 830
configparser, 518

contextlib, 1729
contextvars, 858
copy, 258
copyreg, 437
cProfile, 1649
crypt (*Unix*), 1884
csv, 511
ctypes, 736
curses (*Unix*), 705
curses.ascii, 724
curses.panel, 726
curses.textpad, 722

d

dataclasses, 1721
datetime, 177
dbm, 442
dbm.dumb, 445
dbm.gnu (*Unix*), 443
dbm.ndbm (*Unix*), 445
decimal, 298
difflib, 129
dis, 1841
distutils, 1673
doctest, 1497
dummy_threading, 864

e

email, 1037
email.charset, 1089
email.contentmanager, 1067
email.encoders, 1091
email.errors, 1060
email.generator, 1050
email.header, 1086
email.headerregistry, 1061
email.iterators, 1095
email.message, 1038
email.mime, 1084
email.parser, 1046
email.policy, 1053
email.utils, 1092
encodings.idna, 174
encodings.mbc, 175

`encodings.utf_8_sig`, 175
`ensurepip`, 1674
`enum`, 266
`errno`, 730

f

`faulthandler`, 1638
`fcntl` (*Unix*), 1889
`filecmp`, 401
`fileinput`, 394
`fnmatch`, 409
`formatter`, 1857
`fractions`, 324
`ftplib`, 1257
`functools`, 357

g

`gc`, 1754
`getopt`, 663
`getpass`, 704
`gettext`, 1367
`glob`, 407
`grp` (*Unix*), 1884
`gzip`, 473

h

`hashlib`, 543
`heapq`, 237
`hmac`, 553
`html`, 1135
`html.entities`, 1140
`html.parser`, 1136
`http`, 1248
`http.client`, 1250
`http.cookiejar`, 1311
`http.cookies`, 1308
`http.server`, 1302

i

`imaplib`, 1265
`imghdr`, 1360
`imp`, 1928
`importlib`, 1789
`importlib.abc`, 1792
`importlib.machinery`, 1799
`importlib.metadata`, 1808
`importlib.resources`, 1798
`importlib.util`, 1803
`inspect`, 1758
`io`, 610
`ipaddress`, 1333
`itertools`, 343

j

`json`, 1096
`json.tool`, 1104

k

`keyword`, 1830

l

`lib2to3`, 1612
`linecache`, 410
`locale`, 1376
`logging`, 665
`logging.config`, 681
`logging.handlers`, 692
`lzma`, 479

m

`mailbox`, 1106
`mailcap`, 1105
`marshal`, 441
`math`, 288
`mimetypes`, 1124
`mmap`, 1031
`modulefinder`, 1786
`msilib` (*Windows*), 1861
`msvcrt` (*Windows*), 1867
`multiprocessing`, 782
`multiprocessing.connection`, 812
`multiprocessing.dummy`, 816
`multiprocessing.managers`, 803
`multiprocessing.pool`, 809
`multiprocessing.shared_memory`, 825
`multiprocessing.sharedctypes`, 800

n

`netrc`, 535
`nis` (*Unix*), 1897
`nntplib`, 1271
`numbers`, 285

o

`operator`, 365
`optparse`, 1901
`os`, 559
`os.path`, 389
`ossaudiodev` (*Linux, FreeBSD*), 1361

p

`parser`, 1813
`pathlib`, 373
`pdb`, 1640
`pickle`, 421
`pickletools`, 1855
`pipes` (*Unix*), 1892
`pkgutil`, 1783
`platform`, 727
`plistlib`, 539
`poplib`, 1262
`posix` (*Unix*), 1881
`pprint`, 259
`profile`, 1649
`pstats`, 1650

pty (*Linux*), 1888
pwd (*Unix*), 1882
py_compile, 1836
pyclbr, 1835
pydoc, 1496

q

queue, 855
quopri, 1133

r

random, 326
re, 110
readline (*Unix*), 147
reprlib, 264
resource (*Unix*), 1893
rlcompleter, 151
runpy, 1787

s

sched, 854
secrets, 555
select, 1007
selectors, 1014
shelve, 438
shlex, 1423
shutil, 410
signal, 1024
site, 1773
smtpd, 1285
smtpplib, 1278
sndhdr, 1361
socket, 950
socketserver, 1294
spwd (*Unix*), 1883
sqlite3, 446
ssl, 973
stat, 396
statistics, 332
string, 99
stringprep, 145
struct, 153
subprocess, 836
sunau, 1353
symbol, 1826
symtable, 1824
sys, 1691
sysconfig, 1710
syslog (*Unix*), 1897

t

tabnanny, 1834
tarfile, 495
telnetlib, 1288
tempfile, 403
termios (*Unix*), 1886
test, 1612
test.support, 1614

test.support.script_helper, 1627
textwrap, 140
threading, 769
time, 622
timeit, 1654
tkinter, 1429
tkinter.scrolledtext (*Tk*), 1463
tkinter.tix, 1459
tkinter.ttk, 1440
token, 1826
tokenize, 1830
trace, 1659
traceback, 1747
tracemalloc, 1662
tty (*Unix*), 1888
turtle, 1385
turtledemo, 1416
types, 253
typing, 1477

u

unicodedata, 144
unittest, 1519
unittest.mock, 1548
urllib, 1219
urllib.error, 1246
urllib.parse, 1238
urllib.request, 1220
urllib.response, 1238
urllib.robotparser, 1247
uu, 1133
uuid, 1291

v

venv, 1675

w

warnings, 1715
wave, 1355
weakref, 246
webbrowser, 1199
winreg (*Windows*), 1868
winsound (*Windows*), 1877
wsgiref, 1209
wsgiref.handlers, 1215
wsgiref.headers, 1211
wsgiref.simple_server, 1212
wsgiref.util, 1210
wsgiref.validate, 1214

x

xdrlib, 536
xml, 1140
xml.dom, 1161
xml.dom.minidom, 1171
xml.dom.pulldom, 1175
xml.etree.ElementTree, 1142
xml.parsers.expat, 1189

`xml.parsers.expat.errors`, 1196
`xml.parsers.expat.model`, 1195
`xml.sax`, 1177
`xml.sax.handler`, 1179
`xml.sax.saxutils`, 1184
`xml.sax.xmlreader`, 1185
`xmlrpc.client`, 1320
`xmlrpc.server`, 1327

Z

`zipapp`, 1684
`zipfile`, 485
`zipimport`, 1781
`zlib`, 469

No alfabético

- ..
 - in pathnames, 608
- ..., 1935
 - ellipsis literal, 27, 85
 - in doctests, 1504
 - interpreter prompt, 1501, 1704
 - placeholder, 143, 259, 265
- ??
 - in regular expressions, 111
- . (*dot*)
 - in glob-style wildcards, 407
 - in pathnames, 608
 - in printf-style formatting, 52, 66
 - in regular expressions, 111
 - in string formatting, 101
 - in Tkinter, 1432
- ! (*exclamation*)
 - in a command interpreter, 1419
 - in curses module, 725
 - in glob-style wildcards, 407, 409
 - in string formatting, 101
 - in struct format strings, 154
- (*minus*)
 - binary operator, 31
 - in doctests, 1506
 - in glob-style wildcards, 407, 409
 - in printf-style formatting, 53, 67
 - in regular expressions, 112
 - in string formatting, 103
 - unary operator, 31
- ! (*pdb command*), 1645
- ? (*question mark*)
 - in a command interpreter, 1419
 - in argparse module, 644
 - in AST grammar, 1818
 - in glob-style wildcards, 407, 409
 - in regular expressions, 111
 - in SQL statements, 456
 - in struct format strings, 156
 - replacement character, 162
- # (*hash*)
 - comment, 1773
- in doctests, 1506
 - in printf-style formatting, 53, 67
 - in regular expressions, 117
 - in string formatting, 103
- \$ (*dollar*)
 - environment variables expansion, 391
 - in regular expressions, 111
 - in template strings, 108
 - interpolation in configuration files, 522
- % (*percent*)
 - datetime format, 208, 626, 627
 - environment variables expansion (*Windows*), 391, 1871
 - interpolation in configuration files, 522
 - operador, 31
 - printf-style formatting, 52, 66
- & (*ampersand*)
 - operador, 32
- (?
 - in regular expressions, 112
- (?!
 - in regular expressions, 113
- (?#
 - in regular expressions, 113
- () (*parentheses*)
 - in printf-style formatting, 52, 66
 - in regular expressions, 112
- (?:
 - in regular expressions, 112
- (?<!
 - in regular expressions, 114
- (?<=
 - in regular expressions, 113
- (?=
 - in regular expressions, 113
- (?P<
 - in regular expressions, 113
- (?P=
 - in regular expressions, 113
- *?
 - in regular expressions, 111

- * (*asterisk*)
 - in argparse module, 645
 - in AST grammar, 1818
 - in glob-style wildcards, 407, 409
 - in printf-style formatting, 52, 66
 - in regular expressions, 111operador, 31
- **
 - in glob-style wildcards, 408operador, 31
- +?
 - in regular expressions, 111
- + (*plus*)
 - binary operator, 31
 - in argparse module, 645
 - in doctests, 1506
 - in printf-style formatting, 53, 67
 - in regular expressions, 111
 - in string formatting, 103unary operator, 31
- , (*comma*)
 - in string formatting, 103
- / (*slash*)
 - in pathnames, 608operador, 31
- //
 - operador, 31
- 2-digit years, 622
- 2to3, 1935
- : (*colon*)
 - in SQL statements, 456
 - in string formatting, 101
 - path separator (*POSIX*), 608
- ; (*semicolon*), 608
- < (*less*)
 - in string formatting, 103
 - in struct format strings, 154operador, 30
- <<
 - operador, 32
- <=
 - operador, 30
- <BLANKLINE>, 1504
- !=
 - operador, 30
- = (*equals*)
 - in string formatting, 103
 - in struct format strings, 154
- ==
 - operador, 30
- > (*greater*)
 - in string formatting, 103
 - in struct format strings, 154operador, 30
- >=
 - operador, 30
- >>
 - operador, 32
- >>>, 1935
 - interpreter prompt, 1501, 1704
- @ (*at*)
 - in struct format strings, 154
- [] (*square brackets*)
 - in glob-style wildcards, 407, 409
 - in regular expressions, 111
 - in string formatting, 101
- \ (*backslash*)
 - escape sequence, 162
 - in pathnames (*Windows*), 608
 - in regular expressions, 111, 112, 114
- \\
 - in regular expressions, 115
- \\a
 - in regular expressions, 115
- \\A
 - in regular expressions, 114
- \\b
 - in regular expressions, 114, 115
- \\B
 - in regular expressions, 114
- \\d
 - in regular expressions, 114
- \\D
 - in regular expressions, 114
- \\f
 - in regular expressions, 115
- \\g
 - in regular expressions, 119
- \\n
 - in regular expressions, 115
- \\N
 - escape sequence, 162
 - in regular expressions, 115
- \\r
 - in regular expressions, 115
- \\s
 - in regular expressions, 114
- \\S
 - in regular expressions, 115
- \\t
 - in regular expressions, 115
- \\u
 - escape sequence, 162
 - in regular expressions, 115
- \\U
 - escape sequence, 162
 - in regular expressions, 115
- \\v
 - in regular expressions, 115
- \\w
 - in regular expressions, 115
- \\W
 - in regular expressions, 115
- \\x
 - escape sequence, 162
 - in regular expressions, 115

- \Z
 - in regular expressions, 115
- ^ (caret)
 - in curses module, 725
 - in regular expressions, 111, 112
 - in string formatting, 103
 - marker, 1503, 1747
 - operador, 32
- _ (underscore)
 - gettext, 1368
 - in string formatting, 103
- __abs__() (en el módulo operator), 366
- __add__() (en el módulo operator), 366
- __and__() (en el módulo operator), 366
- __bases__ (atributo de class), 85
- __breakpointhook__ (en el módulo sys), 1695
- __bytes__() (método de email.message.EmailMessage), 1039
- __bytes__() (método de email.message.Message), 1077
- __call__() (método de email.headerregistry.HeaderRegistry), 1065
- __call__() (método de weakref.finalize), 249
- __callback__ (atributo de weakref.ref), 247
- __cause__ (atributo de traceback.TracebackException), 1749
- __ceil__() (método de fractions.Fraction), 325
- __class__ (atributo de instance), 85
- __class__ (atributo de unittest.mock.Mock), 1557
- __code__ (function object attribute), 84
- __concat__() (en el módulo operator), 367
- __contains__() (en el módulo operator), 367
- __contains__() (método de email.message.EmailMessage), 1040
- __contains__() (método de email.message.Message), 1079
- __contains__() (método de mailbox.Mailbox), 1109
- __context__ (atributo de traceback.TracebackException), 1749
- __copy__() (copy protocol), 259
- __debug__ (variable incorporada), 27
- __deepcopy__() (copy protocol), 259
- __del__() (método de io.IOBase), 614
- __delitem__() (en el módulo operator), 367
- __delitem__() (método de email.message.EmailMessage), 1040
- __delitem__() (método de email.message.Message), 1079
- __delitem__() (método de mailbox.Mailbox), 1107
- __delitem__() (método de mailbox.MH), 1113
- __dict__ (atributo de object), 85
- __dir__() (método de unittest.mock.Mock), 1554
- __displayhook__ (en el módulo sys), 1695
- __doc__ (atributo de types.ModuleType), 255
- __enter__() (método de contextmanager), 82
- __enter__() (método de winreg.PyHKEY), 1877
- __eq__() (en el módulo operator), 365
- __eq__() (instance method), 30
- __eq__() (método de email.charset.Charset), 1090
- __eq__() (método de email.header.Header), 1088
- __eq__() (método de memoryview), 70
- __excepthook__ (en el módulo sys), 1695
- __exit__() (método de contextmanager), 82
- __exit__() (método de winreg.PyHKEY), 1877
- __floor__() (método de fractions.Fraction), 325
- __floordiv__() (en el módulo operator), 366
- __format__, 12
- __format__() (método de datetime.date), 186
- __format__() (método de datetime.datetime), 195
- __format__() (método de datetime.time), 200
- __fspath__() (método de os.PathLike), 561
- __future__, 1939
- __future__ (módulo), 1753
- __ge__() (en el módulo operator), 365
- __ge__() (instance method), 30
- __getitem__() (en el módulo operator), 367
- __getitem__() (método de email.headerregistry.HeaderRegistry), 1065
- __getitem__() (método de email.message.EmailMessage), 1040
- __getitem__() (método de email.message.Message), 1079
- __getitem__() (método de mailbox.Mailbox), 1108
- __getitem__() (método de re.Match), 123
- __getnewargs__() (método de object), 428
- __getnewargs_ex__() (método de object), 428
- __getstate__() (copy protocol), 432
- __getstate__() (método de object), 428
- __gt__() (en el módulo operator), 365
- __gt__() (instance method), 30
- __iadd__() (en el módulo operator), 370
- __iand__() (en el módulo operator), 370
- __iconcat__() (en el módulo operator), 370
- __ifloordiv__() (en el módulo operator), 371
- __ilshift__() (en el módulo operator), 371
- __imatmul__() (en el módulo operator), 371
- __imod__() (en el módulo operator), 371
- __import__() (en el módulo importlib), 1790
- __import__() (función incorporada), 25
- __imul__() (en el módulo operator), 371
- __index__() (en el módulo operator), 366
- __init__() (método de difflib.HtmlDiff), 130
- __init__() (método de logging.Handler), 669
- __interactivehook__ (en el módulo sys), 1702
- __inv__() (en el módulo operator), 366
- __invert__() (en el módulo operator), 366
- __ior__() (en el módulo operator), 371
- __ipow__() (en el módulo operator), 371
- __irshift__() (en el módulo operator), 371
- __isub__() (en el módulo operator), 371
- __iter__() (método de container), 37
- __iter__() (método de iterator), 37
- __iter__() (método de mailbox.Mailbox), 1108
- __iter__() (método de unittest.TestSuite), 1538
- __itruediv__() (en el módulo operator), 371

- `__ixor__()` (en el módulo *operator*), 371
- `__le__()` (en el módulo *operator*), 365
- `__le__()` (instance method), 30
- `__len__()` (método de *email.message.EmailMessage*), 1040
- `__len__()` (método de *email.message.Message*), 1078
- `__len__()` (método de *mailbox.Mailbox*), 1109
- `__loader__` (atributo de *types.ModuleType*), 255
- `__lshift__()` (en el módulo *operator*), 366
- `__lt__()` (en el módulo *operator*), 365
- `__lt__()` (instance method), 30
- `__main__`
 - módulo, 1787, 1788
- `__main__` (módulo), 1714
- `__matmul__()` (en el módulo *operator*), 366
- `__missing__()`, 78
- `__missing__()` (método de *collections.defaultdict*), 226
- `__mod__()` (en el módulo *operator*), 366
- `__mro__` (atributo de class), 86
- `__mul__()` (en el módulo *operator*), 366
- `__name__` (atributo de definición), 85
- `__name__` (atributo de *types.ModuleType*), 256
- `__ne__()` (en el módulo *operator*), 365
- `__ne__()` (instance method), 30
- `__ne__()` (método de *email.charset.Charset*), 1090
- `__ne__()` (método de *email.header.Header*), 1088
- `__neg__()` (en el módulo *operator*), 367
- `__next__()` (método de *csv.csvreader*), 516
- `__next__()` (método de iterator), 37
- `__not__()` (en el módulo *operator*), 366
- `__or__()` (en el módulo *operator*), 367
- `__package__` (atributo de *types.ModuleType*), 256
- `__pos__()` (en el módulo *operator*), 367
- `__pow__()` (en el módulo *operator*), 367
- `__qualname__` (atributo de definición), 85
- `__reduce__()` (método de object), 429
- `__reduce_ex__()` (método de object), 429
- `__repr__()` (método de *multiprocessing.managers.BaseProxy*), 808
- `__repr__()` (método de *netrc.netrc*), 536
- `__round__()` (método de *fractions.Fraction*), 325
- `__rshift__()` (en el módulo *operator*), 367
- `__setitem__()` (en el módulo *operator*), 367
- `__setitem__()` (método de *email.message.EmailMessage*), 1040
- `__setitem__()` (método de *email.message.Message*), 1079
- `__setitem__()` (método de *mailbox.Mailbox*), 1107
- `__setitem__()` (método de *mailbox.Maildir*), 1110
- `__setstate__()` (copy protocol), 432
- `__setstate__()` (método de object), 428
- `__slots__`, 1946
- `__spec__` (atributo de *types.ModuleType*), 256
- `__stderr__` (en el módulo *sys*), 1708
- `__stdin__` (en el módulo *sys*), 1708
- `__stdout__` (en el módulo *sys*), 1708
- `__str__()` (método de *datetime.date*), 185
- `__str__()` (método de *datetime.datetime*), 195
- `__str__()` (método de *datetime.time*), 200
- `__str__()` (método de *email.charset.Charset*), 1090
- `__str__()` (método de *email.header.Header*), 1088
- `__str__()` (método de *email.headerregistry.Address*), 1066
- `__str__()` (método de *email.headerregistry.Group*), 1066
- `__str__()` (método de *email.message.EmailMessage*), 1039
- `__str__()` (método de *email.message.Message*), 1076
- `__str__()` (método de *multiprocessing.managers.BaseProxy*), 808
- `__sub__()` (en el módulo *operator*), 367
- `__subclasses__()` (método de class), 86
- `__subclasshook__()` (método de *abc.ABCMeta*), 1742
- `__suppress_context__` (atributo de *traceback.TracebackException*), 1749
- `__truediv__()` (en el módulo *operator*), 367
- `__unraisablehook__` (en el módulo *sys*), 1695
- `__xor__()` (en el módulo *operator*), 367
- `_anonymous_` (atributo de *ctypes.Structure*), 766
- `_asdict()` (método de *collections.somenamedtuple*), 228
- `_b_base_` (atributo de *ctypes._CData*), 763
- `_b_needsfree_` (atributo de *ctypes._CData*), 763
- `_callmethod()` (método de *multiprocessing.managers.BaseProxy*), 808
- `_CData` (clase en *ctypes*), 762
- `_clear_type_cache()` (en el módulo *sys*), 1693
- `_current_frames()` (en el módulo *sys*), 1693
- `_debugmallocstats()` (en el módulo *sys*), 1693
- `_dummy_thread` (módulo), 864
- `_enablelegacywindowsfsencoding()` (en el módulo *sys*), 1707
- `_exit()` (en el módulo *os*), 597
- `_field_defaults` (atributo de *collections.somenamedtuple*), 229
- `_fields` (atributo de *ast.AST*), 1818
- `_fields` (atributo de *collections.somenamedtuple*), 229
- `_fields_` (atributo de *ctypes.Structure*), 766
- `_flush()` (método de *wsgiref.handlers.BaseHandler*), 1216
- `_FuncPtr` (clase en *ctypes*), 757
- `_get_child_mock()` (método de *unittest.mock.Mock*), 1554
- `_getframe()` (en el módulo *sys*), 1699
- `_getvalue()` (método de *multiprocessing.managers.BaseProxy*), 808
- `_handle` (atributo de *ctypes.PyDLL*), 756
- `_length_` (atributo de *ctypes.Array*), 767
- `_locale`
 - módulo, 1376
- `_make()` (método de clase de *collections.somenamedtuple*), 228
- `_makeResult()` (método de *unittest.TextTestRunner*), 1543

_name (atributo de `ctypes.PyDLL`), 756
 _objects (atributo de `ctypes._CData`), 763
 pack (atributo de `ctypes.Structure`), 766
 _parse() (método de `gettext.NullTranslations`), 1370
 _Pointer (clase en `ctypes`), 767
 _replace() (método de `collections.somenamedtuple`), 229
 _setroot() (método de `xml.etree.ElementTree.ElementTree`), 1156
 _SimpleCData (clase en `ctypes`), 763
 _structure() (en el módulo `email.iterators`), 1095
 _thread (módulo), 862
 type (atributo de `ctypes._Pointer`), 767
 type (atributo de `ctypes.Array`), 767
 _write() (método de `wsgiref.handlers.BaseHandler`), 1216
 _xoptions (en el módulo `sys`), 1710
 {} (curly brackets)
 in regular expressions, 111
 in string formatting, 101
 | (vertical bar)
 in regular expressions, 112
 operador, 32
 ~ (tilde)
 home directory expansion, 391
 operador, 32

A

-a
 pickletools command line option, 1855
 A (en el módulo `re`), 116
 a la espera, 1936
 a2b_base64() (en el módulo `binascii`), 1131
 a2b_hex() (en el módulo `binascii`), 1132
 a2b_hqx() (en el módulo `binascii`), 1131
 a2b_qp() (en el módulo `binascii`), 1131
 a2b_uu() (en el módulo `binascii`), 1130
 a85decode() (en el módulo `base64`), 1128
 a85encode() (en el módulo `base64`), 1128
 ABC (clase en `abc`), 1741
 abc (módulo), 1741
 ABCMeta (clase en `abc`), 1742
 abiflags (en el módulo `sys`), 1691
 abort() (en el módulo `os`), 596
 abort() (método de `asyncio.DatagramTransport`), 925
 abort() (método de `asyncio.WriteTransport`), 924
 abort() (método de `ftplib.FTP`), 1259
 abort() (método de `threading.Barrier`), 781
 above() (método de `curses.panel.Panel`), 726
 ABOVE_NORMAL_PRIORITY_CLASS (en el módulo `subprocess`), 847
 abs() (en el módulo `operator`), 366
 abs() (función incorporada), 5
 abs() (método de `decimal.Context`), 311
 AbsoluteLinkError, 497
 AbsolutePathError, 497
 abspath() (en el módulo `os.path`), 390

AbstractAsyncContextManager (clase en `contextlib`), 1729
 AbstractBasicAuthHandler (clase en `urllib.request`), 1223
 AbstractChildWatcher (clase en `asyncio`), 936
 abstractclassmethod() (en el módulo `abc`), 1744
 AbstractContextManager (clase en `contextlib`), 1729
 AbstractDigestAuthHandler (clase en `urllib.request`), 1224
 AbstractEventLoop (clase en `asyncio`), 915
 AbstractEventLoopPolicy (clase en `asyncio`), 935
 AbstractFormatter (clase en `formatter`), 1859
 abstractmethod() (en el módulo `abc`), 1743
 abstractproperty() (en el módulo `abc`), 1745
 AbstractSet (clase en `typing`), 1486
 abstractstaticmethod() (en el módulo `abc`), 1744
 AbstractWriter (clase en `formatter`), 1860
 accept() (método de `asyncore.dispatcher`), 1019
 accept() (método de `multiprocessing.connection.Listener`), 812
 accept() (método de `socket.socket`), 962
 access() (en el módulo `os`), 576
 accumulate() (en el módulo `itertools`), 345
 aclose() (método de `contextlib.AsyncExitStack`), 1735
 acos() (en el módulo `cmath`), 296
 acos() (en el módulo `math`), 292
 acosh() (en el módulo `cmath`), 296
 acosh() (en el módulo `math`), 293
 acquire() (método de `_thread.lock`), 863
 acquire() (método de `asyncio.Condition`), 887
 acquire() (método de `asyncio.Lock`), 885
 acquire() (método de `asyncio.Semaphore`), 888
 acquire() (método de `logging.Handler`), 669
 acquire() (método de `multiprocessing.Lock`), 798
 acquire() (método de `multiprocessing.RLock`), 798
 acquire() (método de `threading.Condition`), 776
 acquire() (método de `threading.Lock`), 774
 acquire() (método de `threading.RLock`), 775
 acquire() (método de `threading.Semaphore`), 778
 acquire_lock() (en el módulo `imp`), 1931
 action (atributo de `optparse.Option`), 1914
 Action (clase en `argparse`), 651
 ACTIONS (atributo de `optparse.Option`), 1926
 active_children() (en el módulo `multiprocessing`), 794
 active_count() (en el módulo `threading`), 769
 add() (en el módulo `audioop`), 1347
 add() (en el módulo `operator`), 366
 add() (método de `decimal.Context`), 311
 add() (método de `frozenset`), 77
 add() (método de `mailbox.Mailbox`), 1107
 add() (método de `mailbox.Maildir`), 1110
 add() (método de `msilib.RadioButtonGroup`), 1866
 add() (método de `pstats.Stats`), 1650

- `add()` (método de `tarfile.TarFile`), 500
- `add()` (método de `tkinter.ttk.Notebook`), 1447
- `add_alias()` (en el módulo `email.charset`), 1091
- `add_alternative()` (método de `email.message.EmailMessage`), 1045
- `add_argument()` (método de `argparse.ArgumentParser`), 641
- `add_argument_group()` (método de `argparse.ArgumentParser`), 658
- `add_attachment()` (método de `email.message.EmailMessage`), 1045
- `add_cgi_vars()` (método de `wsgiref.handlers.BaseHandler`), 1216
- `add_charset()` (en el módulo `email.charset`), 1090
- `add_child_handler()` (método de `asyncio.AbstractChildWatcher`), 936
- `add_codec()` (en el módulo `email.charset`), 1091
- `add_cookie_header()` (método de `http.cookiejar.CookieJar`), 1313
- `add_data()` (en el módulo `msilib`), 1862
- `add_dll_directory()` (en el módulo `os`), 596
- `add_done_callback()` (método de `asyncio.Future`), 919
- `add_done_callback()` (método de `asyncio.Task`), 877
- `add_done_callback()` (método de `concurrent.futures.Future`), 834
- `add_fallback()` (método de `gettext.NullTranslations`), 1370
- `add_file()` (método de `msilib.Directory`), 1865
- `add_flag()` (método de `mailbox.MaildirMessage`), 1116
- `add_flag()` (método de `mailbox.mboxMessage`), 1117
- `add_flag()` (método de `mailbox.MMDfMessage`), 1121
- `add_flowng_data()` (método de `formatter.formatter`), 1858
- `add_folder()` (método de `mailbox.Maildir`), 1110
- `add_folder()` (método de `mailbox.MH`), 1112
- `add_get_handler()` (método de `email.contentmanager.ContentManager`), 1067
- `add_handler()` (método de `urllib.request.OpenerDirector`), 1226
- `add_header()` (método de `email.message.EmailMessage`), 1041
- `add_header()` (método de `email.message.Message`), 1079
- `add_header()` (método de `urllib.request.Request`), 1225
- `add_header()` (método de `wsgiref.headers.Headers`), 1212
- `add_history()` (en el módulo `readline`), 149
- `add_hor_rule()` (método de `formatter.formatter`), 1858
- `add_label()` (método de `mailbox.BabylMessage`), 1120
- `add_label_data()` (método de `formatter.formatter`), 1858
- `add_line_break()` (método de `formatter.formatter`), 1858
- `add_literal_data()` (método de `formatter.formatter`), 1858
- `add_mutually_exclusive_group()` (método de `argparse.ArgumentParser`), 659
- `add_option()` (método de `optparse.OptionParser`), 1913
- `add_parent()` (método de `urllib.request.BaseHandler`), 1227
- `add_password()` (método de `urllib.request.HTTPPasswordMgr`), 1229
- `add_password()` (método de `urllib.request.HTTPPasswordMgrWithPriorAuth`), 1230
- `add_reader()` (método de `asyncio.loop`), 907
- `add_related()` (método de `email.message.EmailMessage`), 1045
- `add_section()` (método de `configparser.ConfigParser`), 531
- `add_section()` (método de `configparser.RawConfigParser`), 534
- `add_sequence()` (método de `mailbox.MHMessage`), 1119
- `add_set_handler()` (método de `email.contentmanager.ContentManager`), 1067
- `add_signal_handler()` (método de `asyncio.loop`), 909
- `add_stream()` (en el módulo `msilib`), 1862
- `add_subparsers()` (método de `argparse.ArgumentParser`), 654
- `add_tables()` (en el módulo `msilib`), 1862
- `add_type()` (en el módulo `mimetypes`), 1125
- `add_unredirected_header()` (método de `urllib.request.Request`), 1225
- `add_writer()` (método de `asyncio.loop`), 907
- `addAsyncCleanup()` (método de `unittest.IsolatedAsyncioTestCase`), 1536
- `addaudithook()` (en el módulo `sys`), 1691
- `addch()` (método de `curses.window`), 712
- `addClassCleanup()` (método de clase de `unittest.TestCase`), 1536
- `addCleanup()` (método de `unittest.TestCase`), 1535
- `addcomponent()` (método de `turtle.Shape`), 1413
- `addError()` (método de `unittest.TestResult`), 1542
- `addExpectedFailure()` (método de `unittest.TestResult`), 1543
- `addFailure()` (método de `unittest.TestResult`), 1542
- `addfile()` (método de `tarfile.TarFile`), 500
- `addFilter()` (método de `logging.Handler`), 670
- `addFilter()` (método de `logging.Logger`), 668
- `addHandler()` (método de `logging.Logger`), 668
- `addLevelName()` (en el módulo `logging`), 678
- `addModuleCleanup()` (en el módulo `unittest`), 1547
- `addnstr()` (método de `curses.window`), 712
- `AddPackagePath()` (en el módulo `modulefinder`),

- 1786
- `addr` (atributo de `smtplib.SMTPChannel`), 1287
- `addr_spec` (atributo de `email.headerregistry.Address`), 1066
- `address` (atributo de `email.headerregistry.SingleAddressHeader`), 1064
- `address` (atributo de `multiprocessing.connection.Listener`), 813
- `address` (atributo de `multiprocessing.managers.BaseManager`), 804
- `Address` (clase en `email.headerregistry`), 1066
- `address_exclude()` (método de `ipaddress.IPv4Network`), 1340
- `address_exclude()` (método de `ipaddress.IPv6Network`), 1342
- `address_family` (atributo de `socketserver.BaseServer`), 1297
- `address_string()` (método de `http.server.BaseHTTPRequestHandler`), 1305
- `addresses` (atributo de `email.headerregistry.AddressHeader`), 1063
- `addresses` (atributo de `email.headerregistry.Group`), 1066
- `AddressHeader` (clase en `email.headerregistry`), 1063
- `addressof()` (en el módulo `ctypes`), 760
- `AddressValueError`, 1346
- `addshape()` (en el módulo `turtle`), 1411
- `addsitedir()` (en el módulo `site`), 1775
- `addSkip()` (método de `unittest.TestResult`), 1543
- `addstr()` (método de `curses.window`), 712
- `addSubTest()` (método de `unittest.TestResult`), 1543
- `addSuccess()` (método de `unittest.TestResult`), 1543
- `addTest()` (método de `unittest.TestSuite`), 1538
- `addTests()` (método de `unittest.TestSuite`), 1538
- `addTypeEqualityFunc()` (método de `unittest.TestCase`), 1534
- `addUnexpectedSuccess()` (método de `unittest.TestResult`), 1543
- `adjust_int_max_str_digits()` (en el módulo `test.support`), 1626
- `adjusted()` (método de `decimal.Decimal`), 303
- `adler32()` (en el módulo `zlib`), 469
- administrador asincrónico de contexto, 1936
- administrador de contextos, 1937
- ADPCM, Intel/DVI, 1347
- `adpcm2lin()` (en el módulo `audioop`), 1347
- AF_ALG (en el módulo `socket`), 955
- AF_CAN (en el módulo `socket`), 954
- AF_INET (en el módulo `socket`), 953
- AF_INET6 (en el módulo `socket`), 953
- AF_LINK (en el módulo `socket`), 956
- AF_PACKET (en el módulo `socket`), 955
- AF_QIPCRTR (en el módulo `socket`), 956
- AF_RDS (en el módulo `socket`), 955
- AF_UNIX (en el módulo `socket`), 953
- AF_VSOCK (en el módulo `socket`), 955
- `aifc` (módulo), 1350
- `aifc()` (método de `aifc.aifc`), 1352
- AIFF, 1350, 1358
- `aiff()` (método de `aifc.aifc`), 1351
- AIFF-C, 1350, 1358
- `alarm()` (en el módulo `signal`), 1027
- a-LAW, 1347
- A-LAW, 1352, 1361
- `alaw2lin()` (en el módulo `audioop`), 1347
- alcances anidados, 1943
- ALERT_DESCRIPTION_HANDSHAKE_FAILURE (en el módulo `ssl`), 984
- ALERT_DESCRIPTION_INTERNAL_ERROR (en el módulo `ssl`), 984
- `AlertDescription` (clase en `ssl`), 984
- `algorithms_available` (en el módulo `hashlib`), 544
- `algorithms_guaranteed` (en el módulo `hashlib`), 544
- `alias` (`pdb` command), 1645
- alias de tipos, 1946
- `alignment()` (en el módulo `ctypes`), 760
- `alive` (atributo de `weakref.finalize`), 249
- `all()` (función incorporada), 5
- `all_errors` (en el módulo `ftplib`), 1259
- `all_features` (en el módulo `xml.sax.handler`), 1180
- `all_frames` (atributo de `tracemalloc.Filter`), 1668
- `all_properties` (en el módulo `xml.sax.handler`), 1181
- `all_suffixes()` (en el módulo `importlib.machinery`), 1799
- `all_tasks()` (en el módulo `asyncio`), 875
- `all_tasks()` (método de clase de `asyncio.Task`), 878
- `allocate_lock()` (en el módulo `thread`), 863
- `allow_reuse_address` (atributo de `socketserver.BaseServer`), 1297
- `allowed_domains()` (método de `http.cookiejar.DefaultCookiePolicy`), 1316
- `alt()` (en el módulo `curses.ascii`), 725
- ALT_DIGITS (en el módulo `locale`), 1379
- `altsep` (en el módulo `os`), 608
- `altzone` (en el módulo `time`), 631
- ALWAYS_EQ (en el módulo `test.support`), 1616
- ALWAYS_TYPED_ACTIONS (atributo de `optparse.Option`), 1927
- AMPER (en el módulo `token`), 1827
- AMPEREQUAL (en el módulo `token`), 1828
- and
- operador, 29, 30
- `and_()` (en el módulo `operator`), 366
- annotate
- `pickletools` command line option, 1855
- annotation (atributo de `inspect.Parameter`), 1764
- anotación, 1935
- anotación de función, 1939
- anotación de variable, 1947

- `answer_challenge()` (en el módulo `multiprocessing.connection`), 812
- `anticipate_failure()` (en el módulo `test.support`), 1621
- `Any` (en el módulo `typing`), 1493
- `ANY` (en el módulo `unittest.mock`), 1581
- `any()` (función incorporada), 5
- `AnyStr` (en el módulo `typing`), 1495
- apagado del intérprete, 1941
- API provisoria, 1945
- `api_version` (en el módulo `sys`), 1710
- `apop()` (método de `poplib.POP3`), 1263
- `append()` (método de `array.array`), 244
- `append()` (método de `collections.deque`), 223
- `append()` (método de `email.header.Header`), 1087
- `append()` (método de `imaplib.IMAP4`), 1267
- `append()` (método de `msilib.CAB`), 1864
- `append()` (método de `pipes.Template`), 1892
- `append()` (método de `xml.etree.ElementTree.Element`), 1154
- `append()` (sequence method), 40
- `append_history_file()` (en el módulo `readline`), 148
- `appendChild()` (método de `xml.dom.Node`), 1164
- `appendleft()` (método de `collections.deque`), 223
- `application_uri()` (en el módulo `wsgiref.util`), 1210
- `apply (2to3 fixer)`, 1608
- `apply()` (método de `multiprocessing.pool.Pool`), 809
- `apply_async()` (método de `multiprocessing.pool.Pool`), 809
- `apply_defaults()` (método de `inspect.BoundArguments`), 1766
- `architecture()` (en el módulo `platform`), 727
- `archive` (atributo de `zipimport.zipimporter`), 1782
- archivo binario, 1936
- archivo de texto, 1946
- `aRepr` (en el módulo `reprlib`), 264
- `argparse` (módulo), 632
- `args` (atributo de `BaseException`), 90
- `args` (atributo de `functools.partial`), 365
- `args` (atributo de `inspect.BoundArguments`), 1766
- `args` (atributo de `subprocess.CompletedProcess`), 837
- `args` (atributo de `subprocess.Popen`), 845
- `args` (`pdb command`), 1644
- `args_from_interpreter_flags()` (en el módulo `test.support`), 1619
- `argtypes` (atributo de `ctypes._FuncPtr`), 757
- `ArgumentDefaultsHelpFormatter` (clase en `argparse`), 637
- `ArgumentError`, 757
- argumento, 1935
- argumento nombrado, 1942
- argumento posicional, 1945
- `ArgumentParser` (clase en `argparse`), 634
- `arguments` (atributo de `inspect.BoundArguments`), 1766
- `argv` (en el módulo `sys`), 1691
- `arithmetic`, 31
- `ArithmeticError`, 90
- `array`
módulo, 54
- `array` (clase en `array`), 244
- `Array` (clase en `ctypes`), 767
- `array` (módulo), 243
- `Array()` (en el módulo `multiprocessing`), 800
- `Array()` (en el módulo `multiprocessing.sharedctypes`), 801
- `Array()` (método de `multiprocessing.managers.SyncManager`), 804
- `arrays`, 243
- `arraysize` (atributo de `sqlite3.Cursor`), 459
- `article()` (método de `nnplib.NNTP`), 1276
- `as_bytes()` (método de `email.message.EmailMessage`), 1039
- `as_bytes()` (método de `email.message.Message`), 1076
- `as_completed()` (en el módulo `asyncio`), 874
- `as_completed()` (en el módulo `concurrent.futures`), 835
- `as_integer_ratio()` (método de `decimal.Decimal`), 303
- `as_integer_ratio()` (método de `float`), 34
- `as_integer_ratio()` (método de `fractions.Fraction`), 325
- `as_integer_ratio()` (método de `int`), 34
- `AS_IS` (en el módulo `formatter`), 1857
- `as_posix()` (método de `pathlib.PurePath`), 379
- `as_string()` (método de `email.message.EmailMessage`), 1039
- `as_string()` (método de `email.message.Message`), 1076
- `as_tuple()` (método de `decimal.Decimal`), 304
- `as_uri()` (método de `pathlib.PurePath`), 380
- `ASCII` (en el módulo `re`), 116
- `ascii()` (en el módulo `curses.ascii`), 725
- `ascii()` (función incorporada), 6
- `ascii_letters` (en el módulo `string`), 99
- `ascii_lowercase` (en el módulo `string`), 99
- `ascii_uppercase` (en el módulo `string`), 99
- `asctime()` (en el módulo `time`), 623
- `asdict()` (en el módulo `dataclasses`), 1725
- `asin()` (en el módulo `cmath`), 296
- `asin()` (en el módulo `math`), 292
- `asinh()` (en el módulo `cmath`), 296
- `asinh()` (en el módulo `math`), 293
- `assert`
sentencia, 90
- `assert_any_await()` (método de `unittest.mock.AsyncMock`), 1561
- `assert_any_call()` (método de `unittest.mock.Mock`), 1552
- `assert_awaited()` (método de `unittest.mock.AsyncMock`), 1560
- `assert_awaited_once()` (método de `unittest.mock.AsyncMock`), 1560

- `assert_awaited_once_with()` (método de `unittest.mock.AsyncMock`), 1561
- `assert_awaited_with()` (método de `unittest.mock.AsyncMock`), 1561
- `assert_called()` (método de `unittest.mock.Mock`), 1551
- `assert_called_once()` (método de `unittest.mock.Mock`), 1551
- `assert_called_once_with()` (método de `unittest.mock.Mock`), 1552
- `assert_called_with()` (método de `unittest.mock.Mock`), 1551
- `assert_has_awaits()` (método de `unittest.mock.AsyncMock`), 1561
- `assert_has_calls()` (método de `unittest.mock.Mock`), 1552
- `assert_line_data()` (método de `formatter.formatter`), 1859
- `assert_not_awaited()` (método de `unittest.mock.AsyncMock`), 1562
- `assert_not_called()` (método de `unittest.mock.Mock`), 1552
- `assert_python_failure()` (en el módulo `test.support.script_helper`), 1627
- `assert_python_ok()` (en el módulo `test.support.script_helper`), 1627
- `assertAlmostEqual()` (método de `unittest.TestCase`), 1533
- `assertCountEqual()` (método de `unittest.TestCase`), 1533
- `assertDictEqual()` (método de `unittest.TestCase`), 1535
- `assertEqual()` (método de `unittest.TestCase`), 1529
- `assertFalse()` (método de `unittest.TestCase`), 1530
- `assertGreater()` (método de `unittest.TestCase`), 1533
- `assertGreaterEqual()` (método de `unittest.TestCase`), 1533
- `assertIn()` (método de `unittest.TestCase`), 1530
- `AssertionError`, 90
- `assertIs()` (método de `unittest.TestCase`), 1530
- `assertIsInstance()` (método de `unittest.TestCase`), 1530
- `assertIsNone()` (método de `unittest.TestCase`), 1530
- `assertIsNot()` (método de `unittest.TestCase`), 1530
- `assertIsNotNone()` (método de `unittest.TestCase`), 1530
- `assertLess()` (método de `unittest.TestCase`), 1533
- `assertLessEqual()` (método de `unittest.TestCase`), 1533
- `assertListEqual()` (método de `unittest.TestCase`), 1534
- `assertLogs()` (método de `unittest.TestCase`), 1532
- `assertMultiLineEqual()` (método de `unittest.TestCase`), 1534
- `assertNotAlmostEqual()` (método de `unittest.TestCase`), 1533
- `assertNotEqual()` (método de `unittest.TestCase`), 1530
- `assertNotIn()` (método de `unittest.TestCase`), 1530
- `assertNotIsInstance()` (método de `unittest.TestCase`), 1530
- `assertNotRegex()` (método de `unittest.TestCase`), 1533
- `assertRaises()` (método de `unittest.TestCase`), 1530
- `assertRaisesRegex()` (método de `unittest.TestCase`), 1531
- `assertRegex()` (método de `unittest.TestCase`), 1533
- `asserts (2to3 fixer)`, 1608
- `assertSequenceEqual()` (método de `unittest.TestCase`), 1534
- `assertSetEqual()` (método de `unittest.TestCase`), 1534
- `assertTrue()` (método de `unittest.TestCase`), 1530
- `assertTupleEqual()` (método de `unittest.TestCase`), 1534
- `assertWarns()` (método de `unittest.TestCase`), 1531
- `assertWarnsRegex()` (método de `unittest.TestCase`), 1532
- `assignment`
- `slice`, 40
 - `subscript`, 40
- `AST (clase en ast)`, 1818
- `ast (módulo)`, 1817
- `astimezone()` (método de `datetime.datetime`), 192
- `astuple()` (en el módulo `dataclasses`), 1725
- `ASYNC (en el módulo token)`, 1829
- `async_chat (clase en asynchat)`, 1021
- `async_chat.ac_in_buffer_size` (en el módulo `asynchat`), 1021
- `async_chat.ac_out_buffer_size` (en el módulo `asynchat`), 1022
- `AsyncContextManager (clase en typing)`, 1487
- `asyncontextmanager()` (en el módulo `contextlib`), 1730
- `AsyncExitStack (clase en contextlib)`, 1735
- `AsyncGenerator (clase en collections.abc)`, 236
- `AsyncGenerator (clase en typing)`, 1488
- `AsyncGeneratorType (en el módulo types)`, 254
- `asynchat (módulo)`, 1021
- `asyncio (módulo)`, 865
- `asyncio.subprocess.DEVNULL` (en el módulo `asyncio`), 890
- `asyncio.subprocess.PIPE` (en el módulo `asyncio`), 890
- `asyncio.subprocess.Process (clase en asyncio)`, 891
- `asyncio.subprocess.STDOUT` (en el módulo `asyncio`), 890
- `AsyncIterable (clase en collections.abc)`, 236
- `AsyncIterable (clase en typing)`, 1487
- `AsyncIterator (clase en collections.abc)`, 236
- `AsyncIterator (clase en typing)`, 1487
- `AsyncMock (clase en unittest.mock)`, 1559
- `asyncore (módulo)`, 1017
- `AsyncResult (clase en multiprocessing.pool)`, 811

- `asyncSetUp()` (método de `unit-test.IsolatedAsyncioTestCase`), 1536
- `asyncTearDown()` (método de `unit-test.IsolatedAsyncioTestCase`), 1536
- `AT` (en el módulo `token`), 1829
- `at_eof()` (método de `asyncio.StreamReader`), 881
- `atan()` (en el módulo `cmath`), 296
- `atan()` (en el módulo `math`), 292
- `atan2()` (en el módulo `math`), 292
- `atanh()` (en el módulo `cmath`), 296
- `atanh()` (en el módulo `math`), 293
- `ATEQUAL` (en el módulo `token`), 1829
- `atexit` (atributo de `weakref.finalize`), 249
- `atexit` (módulo), 1746
- `atof()` (en el módulo `locale`), 1381
- `atoi()` (en el módulo `locale`), 1381
- `atributo`, 1936
- `attach()` (método de `email.message.Message`), 1077
- `attach_loop()` (método de `asyncio.AbstractChildWatcher`), 936
- `attach_mock()` (método de `unittest.mock.Mock`), 1553
- `AttlistDeclHandler()` (método de `xml.parsers.expat.xmlparser`), 1192
- `attrgetter()` (en el módulo `operator`), 368
- `attrib` (atributo de `xml.etree.ElementTree.Element`), 1154
- `AttributeError`, 90
- `attributes` (atributo de `xml.dom.Node`), 1163
- `AttributesImpl` (clase en `xml.sax.xmlreader`), 1185
- `AttributesNSImpl` (clase en `xml.sax.xmlreader`), 1185
- `attroff()` (método de `curses.window`), 712
- `attron()` (método de `curses.window`), 712
- `attrset()` (método de `curses.window`), 712
- `Audio Interchange File Format`, 1350, 1358
- `AUDIO_FILE_ENCODING_ADPCM_G721` (en el módulo `sunau`), 1353
- `AUDIO_FILE_ENCODING_ADPCM_G722` (en el módulo `sunau`), 1353
- `AUDIO_FILE_ENCODING_ADPCM_G723_3` (en el módulo `sunau`), 1353
- `AUDIO_FILE_ENCODING_ADPCM_G723_5` (en el módulo `sunau`), 1353
- `AUDIO_FILE_ENCODING_ALAW_8` (en el módulo `sunau`), 1353
- `AUDIO_FILE_ENCODING_DOUBLE` (en el módulo `sunau`), 1353
- `AUDIO_FILE_ENCODING_FLOAT` (en el módulo `sunau`), 1353
- `AUDIO_FILE_ENCODING_LINEAR_8` (en el módulo `sunau`), 1353
- `AUDIO_FILE_ENCODING_LINEAR_16` (en el módulo `sunau`), 1353
- `AUDIO_FILE_ENCODING_LINEAR_24` (en el módulo `sunau`), 1353
- `AUDIO_FILE_ENCODING_LINEAR_32` (en el módulo `sunau`), 1353
- `AUDIO_FILE_ENCODING_MULAW_8` (en el módulo `sunau`), 1353
- `AUDIO_FILE_MAGIC` (en el módulo `sunau`), 1353
- `AUDIODEV`, 1362
- `audioop` (módulo), 1347
- `audit events`, 1629
- `audit()` (en el módulo `sys`), 1692
- `auditing`, 1692
- `auth()` (método de `ftplib.FTP_TLS`), 1262
- `auth()` (método de `smtplib.SMTP`), 1281
- `authenticate()` (método de `imaplib.IMAP4`), 1267
- `AuthenticationError`, 791
- `authenticators()` (método de `netrc.netrc`), 536
- `authkey` (atributo de `multiprocessing.Process`), 789
- `auto` (clase en `enum`), 267
- `autorange()` (método de `timeit.Timer`), 1656
- `avg()` (en el módulo `audioop`), 1347
- `avgpp()` (en el módulo `audioop`), 1348
- `avoids_symlink_attacks` (atributo de `shutil.rmtree`), 414
- `AWAIT` (en el módulo `token`), 1829
- `await_args` (atributo de `unittest.mock.AsyncMock`), 1562
- `await_args_list` (atributo de `unittest.mock.AsyncMock`), 1562
- `await_count` (atributo de `unittest.mock.AsyncMock`), 1562
- `Awaitable` (clase en `collections.abc`), 235
- `Awaitable` (clase en `typing`), 1487
- ## B
- `-b`
- `compileall` command line option, 1838
- `unittest` command line option, 1522
- `b2a_base64()` (en el módulo `binascii`), 1131
- `b2a_hex()` (en el módulo `binascii`), 1132
- `b2a_hqx()` (en el módulo `binascii`), 1131
- `b2a_qp()` (en el módulo `binascii`), 1131
- `b2a_uu()` (en el módulo `binascii`), 1131
- `b16decode()` (en el módulo `base64`), 1128
- `b16encode()` (en el módulo `base64`), 1128
- `b32decode()` (en el módulo `base64`), 1128
- `b32encode()` (en el módulo `base64`), 1128
- `b64decode()` (en el módulo `base64`), 1127
- `b64encode()` (en el módulo `base64`), 1127
- `b85decode()` (en el módulo `base64`), 1129
- `b85encode()` (en el módulo `base64`), 1129
- `Babyl` (clase en `mailbox`), 1113
- `BabylMessage` (clase en `mailbox`), 1119
- `back()` (en el módulo `turtle`), 1390
- `backslashreplace_errors()` (en el módulo `codecs`), 163
- `backup()` (método de `sqlite3.Connection`), 455
- `backward()` (en el módulo `turtle`), 1390
- `BadGzipFile`, 473
- `BadStatusLine`, 1252

- BadZipfile, 485
- BadZipFile, 485
- Balloon (clase en *tkinter.tix*), 1460
- Barrier (clase en *multiprocessing*), 797
- Barrier (clase en *threading*), 780
- Barrier() (método de *multiprocessing.managers.SyncManager*), 804
- base64
 - encoding, 1127
 - módulo, 1130
- base64 (módulo), 1127
- base_exec_prefix (en el módulo *sys*), 1692
- base_prefix (en el módulo *sys*), 1692
- BaseCGIHandler (clase en *wsgiref.handlers*), 1215
- BaseCookie (clase en *http.cookies*), 1308
- BaseException, 90
- BaseHandler (clase en *urllib.request*), 1223
- BaseHandler (clase en *wsgiref.handlers*), 1215
- BaseHeader (clase en *email.headerregistry*), 1062
- BaseHTTPRequestHandler (clase en *http.server*), 1302
- BaseManager (clase en *multiprocessing.managers*), 803
- basename() (en el módulo *os.path*), 390
- BaseProtocol (clase en *asyncio*), 926
- BaseProxy (clase en *multiprocessing.managers*), 808
- BaseRequestHandler (clase en *socketserver*), 1298
- BaseRotatingHandler (clase en *logging.handlers*), 694
- BaseSelector (clase en *selectors*), 1015
- BaseServer (clase en *socketserver*), 1296
- basestring (2to3 fixer), 1608
- BaseTransport (clase en *asyncio*), 922
- basicConfig() (en el módulo *logging*), 679
- BasicContext (clase en *decimal*), 309
- BasicInterpolation (clase en *configparser*), 522
- BasicTestRunner (clase en *test.support*), 1627
- baudrate() (en el módulo *curses*), 706
- bbox() (método de *tkinter.ttk.Treeview*), 1452
- BDADDR_ANY (en el módulo *socket*), 956
- BDADDR_LOCAL (en el módulo *socket*), 956
- bdb
 - módulo, 1640
- Bdb (clase en *bdb*), 1634
- bdb (módulo), 1633
- BdbQuit, 1633
- BDFL, 1936
- beep() (en el módulo *curses*), 706
- Beep() (en el módulo *winsound*), 1877
- BEFORE_ASYNC_WITH (opcode), 1847
- begin_fill() (en el módulo *turtle*), 1399
- BEGIN_FINALLY (opcode), 1849
- begin_poly() (en el módulo *turtle*), 1404
- below() (método de *curses.panel.Panel*), 726
- BELOW_NORMAL_PRIORITY_CLASS (en el módulo *subprocess*), 847
- benchmarking, 625, 628
- Benchmarking, 1654
- best
 - gzip command line option, 475
- betavariate() (en el módulo *random*), 329
- bgcolor() (en el módulo *turtle*), 1406
- bgpic() (en el módulo *turtle*), 1406
- bias() (en el módulo *audioop*), 1348
- bidirectional() (en el módulo *unicodedata*), 144
- bigaddrspacetest() (en el módulo *test.support*), 1622
- BigEndianStructure (clase en *ctypes*), 766
- bigmemtest() (en el módulo *test.support*), 1622
- bin() (función incorporada), 6
- binary
 - data, packing, 153
 - literals, 31
- Binary (clase en *msilib*), 1862
- Binary (clase en *xmlrpc.client*), 1323
- binary mode, 19
- binary semaphores, 862
- BINARY_ADD (opcode), 1846
- BINARY_AND (opcode), 1846
- BINARY_FLOOR_DIVIDE (opcode), 1846
- BINARY_LSHIFT (opcode), 1846
- BINARY_MATRIX_MULTIPLY (opcode), 1846
- BINARY_MODULO (opcode), 1846
- BINARY_MULTIPLY (opcode), 1846
- BINARY_OR (opcode), 1846
- BINARY_POWER (opcode), 1846
- BINARY_RSHIFT (opcode), 1846
- BINARY_SUBSCR (opcode), 1846
- BINARY_SUBTRACT (opcode), 1846
- BINARY_TRUE_DIVIDE (opcode), 1846
- BINARY_XOR (opcode), 1846
- BinaryIO (clase en *typing*), 1489
- binascii (módulo), 1130
- bind (widgets), 1438
- bind() (método de *asyncore.dispatcher*), 1019
- bind() (método de *inspect.Signature*), 1763
- bind() (método de *socket.socket*), 963
- bind_partial() (método de *inspect.Signature*), 1763
- bind_port() (en el módulo *test.support*), 1623
- bind_textdomain_codeset() (en el módulo *gettext*), 1368
- bind_unix_socket() (en el módulo *test.support*), 1623
- bindtextdomain() (en el módulo *gettext*), 1367
- bindtextdomain() (en el módulo *locale*), 1383
- binhex
 - módulo, 1130
- binhex (módulo), 1130
- binhex() (en el módulo *binhex*), 1130
- bisect (módulo), 241
- bisect() (en el módulo *bisect*), 241
- bisect_left() (en el módulo *bisect*), 241
- bisect_right() (en el módulo *bisect*), 241
- bit_length() (método de *int*), 33
- bitmap() (método de *msilib.Dialog*), 1866

- bitwise
 - operations, 32
- bk() (en el módulo turtle), 1390
- bkgd() (método de curses.window), 712
- bkgdset() (método de curses.window), 713
- blake2b() (en el módulo hashlib), 547
- blake2b, blake2s, 546
- blake2b.MAX_DIGEST_SIZE (en el módulo hashlib), 548
- blake2b.MAX_KEY_SIZE (en el módulo hashlib), 548
- blake2b.PERSON_SIZE (en el módulo hashlib), 548
- blake2b.SALT_SIZE (en el módulo hashlib), 548
- blake2s() (en el módulo hashlib), 547
- blake2s.MAX_DIGEST_SIZE (en el módulo hashlib), 548
- blake2s.MAX_KEY_SIZE (en el módulo hashlib), 548
- blake2s.PERSON_SIZE (en el módulo hashlib), 548
- blake2s.SALT_SIZE (en el módulo hashlib), 548
- block_size (atributo de hmac.HMAC), 554
- blocked_domains() (método de http.cookiejar.DefaultCookiePolicy), 1316
- BlockingIOError, 95, 611
- blocksize (atributo de http.client.HTTPConnection), 1254
- bloqueo global del intérprete, 1940
- body() (método de nnplib.NNTP), 1277
- body_encode() (método de email.charset.Charset), 1090
- body_encoding (atributo de email.charset.Charset), 1089
- body_line_iterator() (en el módulo email.iterators), 1095
- BOM (en el módulo codecs), 161
- BOM_BE (en el módulo codecs), 161
- BOM_LE (en el módulo codecs), 161
- BOM_UTF8 (en el módulo codecs), 161
- BOM_UTF16 (en el módulo codecs), 161
- BOM_UTF16_BE (en el módulo codecs), 161
- BOM_UTF16_LE (en el módulo codecs), 161
- BOM_UTF32 (en el módulo codecs), 161
- BOM_UTF32_BE (en el módulo codecs), 161
- BOM_UTF32_LE (en el módulo codecs), 161
- bool (clase incorporada), 6
- Boolean
 - objeto, 31
 - operations, 29, 30
 - type, 6
 - values, 85
- BOOLEAN_STATES (atributo de configparser.ConfigParser), 527
- bootstrap() (en el módulo ensurepip), 1675
- border() (método de curses.window), 713
- bottom() (método de curses.panel.Panel), 726
- bottom_panel() (en el módulo curses.panel), 726
- BoundArguments (clase en inspect), 1766
- BoundaryError, 1060
- BoundedSemaphore (clase en asyncio), 889
- BoundedSemaphore (clase en multiprocessing), 797
- BoundedSemaphore (clase en threading), 778
- BoundedSemaphore() (método de multiprocessing.managers.SyncManager), 804
- box() (método de curses.window), 713
- bpformat() (método de bdb.Breakpoint), 1633
- bpprint() (método de bdb.Breakpoint), 1634
- break (pdb command), 1643
- break_anywhere() (método de bdb.Bdb), 1635
- break_here() (método de bdb.Bdb), 1635
- break_long_words (atributo de textwrap.TextWrapper), 143
- break_on_hyphens (atributo de textwrap.TextWrapper), 143
- Breakpoint (clase en bdb), 1633
- breakpoint() (función incorporada), 6
- breakpointhook() (en el módulo sys), 1693
- breakpoints, 1467
- broadcast_address (atributo de ipaddress.IPv4Network), 1339
- broadcast_address (atributo de ipaddress.IPv6Network), 1342
- broken (atributo de threading.Barrier), 781
- BrokenBarrierError, 781
- BrokenExecutor, 836
- BrokenPipeError, 95
- BrokenProcessPool, 836
- BrokenThreadPool, 836
- BROWSER, 1199, 1200
- BsdDbShelf (clase en shelve), 440
- buf (atributo de multiprocessing.shared_memory.SharedMemory), 826
- buffer
 - unittest command line option, 1522
- buffer (2to3 fixer), 1608
- buffer (atributo de io.TextIOBase), 619
- buffer (atributo de unittest.TestResult), 1542
- buffer protocol
 - binary sequence types, 54
 - str (built-in class), 44
- buffer size, I/O, 19
- buffer_info() (método de array.array), 244
- buffer_size (atributo de xml.parsers.expat.xmlparser), 1191
- buffer_text (atributo de xml.parsers.expat.xmlparser), 1191
- buffer_updated() (método de asyncio.BufferedProtocol), 928
- buffer_used (atributo de xml.parsers.expat.xmlparser), 1191
- BufferedIOBase (clase en io), 615
- BufferedProtocol (clase en asyncio), 926
- BufferedRandom (clase en io), 618
- BufferedReader (clase en io), 617
- BufferedRWPair (clase en io), 618
- BufferedWriter (clase en io), 618
- BufferError, 90

- BufferingHandler (*clase en logging.handlers*), 701
 BufferTooShort, 791
 bufsize() (*método de ossaudio-dev.oss_audio_device*), 1364
 BUILD_CONST_KEY_MAP (*opcode*), 1850
 BUILD_LIST (*opcode*), 1850
 BUILD_LIST_UNPACK (*opcode*), 1851
 BUILD_MAP (*opcode*), 1850
 BUILD_MAP_UNPACK (*opcode*), 1851
 BUILD_MAP_UNPACK_WITH_CALL (*opcode*), 1851
 build_opener() (*en el módulo urllib.request*), 1221
 BUILD_SET (*opcode*), 1850
 BUILD_SET_UNPACK (*opcode*), 1851
 BUILD_SLICE (*opcode*), 1854
 BUILD_STRING (*opcode*), 1850
 BUILD_TUPLE (*opcode*), 1850
 BUILD_TUPLE_UNPACK (*opcode*), 1850
 BUILD_TUPLE_UNPACK_WITH_CALL (*opcode*), 1850
 built-in types, 29
 builtin_module_names (*en el módulo sys*), 1692
 BuiltinFunctionType (*en el módulo types*), 255
 BuiltinImporter (*clase en importlib.machinery*), 1800
 BuiltinMethodType (*en el módulo types*), 255
 builtins (*módulo*), 1714
 buscador, 1939
 buscador basado en ruta, 1944
 buscador de entradas de ruta, 1944
 ButtonBox (*clase en tkinter.tix*), 1460
 bye() (*en el módulo turtle*), 1412
 byref() (*en el módulo ctypes*), 760
 bytearray
 formatting, 66
 interpolation, 66
 methods, 57
 objeto, 40, 54, 56
 bytearray (*clase incorporada*), 56
 bytecode, 1937
 byte-code
 file, 1836, 1928
 Bytecode (*clase en dis*), 1842
 BYTECODE_SUFFIXES (*en el módulo importlib.machinery*), 1799
 Bytecode.codeobj (*en el módulo dis*), 1842
 Bytecode.first_line (*en el módulo dis*), 1842
 byteorder (*en el módulo sys*), 1692
 bytes
 formatting, 66
 interpolation, 66
 methods, 57
 objeto, 54
 str (*built-in class*), 44
 bytes (*atributo de uuid.UUID*), 1292
 bytes (*clase incorporada*), 54
 bytes_le (*atributo de uuid.UUID*), 1292
 BytesFeedParser (*clase en email.parser*), 1047
 BytesGenerator (*clase en email.generator*), 1050
 BytesHeaderParser (*clase en email.parser*), 1048
 BytesIO (*clase en io*), 617
 BytesParser (*clase en email.parser*), 1048
 ByteString (*clase en collections.abc*), 235
 ByteString (*clase en typing*), 1486
 byteswap() (*en el módulo audioop*), 1348
 byteswap() (*método de array.array*), 245
 BytesWarning, 97
 bz2 (*módulo*), 476
 BZ2Compressor (*clase en bz2*), 477
 BZ2Decompressor (*clase en bz2*), 477
 BZ2File (*clase en bz2*), 476
- ## C
- c
 trace command line option, 1660
 unittest command line option, 1522
 zipapp command line option, 1684
 C
 language, 31, 32
 structures, 153
 -C
 trace command line option, 1660
 -c <tarfile> <source1> ... <sourceN>
 tarfile command line option, 507
 -c <zipfile> <source1> ... <sourceN>
 zipfile command line option, 494
 C14NWriterTarget (*clase en xml.etree.ElementTree*), 1158
 c_bool (*clase en ctypes*), 765
 C_BUILTIN (*en el módulo imp*), 1931
 c_byte (*clase en ctypes*), 764
 c_char (*clase en ctypes*), 764
 c_char_p (*clase en ctypes*), 764
 c_contiguous (*atributo de memoryview*), 75
 c_double (*clase en ctypes*), 764
 C_EXTENSION (*en el módulo imp*), 1931
 c_float (*clase en ctypes*), 764
 c_int (*clase en ctypes*), 764
 c_int8 (*clase en ctypes*), 764
 c_int16 (*clase en ctypes*), 764
 c_int32 (*clase en ctypes*), 764
 c_int64 (*clase en ctypes*), 764
 c_long (*clase en ctypes*), 764
 c_longdouble (*clase en ctypes*), 764
 c_longlong (*clase en ctypes*), 764
 c_short (*clase en ctypes*), 764
 c_size_t (*clase en ctypes*), 764
 c_ssize_t (*clase en ctypes*), 765
 c_ubyte (*clase en ctypes*), 765
 c_uint (*clase en ctypes*), 765
 c_uint8 (*clase en ctypes*), 765
 c_uint16 (*clase en ctypes*), 765
 c_uint32 (*clase en ctypes*), 765
 c_uint64 (*clase en ctypes*), 765
 c_ulong (*clase en ctypes*), 765
 c_ulonglong (*clase en ctypes*), 765

- `c_ushort` (clase en `ctypes`), 765
- `c_void_p` (clase en `ctypes`), 765
- `c_wchar` (clase en `ctypes`), 765
- `c_wchar_p` (clase en `ctypes`), 765
- `CAB` (clase en `msilib`), 1864
- `cache_from_source()` (en el módulo `imp`), 1930
- `cache_from_source()` (en el módulo `importlib.util`), 1803
- `cached` (atributo de `importlib.machinery.ModuleSpec`), 1803
- `cached_property()` (en el módulo `functools`), 357
- `CacheFTPHandler` (clase en `urllib.request`), 1224
- cadena con triple comilla, 1946
- `calcobjsize()` (en el módulo `test.support`), 1620
- `calcsizex()` (en el módulo `struct`), 154
- `calcobjsize()` (en el módulo `test.support`), 1620
- `Calendar` (clase en `calendar`), 213
- `calendar` (módulo), 212
- `calendar()` (en el módulo `calendar`), 216
- `call()` (en el módulo `subprocess`), 848
- `call()` (en el módulo `unittest.mock`), 1579
- `call_args` (atributo de `unittest.mock.Mock`), 1555
- `call_args_list` (atributo de `unittest.mock.Mock`), 1556
- `call_at()` (método de `asyncio.loop`), 900
- `call_count` (atributo de `unittest.mock.Mock`), 1554
- `call_exception_handler()` (método de `asyncio.loop`), 911
- `CALL_FINALLY` (opcode), 1852
- `CALL_FUNCTION` (opcode), 1853
- `CALL_FUNCTION_EX` (opcode), 1853
- `CALL_FUNCTION_KW` (opcode), 1853
- `call_later()` (método de `asyncio.loop`), 900
- `call_list()` (método de `unittest.mock.call`), 1579
- `CALL_METHOD` (opcode), 1853
- `call_soon()` (método de `asyncio.loop`), 900
- `call_soon_threadsafe()` (método de `asyncio.loop`), 900
- `call_tracing()` (en el módulo `sys`), 1692
- `Callable` (clase en `collections.abc`), 234
- `Callable` (en el módulo `typing`), 1494
- `callable()` (función incorporada), 7
- `CallableProxyType` (en el módulo `weakref`), 249
- `callback`, 1937
- `callback` (atributo de `optparse.Option`), 1914
- `callback()` (método de `contextlib.ExitStack`), 1735
- `callback_args` (atributo de `optparse.Option`), 1914
- `callback_kwargs` (atributo de `optparse.Option`), 1914
- `callbacks` (en el módulo `gc`), 1757
- `called` (atributo de `unittest.mock.Mock`), 1554
- `CalledProcessError`, 838
- `CAN_BCM` (en el módulo `socket`), 954
- `can_change_color()` (en el módulo `curses`), 706
- `can_fetch()` (método de `urllib.robotparser.RobotFileParser`), 1247
- `CAN_ISOTP` (en el módulo `socket`), 955
- `CAN_RAW_FD_FRAMES` (en el módulo `socket`), 955
- `can_symlink()` (en el módulo `test.support`), 1620
- `can_write_eof()` (método de `asyncio.StreamWriter`), 881
- `can_write_eof()` (método de `asyncio.WriteTransport`), 924
- `can_xattr()` (en el módulo `test.support`), 1620
- `cancel()` (método de `asyncio.Future`), 920
- `cancel()` (método de `asyncio.Handle`), 913
- `cancel()` (método de `asyncio.Task`), 876
- `cancel()` (método de `concurrent.futures.Future`), 834
- `cancel()` (método de `sched.scheduler`), 855
- `cancel()` (método de `threading.Timer`), 780
- `cancel_dump_traceback_later()` (en el módulo `faulthandler`), 1639
- `cancel_join_thread()` (método de `multiprocessing.Queue`), 793
- `cancelled()` (método de `asyncio.Future`), 919
- `cancelled()` (método de `asyncio.Handle`), 913
- `cancelled()` (método de `asyncio.Task`), 876
- `cancelled()` (método de `concurrent.futures.Future`), 834
- `CancelledError`, 836, 896
- `CannotSendHeader`, 1252
- `CannotSendRequest`, 1252
- `canonic()` (método de `bdb.Bdb`), 1634
- `canonical()` (método de `decimal.Context`), 311
- `canonical()` (método de `decimal.Decimal`), 304
- `canonicalize()` (en el módulo `xml.etree.ElementTree`), 1149
- `capa()` (método de `poplib.POP3`), 1263
- `capitalize()` (método de `bytearray`), 62
- `capitalize()` (método de `bytes`), 62
- `capitalize()` (método de `str`), 45
- `captured_stderr()` (en el módulo `test.support`), 1619
- `captured_stdin()` (en el módulo `test.support`), 1619
- `captured_stdout()` (en el módulo `test.support`), 1619
- `captureWarnings()` (en el módulo `logging`), 681
- `capwords()` (en el módulo `string`), 110
- cargador, 1942
- `casefold()` (método de `str`), 45
- `cast()` (en el módulo `ctypes`), 760
- `cast()` (en el módulo `typing`), 1491
- `cast()` (método de `memoryview`), 72
- `cat()` (en el módulo `nis`), 1897
- `--catch`
 - `unittest` command line option, 1522
- `catch_threading_exception()` (en el módulo `test.support`), 1623
- `catch_unraisable_exception()` (en el módulo `test.support`), 1624
- `catch_warnings` (clase en `warnings`), 1721
- `category()` (en el módulo `unicodedata`), 144
- `cbreak()` (en el módulo `curses`), 706
- `ccc()` (método de `ftplib.FTP_TLS`), 1262
- C-contiguous, 1937

- `cdf()` (método de *statistics.NormalDist*), 340
`CDLL` (clase en *ctypes*), 754
`ceil()` (en el módulo *math*), 289
`ceil()` (in module *math*), 32
`CellType` (en el módulo *types*), 255
`center()` (método de *bytearray*), 59
`center()` (método de *bytes*), 59
`center()` (método de *str*), 45
`CERT_NONE` (en el módulo *ssl*), 979
`CERT_OPTIONAL` (en el módulo *ssl*), 979
`CERT_REQUIRED` (en el módulo *ssl*), 980
`cert_store_stats()` (método de *ssl.SSLContext*), 990
`cert_time_to_seconds()` (en el módulo *ssl*), 977
`CertificateError`, 976
`certificates`, 997
`CFUNCTYPE()` (en el módulo *ctypes*), 758
`CGI`
 debugging, 1207
 exceptions, 1208
 protocol, 1202
 security, 1206
 tracebacks, 1208
`cgi` (módulo), 1202
`cgi_directories` (atributo de *http.server.CGIHTTPRequestHandler*), 1307
`CGIHandler` (clase en *wsgiref.handlers*), 1215
`CGIHTTPRequestHandler` (clase en *http.server*), 1307
`cgitb` (módulo), 1208
`CGIXMLRPCRequestHandler` (clase en *xmlrpc.server*), 1328
`chain()` (en el módulo *itertools*), 346
`chaining`
 comparisons, 30
`ChainMap` (clase en *collections*), 217
`ChainMap` (clase en *typing*), 1488
`change_cwd()` (en el módulo *test.support*), 1619
`CHANNEL_BINDING_TYPES` (en el módulo *ssl*), 984
`channel_class` (atributo de *smtpd.SMTPServer*), 1286
`channels()` (método de *ossaudio-dev.oss_audio_device*), 1363
`CHAR_MAX` (en el módulo *locale*), 1381
`character`, 144
`CharacterDataHandler()` (método de *xml.parsers.expat.xmlparser*), 1193
`characters()` (método de *xml.sax.handler.ContentHandler*), 1182
`characters_written` (atributo de *BlockingIOError*), 95
`Charset` (clase en *email.charset*), 1089
`charset()` (método de *gettext.NullTranslations*), 1371
`chdir()` (en el módulo *os*), 576
`check` (atributo de *lzma.LZMADecompressor*), 482
`check()` (en el módulo *tabnanny*), 1834
`check()` (método de *imaplib.IMAP4*), 1267
`check_all__()` (en el módulo *test.support*), 1625
`check_call()` (en el módulo *subprocess*), 848
`check_free_after_iterating()` (en el módulo *test.support*), 1625
`check_hostname` (atributo de *ssl.SSLContext*), 995
`check_impl_detail()` (en el módulo *test.support*), 1617
`check_no_resource_warning()` (en el módulo *test.support*), 1618
`check_output()` (en el módulo *subprocess*), 849
`check_output()` (método de *doctest.OutputChecker*), 1515
`check_returncode()` (método de *subprocess.CompletedProcess*), 838
`check_syntax_error()` (en el módulo *test.support*), 1622
`check_syntax_warning()` (en el módulo *test.support*), 1622
`check_unused_args()` (método de *string.Formatter*), 101
`check_warnings()` (en el módulo *test.support*), 1617
`checkbox()` (método de *msilib.Dialog*), 1866
`checkcache()` (en el módulo *linecache*), 410
`CHECKED_HASH` (atributo de *py_compile.PycInvalidationMode*), 1837
`checkfuncname()` (en el módulo *bdb*), 1637
`CheckList` (clase en *tkinter.tix*), 1461
`checksizeof()` (en el módulo *test.support*), 1620
`checksum`
 Cyclic Redundancy Check, 470
`chflags()` (en el módulo *os*), 577
`chgat()` (método de *curses.window*), 713
`childNodes` (atributo de *xml.dom.Node*), 1164
`ChildProcessError`, 95
`children` (atributo de *pyclbr.Class*), 1836
`children` (atributo de *pyclbr.Function*), 1835
`chmod()` (en el módulo *os*), 577
`chmod()` (método de *pathlib.Path*), 383
`choice()` (en el módulo *random*), 328
`choice()` (en el módulo *secrets*), 555
`choices` (atributo de *optparse.Option*), 1914
`choices()` (en el módulo *random*), 328
`chown()` (en el módulo *os*), 578
`chown()` (en el módulo *shutil*), 415
`chr()` (función incorporada), 7
`chroot()` (en el módulo *os*), 578
`Chunk` (clase en *chunk*), 1358
`chunk` (módulo), 1358
`cipher`
 DES, 1884
`cipher()` (método de *ssl.SSLSocket*), 987
`circle()` (en el módulo *turtle*), 1392
`CIRCUMFLEX` (en el módulo *token*), 1828
`CIRCUMFLEXEQUAL` (en el módulo *token*), 1828
`Clamped` (clase en *decimal*), 315
`clase`, 1937
`clase base abstracta`, 1935
`clase de nuevo estilo`, 1943

- Class (*clase en symtable*), 1825
- Class browser, 1464
- classmethod() (*función incorporada*), 7
- ClassMethodDescriptorType (*en el módulo types*), 255
- ClassVar (*en el módulo typing*), 1495
- CLD_CONTINUED (*en el módulo os*), 604
- CLD_DUMPED (*en el módulo os*), 604
- CLD_EXITED (*en el módulo os*), 604
- CLD_TRAPPED (*en el módulo os*), 604
- clean() (*método de mailbox.Maildir*), 1110
- cleandoc() (*en el módulo inspect*), 1762
- CleanImport (*clase en test.support*), 1626
- clear (*pdb command*), 1643
- Clear Breakpoint, 1467
- clear() (*en el módulo turtle*), 1406
- clear() (*método de asyncio.Event*), 886
- clear() (*método de collections.deque*), 223
- clear() (*método de curses.window*), 713
- clear() (*método de dict*), 79
- clear() (*método de email.message.EmailMessage*), 1045
- clear() (*método de frozenset*), 77
- clear() (*método de http.cookiejar.CookieJar*), 1313
- clear() (*método de mailbox.Mailbox*), 1109
- clear() (*método de threading.Event*), 779
- clear() (*método de xml.etree.ElementTree.Element*), 1154
- clear() (*sequence method*), 40
- clear_all_breaks() (*método de bdb.Bdb*), 1636
- clear_all_file_breaks() (*método de bdb.Bdb*), 1636
- clear_bppynumber() (*método de bdb.Bdb*), 1636
- clear_break() (*método de bdb.Bdb*), 1636
- clear_cache() (*en el módulo filecmp*), 402
- clear_content() (*método de email.message.EmailMessage*), 1045
- clear_flags() (*método de decimal.Context*), 310
- clear_frames() (*en el módulo traceback*), 1749
- clear_history() (*en el módulo readline*), 148
- clear_session_cookies() (*método de http.cookiejar.CookieJar*), 1314
- clear_traces() (*en el módulo tracemalloc*), 1666
- clear_traps() (*método de decimal.Context*), 310
- clearcache() (*en el módulo linecache*), 410
- ClearData() (*método de msilib.Record*), 1864
- clearok() (*método de curses.window*), 713
- clearscreen() (*en el módulo turtle*), 1406
- clearstamp() (*en el módulo turtle*), 1393
- clearstamps() (*en el módulo turtle*), 1393
- Client() (*en el módulo multiprocessing.connection*), 812
- client_address (*atributo de http.server.BaseHTTPRequestHandler*), 1303
- CLOCK_BOOTTIME (*en el módulo time*), 630
- clock_getres() (*en el módulo time*), 624
- clock_gettime() (*en el módulo time*), 624
- clock_gettime_ns() (*en el módulo time*), 624
- CLOCK_HIGHRES (*en el módulo time*), 630
- CLOCK_MONOTONIC (*en el módulo time*), 630
- CLOCK_MONOTONIC_RAW (*en el módulo time*), 630
- CLOCK_PROCESS_CPUTIME_ID (*en el módulo time*), 630
- CLOCK_PROF (*en el módulo time*), 630
- CLOCK_REALTIME (*en el módulo time*), 631
- clock_settime() (*en el módulo time*), 624
- clock_settime_ns() (*en el módulo time*), 624
- CLOCK_THREAD_CPUTIME_ID (*en el módulo time*), 630
- CLOCK_UPTIME (*en el módulo time*), 630
- CLOCK_UPTIME_RAW (*en el módulo time*), 630
- clone() (*en el módulo turtle*), 1404
- clone() (*método de email.generator.BytesGenerator*), 1051
- clone() (*método de email.generator.Generator*), 1052
- clone() (*método de email.policy.Policy*), 1055
- clone() (*método de pipes.Template*), 1892
- cloneNode() (*método de xml.dom.Node*), 1165
- close() (*en el módulo fileinput*), 395
- close() (*en el módulo os*), 566
- close() (*en el módulo socket*), 958
- close() (*método de aifc.aifc*), 1351
- close() (*método de asyncio.AbstractChildWatcher*), 937
- close() (*método de asyncio.BaseTransport*), 923
- close() (*método de asyncio.loop*), 899
- close() (*método de asyncio.Server*), 914
- close() (*método de asyncio.StreamWriter*), 881
- close() (*método de asyncio.SubprocessTransport*), 926
- close() (*método de asyncore.dispatcher*), 1019
- close() (*método de chunk.Chunk*), 1358
- close() (*método de contextlib.ExitStack*), 1735
- close() (*método de dbm.dumb.dumbdbm*), 446
- close() (*método de dbm.gnu.gdbm*), 444
- close() (*método de dbm.ndbm.ndbm*), 445
- close() (*método de email.parser.BytesFeedParser*), 1047
- close() (*método de ftplib.FTP*), 1262
- close() (*método de html.parser.HTMLParser*), 1137
- close() (*método de http.client.HTTPConnection*), 1254
- close() (*método de imaplib.IMAP4*), 1267
- close() (*método de io.IOBase*), 613
- close() (*método de logging.FileHandler*), 693
- close() (*método de logging.Handler*), 670
- close() (*método de logging.handlers.MemoryHandler*), 702
- close() (*método de logging.handlers.NTEventLogHandler*), 700
- close() (*método de logging.handlers.SocketHandler*), 697
- close() (*método de logging.handlers.SysLogHandler*), 698
- close() (*método de mailbox.Mailbox*), 1109

- close() (método de `mailbox.Maildir`), 1111
- close() (método de `mailbox.MH`), 1113
- close() (método de `mmap.mmap`), 1033
- Close() (método de `msilib.Database`), 1862
- Close() (método de `msilib.View`), 1863
- close() (método de `multiprocessing.connection.Connection`), 796
- close() (método de `multiprocessing.connection.Listener`), 812
- close() (método de `multiprocessing.pool.Pool`), 810
- close() (método de `multiprocessing.Process`), 790
- close() (método de `multiprocessing.Queue`), 793
- close() (método de `multiprocessing.shared_memory.SharedMemory`), 826
- close() (método de `ossaudiodev.oss_audio_device`), 1362
- close() (método de `ossaudiodev.oss_mixer_device`), 1365
- close() (método de `os.scandir`), 584
- close() (método de `select.devpoll`), 1009
- close() (método de `select.epoll`), 1010
- close() (método de `select.kqueue`), 1012
- close() (método de `selectors.BaseSelector`), 1015
- close() (método de `shelve.Shelf`), 439
- close() (método de `socket.socket`), 963
- close() (método de `sqlite3.Connection`), 450
- close() (método de `sqlite3.Cursor`), 458
- close() (método de `sunau.AU_read`), 1354
- close() (método de `sunau.AU_write`), 1355
- close() (método de `tarfile.TarFile`), 501
- close() (método de `telnetlib.Telnet`), 1290
- close() (método de `urllib.request.BaseHandler`), 1227
- close() (método de `wave.Wave_read`), 1356
- close() (método de `wave.Wave_write`), 1357
- Close() (método de `winreg.PyHKEY`), 1877
- close() (método de `xml.etree.ElementTree.TreeBuilder`), 1157
- close() (método de `xml.etree.ElementTree.XMLParser`), 1158
- close() (método de `xml.etree.ElementTree.XMLPullParser`), 1160
- close() (método de `xml.sax.xmlreader.IncrementalParser`), 1187
- close() (método de `zipfile.ZipFile`), 487
- close_connection (atributo de `http.server.BaseHTTPRequestHandler`), 1303
- close_when_done() (método de `asyncio.async_chat`), 1022
- closed (atributo de `http.client.HTTPResponse`), 1255
- closed (atributo de `io.IOBase`), 613
- closed (atributo de `mmap.mmap`), 1033
- closed (atributo de `ossaudiodev.oss_audio_device`), 1364
- closed (atributo de `select.devpoll`), 1009
- closed (atributo de `select.epoll`), 1010
- closed (atributo de `select.kqueue`), 1012
- CloseKey() (en el módulo `winreg`), 1869
- closelog() (en el módulo `syslog`), 1898
- closerange() (en el módulo `os`), 566
- closing() (en el módulo `contextlib`), 1731
- clrtoebot() (método de `curses.window`), 713
- clrtoeol() (método de `curses.window`), 713
- cmath (módulo), 295
- cmd
- módulo, 1640
- cmd (atributo de `subprocess.CalledProcessError`), 838
- cmd (atributo de `subprocess.TimeoutExpired`), 838
- Cmd (clase en `cmd`), 1418
- cmd (módulo), 1418
- cmdloop() (método de `cmd.Cmd`), 1418
- cmdqueue (atributo de `cmd.Cmd`), 1420
- cmp() (en el módulo `filecmp`), 401
- cmp_op (en el módulo `dis`), 1854
- cmp_to_key() (en el módulo `functools`), 358
- cmpfiles() (en el módulo `filecmp`), 401
- MSG_LEN() (en el módulo `socket`), 961
- MSG_SPACE() (en el módulo `socket`), 961
- CO_ASYNC_GENERATOR (en el módulo `inspect`), 1772
- CO_COROUTINE (en el módulo `inspect`), 1772
- CO_GENERATOR (en el módulo `inspect`), 1772
- CO_ITERABLE_COROUTINE (en el módulo `inspect`), 1772
- CO_NESTED (en el módulo `inspect`), 1772
- CO_NEWLOCALS (en el módulo `inspect`), 1772
- CO_NOFREE (en el módulo `inspect`), 1772
- CO_OPTIMIZED (en el módulo `inspect`), 1772
- CO_VARARGS (en el módulo `inspect`), 1772
- CO_VARKEYWORDS (en el módulo `inspect`), 1772
- code (atributo de `SystemExit`), 94
- code (atributo de `urllib.error.HTTPError`), 1247
- code (atributo de `xml.etree.ElementTree.ParseError`), 1161
- code (atributo de `xml.parsers.expat.ExpatError`), 1194
- code (módulo), 1777
- code object, 84, 441
- code_info() (en el módulo `dis`), 1843
- CodecInfo (clase en `codecs`), 159
- Codecs, 158
- decode, 158
- encode, 158
- codecs (módulo), 158
- coded_value (atributo de `http.cookies.Morsel`), 1310
- codeop (módulo), 1779
- codepoint2name (en el módulo `html.entities`), 1140
- codes (en el módulo `xml.parsers.expat.errors`), 1196
- CODESET (en el módulo `locale`), 1378
- CodeType (clase en `types`), 255
- codificación de texto, 1946
- coerción, 1937
- col_offset (atributo de `ast.AST`), 1818
- collapse_addresses() (en el módulo `ipaddress`), 1345
- collapse_rfc2231_value() (en el módulo `email.utils`), 1094

- `collect()` (en el módulo *gc*), 1755
- `collect_incoming_data()` (método de *asyncio.async_chat*), 1022
- `Collection` (clase en *collections.abc*), 235
- `Collection` (clase en *typing*), 1486
- `collections` (módulo), 217
- `collections.abc` (módulo), 233
- `colno` (atributo de *json.JSONDecodeError*), 1102
- `colno` (atributo de *re.error*), 120
- `COLON` (en el módulo *token*), 1827
- `COLONEQUAL` (en el módulo *token*), 1829
- `color()` (en el módulo *turtle*), 1398
- `color_content()` (en el módulo *curses*), 706
- `color_pair()` (en el módulo *curses*), 706
- `colormode()` (en el módulo *turtle*), 1410
- `coloursys` (módulo), 1359
- `COLS`, 711
- `column()` (método de *tkinter.ttk.Treeview*), 1452
- `COLUMNS`, 711
- `columns` (atributo de *os.terminal_size*), 575
- `comb()` (en el módulo *math*), 289
- `combinations()` (en el módulo *itertools*), 346
- `combinations_with_replacement()` (en el módulo *itertools*), 347
- `combine()` (método de clase de *datetime.datetime*), 189
- `combining()` (en el módulo *unicodedata*), 144
- `ComboBox` (clase en *tkinter.tix*), 1460
- `Combobox` (clase en *tkinter.ttk*), 1444
- `COMMA` (en el módulo *token*), 1827
- `command` (atributo de *http.server.BaseHTTPRequestHandler*), 1303
- `CommandCompiler` (clase en *codeop*), 1780
- `commands` (*pdb* *command*), 1643
- `comment` (atributo de *http.cookiejar.Cookie*), 1318
- `comment` (atributo de *zipfile.ZipFile*), 490
- `comment` (atributo de *zipfile.ZipInfo*), 492
- `COMMENT` (en el módulo *token*), 1829
- `Comment()` (en el módulo *xml.etree.ElementTree*), 1150
- `comment()` (método de *xml.etree.ElementTree.TreeBuilder*), 1158
- `comment_url` (atributo de *http.cookiejar.Cookie*), 1318
- `commenters` (atributo de *shlex.shlex*), 1425
- `CommentHandler()` (método de *xml.parsers.expat.xmlparser*), 1193
- `commit()` (método de *msilib.CAB*), 1864
- `Commit()` (método de *msilib.Database*), 1862
- `commit()` (método de *sqlite3.Connection*), 450
- `common` (atributo de *filecmp.dircmp*), 402
- `Common Gateway Interface`, 1202
- `common_dirs` (atributo de *filecmp.dircmp*), 402
- `common_files` (atributo de *filecmp.dircmp*), 402
- `common_funny` (atributo de *filecmp.dircmp*), 402
- `common_types` (en el módulo *mimetypes*), 1125
- `commonpath()` (en el módulo *os.path*), 390
- `commonprefix()` (en el módulo *os.path*), 390
- `communicate()` (método de *asyncio.subprocess.Process*), 891
- `communicate()` (método de *subprocess.Popen*), 844
- `compare()` (método de *decimal.Context*), 311
- `compare()` (método de *decimal.Decimal*), 304
- `compare()` (método de *difflib.Differ*), 138
- `compare_digest()` (en el módulo *hmac*), 554
- `compare_digest()` (en el módulo *secrets*), 556
- `compare_networks()` (método de *ipaddress.IPv4Network*), 1341
- `compare_networks()` (método de *ipaddress.IPv6Network*), 1342
- `COMPARE_OP` (*opcode*), 1851
- `compare_signal()` (método de *decimal.Context*), 311
- `compare_signal()` (método de *decimal.Decimal*), 304
- `compare_to()` (método de *tracemalloc.Snapshot*), 1668
- `compare_total()` (método de *decimal.Context*), 311
- `compare_total()` (método de *decimal.Decimal*), 304
- `compare_total_mag()` (método de *decimal.Context*), 311
- `compare_total_mag()` (método de *decimal.Decimal*), 304
- `comparing` objects, 30
- `comparison` operator, 30
- `COMPARISON_FLAGS` (en el módulo *doctest*), 1505
- `comparisons` chaining, 30
- `Compat32` (clase en *email.policy*), 1059
- `compat32` (en el módulo *email.policy*), 1060
- `compile` función incorporada, 84, 255, 1816
- `Compile` (clase en *codeop*), 1780
- `compile()` (en el módulo *py_compile*), 1836
- `compile()` (en el módulo *re*), 115
- `compile()` (función incorporada), 7
- `compile()` (método de *parser.ST*), 1816
- `compile_command()` (en el módulo *code*), 1778
- `compile_command()` (en el módulo *codeop*), 1779
- `compile_dir()` (en el módulo *compileall*), 1839
- `compile_file()` (en el módulo *compileall*), 1840
- `compile_path()` (en el módulo *compileall*), 1840
- `compileall` (módulo), 1838
- `compileall` command line option
 - `-b`, 1838
 - `-d` *destdir*, 1838
 - `directory ...`, 1838
 - `-f`, 1838
 - `file ...`, 1838
 - `-i` *list*, 1838
 - `--invalidation-mode` [*timestamp|checked-hash|unchecked-hash*], 1839

- j N, 1839
- l, 1838
- q, 1838
- r, 1838
- x regex, 1838
- compilest() (en el módulo parser), 1815
- complete() (método de rcompleter.Completer), 152
- complete_statement() (en el módulo sqlite3), 449
- completedefault() (método de cmd.Cmd), 1419
- CompletedProcess (clase en subprocess), 837
- complex
 - función incorporada, 31
- Complex (clase en numbers), 285
- complex (clase incorporada), 8
- complex number
 - literals, 31
 - objeto, 31
- comprensión de listas, 1942
- compress
 - zipapp command line option, 1684
- compress() (en el módulo bz2), 478
- compress() (en el módulo gzip), 474
- compress() (en el módulo itertools), 347
- compress() (en el módulo lzma), 483
- compress() (en el módulo zlib), 469
- compress() (método de bz2.BZ2Compressor), 477
- compress() (método de lzma.LZMACompressor), 482
- compress() (método de zlib.Compress), 471
- compress_size (atributo de zipfile.ZipInfo), 493
- compress_type (atributo de zipfile.ZipInfo), 492
- compressed (atributo de ipaddress.IPv4Address), 1334
- compressed (atributo de ipaddress.IPv4Network), 1339
- compressed (atributo de ipaddress.IPv6Address), 1336
- compressed (atributo de ipaddress.IPv6Network), 1342
- compression() (método de ssl.SSLSocket), 988
- CompressionError, 496
- compressobj() (en el módulo zlib), 470
- COMSPEC, 603, 841
- concat() (en el módulo operator), 367
- concatenation
 - operation, 38
- concurrent.futures (módulo), 830
- Condition (clase en asyncio), 887
- Condition (clase en multiprocessing), 797
- Condition (clase en threading), 776
- condition (pdb command), 1643
- condition() (método de msilib.Control), 1866
- Condition() (método de multiprocessing.managers.SyncManager), 804
- ConfigParser (clase en configparser), 530
- configparser (módulo), 518
- configuration
 - file, 518
 - file, debugger, 1642
 - file, path, 1774
- configuration information, 1710
- configure() (método de tkinter.ttk.Style), 1455
- configure_mock() (método de unittest.mock.Mock), 1553
- confstr() (en el módulo os), 607
- confstr_names (en el módulo os), 607
- conjugate() (complex number method), 31
- conjugate() (método de decimal.Decimal), 304
- conjugate() (método de numbers.Complex), 285
- conn (atributo de smtpd.SMTPChannel), 1287
- connect() (en el módulo sqlite3), 448
- connect() (método de asyncore.dispatcher), 1019
- connect() (método de ftplib.FTP), 1259
- connect() (método de http.client.HTTPConnection), 1254
- connect() (método de multiprocessing.managers.BaseManager), 803
- connect() (método de smtplib.SMTP), 1280
- connect() (método de socket.socket), 963
- connect_accepted_socket() (método de asyncio.loop), 905
- connect_ex() (método de socket.socket), 963
- connect_read_pipe() (método de asyncio.loop), 909
- connect_write_pipe() (método de asyncio.loop), 909
- connection (atributo de sqlite3.Cursor), 459
- Connection (clase en multiprocessing.connection), 796
- Connection (clase en sqlite3), 450
- connection_lost() (método de asyncio.BaseProtocol), 927
- connection_made() (método de asyncio.BaseProtocol), 927
- ConnectionAbortedError, 95
- ConnectionError, 95
- ConnectionRefusedError, 95
- ConnectionResetError, 95
- ConnectRegistry() (en el módulo winreg), 1869
- const (atributo de optparse.Option), 1914
- constructor() (en el módulo copyreg), 437
- consumed (atributo de asyncio.LimitOverrunError), 896
- contador de referencias, 1945
- container
 - iteration over, 37
- Container (clase en collections.abc), 234
- Container (clase en typing), 1486
- contains() (en el módulo operator), 367
- content type
 - MIME, 1124
- content_disposition (atributo de email.headerregistry.ContentDispositionHeader), 1064
- content_manager (atributo de email.policy.EmailPolicy), 1058

`content_type` (atributo de `email.headerregistry.ContentTypeHeader`), 1064
`ContentDispositionHeader` (clase en `email.headerregistry`), 1064
`ContentHandler` (clase en `xml.sax.handler`), 1179
`ContentManager` (clase en `email.contentmanager`), 1067
`contents` (atributo de `ctypes._Pointer`), 768
`contents()` (en el módulo `importlib.resources`), 1799
`contents()` (método de `importlib.abc.ResourceReader`), 1795
`ContentTooShortError`, 1247
`ContentTransferEncoding` (clase en `email.headerregistry`), 1064
`ContentTypeHeader` (clase en `email.headerregistry`), 1064
`context` (atributo de `ssl.SSLSocket`), 989
`Context` (clase en `contextvars`), 860
`Context` (clase en `decimal`), 310
`context management protocol`, 82
`context manager`, 82
`context_diff()` (en el módulo `difflib`), 131
`ContextDecorator` (clase en `contextlib`), 1733
`contextlib` (módulo), 1729
`ContextManager` (clase en `typing`), 1487
`contextmanager()` (en el módulo `contextlib`), 1729
`ContextVar` (clase en `contextvars`), 859
`contextvars` (módulo), 858
`contiguo`, 1937
`contiguous` (atributo de `memoryview`), 75
`continue` (`pdb` command), 1644
`Control` (clase en `msilib`), 1865
`Control` (clase en `tkinter.tix`), 1460
`control()` (método de `msilib.Dialog`), 1866
`control()` (método de `select.kqueue`), 1012
`controlnames` (en el módulo `curses.ascii`), 726
`controls()` (método de `ossaudio-dev.oss_mixer_device`), 1365
`ConversionError`, 539
`conversions`
 numeric, 32
`convert_arg_line_to_args()` (método de `argparse.ArgumentParser`), 661
`convert_field()` (método de `string.Formatter`), 101
`Cookie` (clase en `http.cookiejar`), 1312
`CookieError`, 1308
`cookiejar` (atributo de `urllib.request.HTTPCookieProcessor`), 1229
`CookieJar` (clase en `http.cookiejar`), 1312
`CookiePolicy` (clase en `http.cookiejar`), 1312
`Coordinated Universal Time`, 622
`copy`
 módulo, 437
 protocol, 429
`Copy`, 1467
`copy` (módulo), 258
`copy()` (en el módulo `copy`), 258
`copy()` (en el módulo `multiprocessing.sharedctypes`), 801
`copy()` (en el módulo `shutil`), 412
`copy()` (método de `collections.deque`), 223
`copy()` (método de `contextvars.Context`), 861
`copy()` (método de `decimal.Context`), 310
`copy()` (método de `dict`), 79
`copy()` (método de `frozenset`), 76
`copy()` (método de `hashlib.hash`), 545
`copy()` (método de `hmac.HMAC`), 554
`copy()` (método de `http.cookies.Morsel`), 1310
`copy()` (método de `imaplib.IMAP4`), 1267
`copy()` (método de `pipes.Template`), 1892
`copy()` (método de `types.MappingProxyType`), 257
`copy()` (método de `zlib.Compress`), 471
`copy()` (método de `zlib.Decompress`), 472
`copy()` (sequence method), 40
`copy2()` (en el módulo `shutil`), 412
`copy_abs()` (método de `decimal.Context`), 311
`copy_abs()` (método de `decimal.Decimal`), 304
`copy_context()` (en el módulo `contextvars`), 860
`copy_decimal()` (método de `decimal.Context`), 310
`copy_file_range()` (en el módulo `os`), 566
`copy_location()` (en el módulo `ast`), 1822
`copy_negate()` (método de `decimal.Context`), 311
`copy_negate()` (método de `decimal.Decimal`), 304
`copy_sign()` (método de `decimal.Context`), 311
`copy_sign()` (método de `decimal.Decimal`), 304
`copyfile()` (en el módulo `shutil`), 411
`copyfileobj()` (en el módulo `shutil`), 411
`copying files`, 410
`copymode()` (en el módulo `shutil`), 411
`copyreg` (módulo), 437
`copyright` (en el módulo `sys`), 1693
`copyright` (variable incorporada), 28
`copysign()` (en el módulo `math`), 289
`copystat()` (en el módulo `shutil`), 411
`copytree()` (en el módulo `shutil`), 413
`Coroutine` (clase en `collections.abc`), 236
`Coroutine` (clase en `typing`), 1487
`coroutine()` (en el módulo `asyncio`), 878
`coroutine()` (en el módulo `types`), 258
`CoroutineType` (en el módulo `types`), 254
`corrutina`, 1937
`cos()` (en el módulo `cmath`), 296
`cos()` (en el módulo `math`), 292
`cosh()` (en el módulo `cmath`), 296
`cosh()` (en el módulo `math`), 293
`--count`
 trace command line option, 1660
`count` (atributo de `tracemalloc.Statistic`), 1669
`count` (atributo de `tracemalloc.StatisticDiff`), 1669
`count()` (en el módulo `itertools`), 348
`count()` (método de `array.array`), 245
`count()` (método de `bytearray`), 57
`count()` (método de `bytes`), 57
`count()` (método de `collections.deque`), 223

- `count()` (método de `multiprocessing.shared_memory.ShareableList`), 828
- `count()` (método de `str`), 45
- `count()` (sequence method), 38
- `count_diff` (atributo de `tracemalloc.StatisticDiff`), 1669
- `Counter` (clase en `collections`), 220
- `Counter` (clase en `typing`), 1488
- `countOf()` (en el módulo `operator`), 367
- `countTestCases()` (método de `unittest.TestCase`), 1535
- `countTestCases()` (método de `unittest.TestSuite`), 1538
- `CoverageResults` (clase en `trace`), 1661
- `--coverdir=<dir>`
trace command line option, 1660
- `cProfile` (módulo), 1649
- CPU time, 625, 628
- `cpu_count()` (en el módulo `multiprocessing`), 794
- `cpu_count()` (en el módulo `os`), 607
- `CPython`, 1938
- `cpython_only()` (en el módulo `test.support`), 1621
- `crawl_delay()` (método de `urllib.robotparser.RobotFileParser`), 1247
- `CRC` (atributo de `zipfile.ZipInfo`), 493
- `crc32()` (en el módulo `binascii`), 1131
- `crc32()` (en el módulo `zlib`), 470
- `crc_hqx()` (en el módulo `binascii`), 1131
- `--create <tarfile> <source1> ... <sourceN>`
tarfile command line option, 507
- `--create <zipfile> <source1> ... <sourceN>`
zipfile command line option, 494
- `create()` (en el módulo `venv`), 1680
- `create()` (método de `imaplib.IMAP4`), 1267
- `create()` (método de `venv.EnvBuilder`), 1678
- `create_aggregate()` (método de `sqlite3.Connection`), 451
- `create_archive()` (en el módulo `zipapp`), 1685
- `create_autospec()` (en el módulo `unittest.mock`), 1581
- `CREATE_BREAKAWAY_FROM_JOB` (en el módulo `subprocess`), 848
- `create_collation()` (método de `sqlite3.Connection`), 452
- `create_configuration()` (método de `venv.EnvBuilder`), 1679
- `create_connection()` (en el módulo `socket`), 957
- `create_connection()` (método de `asyncio.loop`), 902
- `create_datagram_endpoint()` (método de `asyncio.loop`), 903
- `create_decimal()` (método de `decimal.Context`), 310
- `create_decimal_from_float()` (método de `decimal.Context`), 311
- `create_default_context()` (en el módulo `ssl`), 974
- `CREATE_DEFAULT_ERROR_MODE` (en el módulo `subprocess`), 848
- `create_empty_file()` (en el módulo `test.support`), 1617
- `create_function()` (método de `sqlite3.Connection`), 451
- `create_future()` (método de `asyncio.loop`), 901
- `create_module()` (método de `importlib.abc.Loader`), 1793
- `create_module()` (método de `importlib.machinery.ExtensionFileLoader`), 1802
- `CREATE_NEW_CONSOLE` (en el módulo `subprocess`), 847
- `CREATE_NEW_PROCESS_GROUP` (en el módulo `subprocess`), 847
- `CREATE_NO_WINDOW` (en el módulo `subprocess`), 848
- `create_server()` (en el módulo `socket`), 957
- `create_server()` (método de `asyncio.loop`), 904
- `create_socket()` (método de `asyncore.dispatcher`), 1019
- `create_stats()` (método de `profile.Profile`), 1650
- `create_string_buffer()` (en el módulo `ctypes`), 760
- `create_subprocess_exec()` (en el módulo `asyncio`), 890
- `create_subprocess_shell()` (en el módulo `asyncio`), 890
- `create_system` (atributo de `zipfile.ZipInfo`), 492
- `create_task()` (en el módulo `asyncio`), 870
- `create_task()` (método de `asyncio.loop`), 901
- `create_unicode_buffer()` (en el módulo `ctypes`), 760
- `create_unix_connection()` (método de `asyncio.loop`), 904
- `create_unix_server()` (método de `asyncio.loop`), 905
- `create_version` (atributo de `zipfile.ZipInfo`), 492
- `createAttribute()` (método de `xml.dom.Document`), 1166
- `createAttributeNS()` (método de `xml.dom.Document`), 1166
- `createComment()` (método de `xml.dom.Document`), 1166
- `createDocument()` (método de `xml.dom.DOMImplementation`), 1163
- `createDocumentType()` (método de `xml.dom.DOMImplementation`), 1163
- `createElement()` (método de `xml.dom.Document`), 1166
- `createElementNS()` (método de `xml.dom.Document`), 1166
- `createfilehandler()` (método de `tkinter.Widget.tk`), 1439
- `CreateKey()` (en el módulo `winreg`), 1869
- `CreateKeyEx()` (en el módulo `winreg`), 1869
- `createLock()` (método de `logging.Handler`), 669

- `createLock()` (método de `logging.NullHandler`), 693
- `createProcessingInstruction()` (método de `xml.dom.Document`), 1166
- `CreateRecord()` (en el módulo `msilib`), 1861
- `createSocket()` (método de `logging.handlers.SocketHandler`), 697
- `createTextNode()` (método de `xml.dom.Document`), 1166
- `credits` (variable incorporada), 28
- `critical()` (en el módulo `logging`), 678
- `critical()` (método de `logging.Logger`), 668
- `CRNCYSTR` (en el módulo `locale`), 1379
- `cross()` (en el módulo `audioop`), 1348
- `crypt`
módulo, 1882
- `crypt` (módulo), 1884
- `crypt()` (en el módulo `crypt`), 1885
- `crypt(3)`, 1884, 1885
- `cryptography`, 543
- `cssclass_month` (atributo de `calendar.HTMLCalendar`), 215
- `cssclass_month_head` (atributo de `calendar.HTMLCalendar`), 215
- `cssclass_noday` (atributo de `calendar.HTMLCalendar`), 214
- `cssclass_year` (atributo de `calendar.HTMLCalendar`), 215
- `cssclass_year_head` (atributo de `calendar.HTMLCalendar`), 215
- `cssclasses` (atributo de `calendar.HTMLCalendar`), 214
- `cssclasses_weekday_head` (atributo de `calendar.HTMLCalendar`), 215
- `csv`, 511
- `csv` (módulo), 511
- `cte` (atributo de `email.headerregistry.ContentTransferEncoding`), 1064
- `cte_type` (atributo de `email.policy.Policy`), 1055
- `ctermid()` (en el módulo `os`), 560
- `ctime()` (en el módulo `time`), 624
- `ctime()` (método de `datetime.date`), 185
- `ctime()` (método de `datetime.datetime`), 195
- `ctrl()` (en el módulo `curses.ascii`), 725
- `CTRL_BREAK_EVENT` (en el módulo `signal`), 1026
- `CTRL_C_EVENT` (en el módulo `signal`), 1026
- `ctypes` (módulo), 736
- `curdir` (en el módulo `os`), 608
- `currency()` (en el módulo `locale`), 1381
- `current()` (método de `tkinter.ttk.Combobox`), 1444
- `current_process()` (en el módulo `multiprocessing`), 794
- `current_task()` (en el módulo `asyncio`), 875
- `current_task()` (método de clase de `asyncio.Task`), 878
- `current_thread()` (en el módulo `threading`), 769
- `CurrentByteIndex` (atributo de `xml.parsers.expat.xmlparser`), 1192
- `CurrentColumnNumber` (atributo de `xml.parsers.expat.xmlparser`), 1192
- `currentframe()` (en el módulo `inspect`), 1770
- `CurrentLineNumber` (atributo de `xml.parsers.expat.xmlparser`), 1192
- `curs_set()` (en el módulo `curses`), 706
- `curses` (módulo), 705
- `curses.ascii` (módulo), 724
- `curses.panel` (módulo), 726
- `curses.textpad` (módulo), 722
- `Cursor` (clase en `sqlite3`), 456
- `cursor()` (método de `sqlite3.Connection`), 450
- `cursyncup()` (método de `curses.window`), 713
- `Cut`, 1467
- `cwd()` (método de clase de `pathlib.Path`), 383
- `cwd()` (método de `ftplib.FTP`), 1261
- `cycle()` (en el módulo `itertools`), 348
- `Cyclic Redundancy Check`, 470
- ## D
- `-d`
gzip command line option, 475
- `-d destdir`
compileall command line option, 1838
- `D_FMT` (en el módulo `locale`), 1378
- `D_T_FMT` (en el módulo `locale`), 1378
- `daemon` (atributo de `multiprocessing.Process`), 789
- `daemon` (atributo de `threading.Thread`), 773
- `data`
packing binary, 153
tabular, 511
- `data` (atributo de `collections.UserDict`), 232
- `data` (atributo de `collections.UserList`), 232
- `data` (atributo de `collections.UserString`), 233
- `data` (atributo de `select.kevent`), 1013
- `data` (atributo de `selectors.SelectorKey`), 1015
- `data` (atributo de `urllib.request.Request`), 1225
- `data` (atributo de `xml.dom.Comment`), 1168
- `data` (atributo de `xml.dom.ProcessingInstruction`), 1169
- `data` (atributo de `xml.dom.Text`), 1168
- `data` (atributo de `xmlrpc.client.Binary`), 1323
- `Data` (clase en `plistlib`), 541
- `data()` (método de `xml.etree.ElementTree.TreeBuilder`), 1158
- `data_filter()` (en el módulo `tarfile`), 504
- `data_open()` (método de `urllib.request.DataHandler`), 1231
- `data_received()` (método de `asyncio.Protocol`), 927
- `database`
Unicode, 144
- `DatabaseError`, 460
- `databases`, 445
- `dataclass()` (en el módulo `dataclasses`), 1722
- `dataclasses` (módulo), 1721
- `datagram_received()` (método de `asyncio.DatagramProtocol`), 929
- `DatagramHandler` (clase en `logging.handlers`), 698

- DatagramProtocol (clase en *asyncio*), 926
- DatagramRequestHandler (clase en *socketserver*), 1298
- DatagramTransport (clase en *asyncio*), 922
- DataHandler (clase en *urllib.request*), 1224
- date (clase en *datetime*), 183
- date() (método de *datetime.datetime*), 191
- date() (método de *nnplib.NNTP*), 1277
- date_time (atributo de *zipfile.ZipInfo*), 492
- date_time_string() (método *http.server.BaseHTTPRequestHandler*), 1305
- DateHeader (clase en *email.headerregistry*), 1063
- datetime (atributo *email.headerregistry.DateHeader*), 1063
- datetime (clase en *datetime*), 187
- DateTime (clase en *xmlrpc.client*), 1323
- datetime (módulo), 177
- day (atributo de *datetime.date*), 184
- day (atributo de *datetime.datetime*), 190
- day_abbr (en el módulo *calendar*), 216
- day_name (en el módulo *calendar*), 216
- daylight (en el módulo *time*), 631
- Daylight Saving Time, 623
- DbfilenameShelf (clase en *shelve*), 440
- dbm (módulo), 442
- dbm.dumb (módulo), 445
- dbm.gnu
- módulo, 439
- dbm.gnu (módulo), 443
- dbm.ndbm
- módulo, 439
- dbm.ndbm (módulo), 445
- dcgettext() (en el módulo *locale*), 1383
- debug (atributo de *imaplib.IMAP4*), 1271
- debug (atributo de *shlex.shlex*), 1426
- debug (atributo de *zipfile.ZipFile*), 490
- DEBUG (en el módulo *re*), 116
- debug (*pdb* command), 1646
- debug() (en el módulo *doctest*), 1517
- debug() (en el módulo *logging*), 676
- debug() (método de *logging.Logger*), 667
- debug() (método de *pipes.Template*), 1892
- debug() (método de *unittest.TestCase*), 1529
- debug() (método de *unittest.TestSuite*), 1538
- DEBUG_BYTECODE_SUFFIXES (en el módulo *importlib.machinery*), 1799
- DEBUG_COLLECTABLE (en el módulo *gc*), 1757
- DEBUG_LEAK (en el módulo *gc*), 1757
- DEBUG_SAVEALL (en el módulo *gc*), 1757
- debug_src() (en el módulo *doctest*), 1517
- DEBUG_STATS (en el módulo *gc*), 1757
- DEBUG_UNCOLLECTABLE (en el módulo *gc*), 1757
- debugger, 1466, 1699, 1706
- configuration file, 1642
- debugging, 1640
- CGI, 1207
- DebuggingServer (clase en *smtpd*), 1286
- debuglevel (atributo de *http.client.HTTPResponse*), 1255
- DebugRunner (clase en *doctest*), 1518
- Decimal (clase en *decimal*), 302
- decimal (módulo), 298
- decimal() (en el módulo *unicodedata*), 144
- DecimalException (clase en *decimal*), 315
- decode
- Codecs, 158
- de decode (atributo de *codecs.CodecInfo*), 159
- decode() (en el módulo *base64*), 1129
- decode() (en el módulo *codecs*), 159
- decode() (en el módulo *quopri*), 1133
- de decode() (en el módulo *uu*), 1134
- decode() (método de *bytearray*), 57
- decode() (método de *bytes*), 57
- decode() (método de *codecs.Codec*), 163
- decode() (método de *codecs.IncrementalDecoder*), 165
- decode() (método de *json.JSONDecoder*), 1100
- decode() (método de *xmlrpc.client.Binary*), 1323
- decode() (método de *xmlrpc.client.DateTime*), 1323
- decode_header() (en el módulo *email.header*), 1088
- decode_header() (en el módulo *nntplib*), 1278
- decode_params() (en el módulo *email.utils*), 1094
- decode_rfc2231() (en el módulo *email.utils*), 1094
- decode_source() (en el módulo *importlib.util*), 1804
- decodebytes() (en el módulo *base64*), 1129
- DecodedGenerator (clase en *email.generator*), 1052
- decodestring() (en el módulo *base64*), 1129
- decodestring() (en el módulo *quopri*), 1133
- decomposition() (en el módulo *unicodedata*), 144
- decompress
- gzip command line option, 475
- decompress() (en el módulo *bz2*), 478
- decompress() (en el módulo *gzip*), 474
- decompress() (en el módulo *lzma*), 483
- decompress() (en el módulo *zlib*), 470
- decompress() (método de *bz2.BZ2Decompressor*), 477
- decompress() (método de *lzma.LZMADecompressor*), 482
- decompress() (método de *zlib.Decompress*), 472
- decompressobj() (en el módulo *zlib*), 471
- decorador, 1938
- DEDENT (en el módulo *token*), 1827
- dedent() (en el módulo *textwrap*), 141
- deepcopy() (en el módulo *copy*), 258
- def_prog_mode() (en el módulo *curses*), 706
- def_shell_mode() (en el módulo *curses*), 706
- default (atributo de *inspect.Parameter*), 1764
- default (atributo de *optparse.Option*), 1914
- default (en el módulo *email.policy*), 1058
- DEFAULT (en el módulo *unittest.mock*), 1579
- default() (método de *cmd.Cmd*), 1419
- default() (método de *json.JSONEncoder*), 1101

- DEFAULT_BUFFER_SIZE (en el módulo *io*), 611
default_bufsize (en el módulo *xml.dom.pulldom*), 1177
default_exception_handler() (método de *asyncio.loop*), 911
default_factory (atributo de *collections.defaultdict*), 226
DEFAULT_FORMAT (en el módulo *tarfile*), 497
DEFAULT_IGNORES (en el módulo *filecmp*), 403
default_open() (método de *urllib.request.BaseHandler*), 1227
DEFAULT_PROTOCOL (en el módulo *pickle*), 423
default_timer() (en el módulo *timeit*), 1655
DefaultContext (clase en *decimal*), 309
DefaultCookiePolicy (clase en *http.cookiejar*), 1312
defaultdict (clase en *collections*), 226
DefaultDict (clase en *typing*), 1488
DefaultEventLoopPolicy (clase en *asyncio*), 935
DefaultHandler() (método de *xml.parsers.expat.xmlparser*), 1193
DefaultHandlerExpand() (método de *xml.parsers.expat.xmlparser*), 1194
defaults() (método de *configparser.ConfigParser*), 531
DefaultSelector (clase en *selectors*), 1016
defaultTestLoader (en el módulo *unittest*), 1543
defaultTestResult() (método de *unittest.TestCase*), 1535
defects (atributo de *email.headerregistry.BaseHeader*), 1062
defects (atributo de *email.message.EmailMessage*), 1046
defects (atributo de *email.message.Message*), 1083
defpath (en el módulo *os*), 608
DefragResult (clase en *urllib.parse*), 1244
DefragResultBytes (clase en *urllib.parse*), 1244
degrees() (en el módulo *math*), 293
degrees() (en el módulo *turtle*), 1395
del
 sentencia, 40, 77
del_param() (método de *email.message.EmailMessage*), 1042
del_param() (método de *email.message.Message*), 1081
delattr() (función incorporada), 9
delay() (en el módulo *turtle*), 1407
delay_output() (en el módulo *curses*), 706
delayload (atributo de *http.cookiejar.FileCookieJar*), 1314
delch() (método de *curses.window*), 713
dele() (método de *poplib.POP3*), 1264
delete() (método de *ftplib.FTP*), 1261
delete() (método de *imaplib.IMAP4*), 1267
delete() (método de *tkinter.ttk.Treeview*), 1452
DELETE_ATTR (opcode), 1850
DELETE_DEREF (opcode), 1852
DELETE_FAST (opcode), 1852
DELETE_GLOBAL (opcode), 1850
DELETE_NAME (opcode), 1850
DELETE_SUBSCR (opcode), 1847
deleteacl() (método de *imaplib.IMAP4*), 1267
deletefilehandler() (método de *tkinter.Widget.tk*), 1440
DeleteKey() (en el módulo *winreg*), 1870
DeleteKeyEx() (en el módulo *winreg*), 1870
deleteln() (método de *curses.window*), 713
deleteMe() (método de *bdb.Breakpoint*), 1633
DeleteValue() (en el módulo *winreg*), 1870
delimiter (atributo de *csv.Dialect*), 515
delitem() (en el módulo *operator*), 367
deliver_challenge() (en el módulo *multiprocessing.connection*), 812
delocalize() (en el módulo *locale*), 1381
demo_app() (en el módulo *wsgiref.simple_server*), 1213
denominator (atributo de *fractions.Fraction*), 325
denominator (atributo de *numbers.Rational*), 286
DeprecationWarning, 96
deque (clase en *collections*), 222
Deque (clase en *typing*), 1486
dequeue() (método de *logging.handlers.QueueListener*), 704
DER_cert_to_PEM_cert() (en el módulo *ssl*), 978
derwin() (método de *curses.window*), 714
DES
 cipher, 1884
description (atributo de *inspect.Parameter.kind*), 1765
description (atributo de *sqlite3.Cursor*), 459
description() (método de *nnplib.NNTP*), 1275
descriptions() (método de *nnplib.NNTP*), 1275
descriptor, 1938
despacho único, 1946
dest (atributo de *optparse.Option*), 1914
detach() (método de *io.BufferedIOBase*), 615
detach() (método de *io.TextIOBase*), 619
detach() (método de *socket.socket*), 963
detach() (método de *tkinter.ttk.Treeview*), 1452
detach() (método de *weakref.finalize*), 249
Detach() (método de *winreg.PyHKEY*), 1877
DETACHED_PROCESS (en el módulo *subprocess*), 848
--details
 inspect command line option, 1773
detect_api_mismatch() (en el módulo *test.support*), 1625
detect_encoding() (en el módulo *tokenize*), 1831
deterministic profiling, 1646
device_encoding() (en el módulo *os*), 567
devnull (en el módulo *os*), 608
DEVNULL (en el módulo *subprocess*), 838
devpoll() (en el módulo *select*), 1007
DevpollSelector (clase en *selectors*), 1016
dgettext() (en el módulo *gettext*), 1368
dgettext() (en el módulo *locale*), 1383
dialect (atributo de *csv.csvreader*), 516

- dialect (*atributo de csv.csvwriter*), 516
 Dialect (*clase en csv*), 514
 Dialog (*clase en msilib*), 1866
 diccionario, **1938**
 dict (*2to3 fixer*), 1609
 Dict (*clase en typing*), 1488
 dict (*clase incorporada*), 78
 dict() (*método de multiprocessing.managers.SyncManager*), 804
 dictConfig() (*en el módulo logging.config*), 681
 dictionary
 objeto, 77
 type, operations on, 77
 dictionary comprehension, **1938**
 DictReader (*clase en csv*), 513
 DictWriter (*clase en csv*), 513
 diff_bytes() (*en el módulo difflib*), 133
 diff_files (*atributo de filecmp.dircmp*), 403
 Differ (*clase en difflib*), 130
 difference() (*método de frozenset*), 76
 difference_update() (*método de frozenset*), 77
 difflib (*módulo*), 129
 digest() (*en el módulo hmac*), 553
 digest() (*método de hashlib.hash*), 545
 digest() (*método de hashlib.shake*), 545
 digest() (*método de hmac.HMAC*), 554
 digest_size (*atributo de hmac.HMAC*), 554
 digit() (*en el módulo unicodedata*), 144
 digits (*en el módulo string*), 99
 dir() (*función incorporada*), 9
 dir() (*método de ftplib.FTP*), 1261
 dircmp (*clase en filecmp*), 402
 directory
 changing, 577
 creating, 580
 deleting, 413, 583
 site-packages, 1773
 traversal, 592, 593
 walking, 592, 593
 directory ...
 compileall command line option, 1838
 directory (*atributo de http.server.SimpleHTTPRequestHandler*), 1306
 Directory (*clase en msilib*), 1865
 DirEntry (*clase en os*), 584
 DirList (*clase en tkinter.tix*), 1460
 dirname() (*en el módulo os.path*), 390
 DirSelectBox (*clase en tkinter.tix*), 1461
 DirSelectDialog (*clase en tkinter.tix*), 1460
 DirsOnSysPath (*clase en test.support*), 1626
 DirTree (*clase en tkinter.tix*), 1460
 dis (*módulo*), 1841
 dis() (*en el módulo dis*), 1843
 dis() (*en el módulo pickletools*), 1856
 dis() (*método de dis.Bytecode*), 1842
 disable (*pdb command*), 1643
 disable() (*en el módulo faulthandler*), 1638
 disable() (*en el módulo gc*), 1754
 disable() (*en el módulo logging*), 678
 disable() (*método de bdb.Breakpoint*), 1633
 disable() (*método de profile.Profile*), 1650
 disable_faulthandler() (*en el módulo test.support*), 1620
 disable_gc() (*en el módulo test.support*), 1620
 disable_interspersed_args() (*método de optparse.OptionParser*), 1919
 DisableReflectionKey() (*en el módulo winreg*), 1874
 disassemble() (*en el módulo dis*), 1843
 discard (*atributo de http.cookiejar.Cookie*), 1318
 discard() (*método de frozenset*), 77
 discard() (*método de mailbox.Mailbox*), 1107
 discard() (*método de mailbox.MH*), 1113
 discard_buffers() (*método de asynchat.async_chat*), 1022
 disco() (*en el módulo dis*), 1843
 discover() (*método de unittest.TestLoader*), 1540
 disk_usage() (*en el módulo shutil*), 414
 dispatch_call() (*método de bdb.Bdb*), 1635
 dispatch_exception() (*método de bdb.Bdb*), 1635
 dispatch_line() (*método de bdb.Bdb*), 1634
 dispatch_return() (*método de bdb.Bdb*), 1635
 dispatch_table (*atributo de pickle.Pickler*), 425
 dispatcher (*clase en asyncore*), 1018
 dispatcher_with_send (*clase en asyncore*), 1020
 display (*pdb command*), 1645
 display_name (*atributo de email.headerregistry.Address*), 1066
 display_name (*atributo de email.headerregistry.Group*), 1066
 displayhook() (*en el módulo sys*), 1694
 dist() (*en el módulo math*), 292
 distance() (*en el módulo turtle*), 1395
 distb() (*en el módulo dis*), 1843
 distutils (*módulo*), 1673
 divide() (*método de decimal.Context*), 312
 divide_int() (*método de decimal.Context*), 312
 división entera, **1939**
 DivisionByZero (*clase en decimal*), 315
 divmod() (*función incorporada*), 10
 divmod() (*método de decimal.Context*), 312
 DllCanUnloadNow() (*en el módulo ctypes*), 761
 DllGetClassObject() (*en el módulo ctypes*), 761
 dllhandle (*en el módulo sys*), 1693
 dngettext() (*en el módulo gettext*), 1368
 dnpgettext() (*en el módulo gettext*), 1368
 do_clear() (*método de bdb.Bdb*), 1635
 do_command() (*método de curses.textpad.Textbox*), 723
 do_GET() (*método de http.server.SimpleHTTPRequestHandler*), 1306
 do_handshake() (*método de ssl.SSLSocket*), 986

- `do_HEAD()` (método *http.server.SimpleHTTPRequestHandler*), 1306
- `do_POST()` (método *http.server.CGIHTTPRequestHandler*), 1307
- `doc` (atributo de *json.JSONDecodeError*), 1102
- `doc_header` (atributo de *cmd.Cmd*), 1420
- `DocCGIXMLRPCRequestHandler` (clase en *xmlrpc.server*), 1332
- `DocFileSuite()` (en el módulo *doctest*), 1509
- `doClassCleanups()` (método de clase de *unittest.TestCase*), 1536
- `doCleanups()` (método de *unittest.TestCase*), 1536
- `doccmd()` (método de *smtpplib.SMTP*), 1280
- `docstring`, 1938
- `docstring` (atributo de *doctest.DocTest*), 1512
- `DocTest` (clase en *doctest*), 1512
- `doctest` (módulo), 1497
- `DocTestFailure`, 1518
- `DocTestFinder` (clase en *doctest*), 1513
- `DocTestParser` (clase en *doctest*), 1514
- `DocTestRunner` (clase en *doctest*), 1514
- `DocTestSuite()` (en el módulo *doctest*), 1510
- `doctype()` (método *xml.etree.ElementTree.TreeBuilder*), 1158
- `documentation`
 - generation, 1496
 - online, 1496
- `documentElement` (atributo de *xml.dom.Document*), 1166
- `DocXMLRPCRequestHandler` (clase en *xmlrpc.server*), 1332
- `DocXMLRPCServer` (clase en *xmlrpc.server*), 1332
- `domain` (atributo de *email.headerregistry.Address*), 1066
- `domain` (atributo de *tracemalloc.DomainFilter*), 1667
- `domain` (atributo de *tracemalloc.Filter*), 1667
- `domain` (atributo de *tracemalloc.Trace*), 1670
- `domain_initial_dot` (atributo *http.cookiejar.Cookie*), 1318
- `domain_return_ok()` (método *http.cookiejar.CookiePolicy*), 1315
- `domain_specified` (atributo *http.cookiejar.Cookie*), 1318
- `DomainFilter` (clase en *tracemalloc*), 1667
- `DomainLiberal` (atributo *http.cookiejar.DefaultCookiePolicy*), 1317
- `DomainRFC2965Match` (atributo *http.cookiejar.DefaultCookiePolicy*), 1317
- `DomainStrict` (atributo *http.cookiejar.DefaultCookiePolicy*), 1317
- `DomainStrictNoDots` (atributo *http.cookiejar.DefaultCookiePolicy*), 1317
- `DomainStrictNonDomain` (atributo *http.cookiejar.DefaultCookiePolicy*), 1317
- `DOMEventStream` (clase en *xml.dom.pulldom*), 1177
- `DOMException`, 1169
- `doModuleCleanups()` (en el módulo *unittest*), 1547
- `de DomstringSizeErr`, 1169
- `done()` (en el módulo *turtle*), 1409
- `done()` (método de *asyncio.Future*), 919
- `done()` (método de *asyncio.Task*), 876
- `done()` (método de *concurrent.futures.Future*), 834
- `done()` (método de *xdrlib.Unpacker*), 538
- `DONT_ACCEPT_BLANKLINE` (en el módulo *doctest*), 1504
- `DONT_ACCEPT_TRUE_FOR_1` (en el módulo *doctest*), 1504
- `dont_write_bytecode` (en el módulo *sys*), 1694
- `doRollover()` (método de *logging.handlers.RotatingFileHandler*), 695
- `doRollover()` (método de *logging.handlers.TimedRotatingFileHandler*), 696
- `DOT` (en el módulo *token*), 1828
- `dot()` (en el módulo *turtle*), 1392
- `DOTALL` (en el módulo *re*), 117
- `doublequote` (atributo de *csv.Dialect*), 515
- `DOUBLESASH` (en el módulo *token*), 1829
- `DOUBLESASHEQUAL` (en el módulo *token*), 1829
- `DOUBLESTAR` (en el módulo *token*), 1828
- `DOUBLESTAREQUAL` (en el módulo *token*), 1829
- `doupdate()` (en el módulo *curses*), 706
- `down` (*pdb* command), 1643
- `down()` (en el módulo *turtle*), 1396
- `dpgettext()` (en el módulo *gettext*), 1368
- `drain()` (método de *asyncio.StreamWriter*), 882
- `drop_whitespace` (atributo de *textwrap.TextWrapper*), 142
- `dropwhile()` (en el módulo *itertools*), 348
- `dst()` (método de *datetime.datetime*), 192
- `dst()` (método de *datetime.time*), 200
- `dst()` (método de *datetime.timezone*), 208
- `dst()` (método de *datetime.tzinfo*), 201
- `DTDHandler` (clase en *xml.sax.handler*), 1179
- `DumbWriter` (clase en *formatter*), 1860
- `dummy_threading` (módulo), 864
- `dump()` (en el módulo *ast*), 1823
- `dump()` (en el módulo *json*), 1098
- `dump()` (en el módulo *marshal*), 441
- `dump()` (en el módulo *pickle*), 424
- `dump()` (en el módulo *plistlib*), 540
- `dump()` (en el módulo *xml.etree.ElementTree*), 1150
- `dump()` (método de *pickle.Pickler*), 425
- `dump()` (método de *tracemalloc.Snapshot*), 1668
- `dump_stats()` (método de *profile.Profile*), 1650
- `dump_stats()` (método de *pstats.Stats*), 1650
- `dump_traceback()` (en el módulo *faulthandler*), 1638
- `dump_traceback_later()` (en el módulo *faulthandler*), 1639
- `dumps()` (en el módulo *json*), 1098
- `dumps()` (en el módulo *marshal*), 441
- `dumps()` (en el módulo *pickle*), 424
- `dumps()` (en el módulo *plistlib*), 540
- `dumps()` (en el módulo *xmlrpc.client*), 1326

dup() (en el módulo *os*), 567
 dup() (método de *socket.socket*), 963
 dup2() (en el módulo *os*), 567
 DUP_TOP (opcode), 1845
 DUP_TOP_TWO (opcode), 1845
 DuplicateOptionError, 534
 DuplicateSectionError, 534
 dwFlags (atributo de *subprocess.STARTUPINFO*), 846
 DynamicClassAttribute() (en el módulo *types*), 257

E

-e
 tokenize command line option, 1832
 e (en el módulo *cmath*), 297
 e (en el módulo *math*), 294
 -e <tarfile> [<output_dir>]
 tarfile command line option, 507
 -e <zipfile> <output_dir>
 zipfile command line option, 494
 E2BIG (en el módulo *errno*), 730
 EACCES (en el módulo *errno*), 731
 EADDRINUSE (en el módulo *errno*), 734
 EADDRNOTAVAIL (en el módulo *errno*), 734
 EADV (en el módulo *errno*), 733
 EAFNOSUPPORT (en el módulo *errno*), 734
 EAFP, 1938
 EAGAIN (en el módulo *errno*), 731
 EALREADY (en el módulo *errno*), 735
 east_asian_width() (en el módulo *unicodedata*), 144
 EBADE (en el módulo *errno*), 732
 EBADF (en el módulo *errno*), 731
 EBADFD (en el módulo *errno*), 733
 EBADMSG (en el módulo *errno*), 733
 EBADR (en el módulo *errno*), 732
 EBADRQC (en el módulo *errno*), 733
 EBADSLT (en el módulo *errno*), 733
 EBFONT (en el módulo *errno*), 733
 EBUSY (en el módulo *errno*), 731
 ECHILD (en el módulo *errno*), 731
 echo() (en el módulo *curses*), 707
 echochar() (método de *curses.window*), 714
 ECHRNQ (en el módulo *errno*), 732
 ECOMM (en el módulo *errno*), 733
 ECONNABORTED (en el módulo *errno*), 735
 ECONNREFUSED (en el módulo *errno*), 735
 ECONNRESET (en el módulo *errno*), 735
 EDEADLK (en el módulo *errno*), 732
 EDEADLOCK (en el módulo *errno*), 733
 EDESTADDRREQ (en el módulo *errno*), 734
 edit() (método de *curses.textpad.Textbox*), 722
 EDOM (en el módulo *errno*), 732
 EDOTDOT (en el módulo *errno*), 733
 EDQUOT (en el módulo *errno*), 735
 EEXIST (en el módulo *errno*), 731
 EFAULT (en el módulo *errno*), 731
 EFBIG (en el módulo *errno*), 731

effective() (en el módulo *bdb*), 1637
 ehlo() (método de *smtpplib.SMTP*), 1281
 ehlo_or_helo_if_needed() (método de *smtpplib.SMTP*), 1281
 EHOSTDOWN (en el módulo *errno*), 735
 EHOSTUNREACH (en el módulo *errno*), 735
 EIDRM (en el módulo *errno*), 732
 EILSEQ (en el módulo *errno*), 734
 EINPROGRESS (en el módulo *errno*), 735
 EINTR (en el módulo *errno*), 730
 EINVAL (en el módulo *errno*), 731
 EIO (en el módulo *errno*), 730
 EISCONN (en el módulo *errno*), 735
 EISDIR (en el módulo *errno*), 731
 EISNAM (en el módulo *errno*), 735
 EL2HLT (en el módulo *errno*), 732
 EL2NSYNC (en el módulo *errno*), 732
 EL3HLT (en el módulo *errno*), 732
 EL3RST (en el módulo *errno*), 732
 Element (clase en *xml.etree.ElementTree*), 1153
 element_create() (método de *tkinter.ttk.Style*), 1457
 element_names() (método de *tkinter.ttk.Style*), 1457
 element_options() (método de *tkinter.ttk.Style*), 1457
 ElementDeclHandler() (método de *xml.parsers.expat.xmlparser*), 1192
 elements() (método de *collections.Counter*), 220
 ElementTree (clase en *xml.etree.ElementTree*), 1156
 ELIBACC (en el módulo *errno*), 734
 ELIBBAD (en el módulo *errno*), 734
 ELIBEXEC (en el módulo *errno*), 734
 ELIBMAX (en el módulo *errno*), 734
 ELIBSCN (en el módulo *errno*), 734
 Ellinghouse, Lance, 1133
 ELLIPSIS (en el módulo *doctest*), 1504
 ELLIPSIS (en el módulo *token*), 1829
 Ellipsis (variable incorporada), 27
 ELNRNG (en el módulo *errno*), 732
 ELOOP (en el módulo *errno*), 732
 email (módulo), 1037
 email.charset (módulo), 1089
 email.contentmanager (módulo), 1067
 email.encoders (módulo), 1091
 email.errors (módulo), 1060
 email.generator (módulo), 1050
 email.header (módulo), 1086
 email.headerregistry (módulo), 1061
 email.iterators (módulo), 1095
 EmailMessage (clase en *email.message*), 1039
 email.message (módulo), 1038
 email.mime (módulo), 1084
 email.parser (módulo), 1046
 EmailPolicy (clase en *email.policy*), 1057
 email.policy (módulo), 1053
 email.utils (módulo), 1092
 EMFILE (en el módulo *errno*), 731
 emit() (método de *logging.FileHandler*), 693

- `emit()` (método de `logging.Handler`), 670
- `emit()` (método de `logging.handlers.BufferingHandler`), 701
- `emit()` (método de `logging.handlers.DatagramHandler`), 698
- `emit()` (método de `logging.handlers.HTTPHandler`), 702
- `emit()` (método de `logging.handlers.NTEventLogHandler`), 700
- `emit()` (método de `logging.handlers.QueueHandler`), 703
- `emit()` (método de `logging.handlers.RotatingFileHandler`), 695
- `emit()` (método de `logging.handlers.SMTPHandler`), 701
- `emit()` (método de `logging.handlers.SocketHandler`), 697
- `emit()` (método de `logging.handlers.SysLogHandler`), 698
- `emit()` (método de `logging.handlers.TimedRotatingFileHandler`), 696
- `emit()` (método de `logging.handlers.WatchedFileHandler`), 694
- `emit()` (método de `logging.NullHandler`), 693
- `emit()` (método de `logging.StreamHandler`), 692
- `EMLINK` (en el módulo `errno`), 731
- `Empty`, 856
- `empty` (atributo de `inspect.Parameter`), 1764
- `empty` (atributo de `inspect.Signature`), 1763
- `empty()` (método de `asyncio.Queue`), 894
- `empty()` (método de `multiprocessing.Queue`), 792
- `empty()` (método de `multiprocessing.SimpleQueue`), 793
- `empty()` (método de `queue.Queue`), 856
- `empty()` (método de `queue.SimpleQueue`), 858
- `empty()` (método de `sched.scheduler`), 855
- `EMPTY_NAMESPACE` (en el módulo `xml.dom`), 1162
- `emptyline()` (método de `cmd.Cmd`), 1419
- `EMSGSIZE` (en el módulo `errno`), 734
- `EMULTIHOP` (en el módulo `errno`), 733
- `enable` (`pdb` command), 1643
- `enable()` (en el módulo `cgitb`), 1209
- `enable()` (en el módulo `faulthandler`), 1638
- `enable()` (en el módulo `gc`), 1754
- `enable()` (método de `bdb.Breakpoint`), 1633
- `enable()` (método de `imaplib.IMAP4`), 1267
- `enable()` (método de `profile.Profile`), 1649
- `enable_callback_tracebacks()` (en el módulo `sqlite3`), 450
- `enable_interspersed_args()` (método de `optparse.OptionParser`), 1919
- `enable_load_extension()` (método de `sqlite3.Connection`), 453
- `enable_traversal()` (método de `tkinter.ttk.Notebook`), 1448
- `ENABLE_USER_SITE` (en el módulo `site`), 1775
- `EnableReflectionKey()` (en el módulo `winreg`), 1874
- `ENAMETOOLONG` (en el módulo `errno`), 732
- `ENAVAIL` (en el módulo `errno`), 735
- `enclose()` (método de `curses.window`), 714
- `encode`
 - `Codecs`, 158
- `encode` (atributo de `codecs.CodecInfo`), 159
- `encode()` (en el módulo `base64`), 1129
- `encode()` (en el módulo `codecs`), 158
- `encode()` (en el módulo `quopri`), 1133
- `encode()` (en el módulo `uu`), 1133
- `encode()` (método de `codecs.Codec`), 163
- `encode()` (método de `codecs.IncrementalEncoder`), 164
- `encode()` (método de `email.header.Header`), 1088
- `encode()` (método de `json.JSONEncoder`), 1102
- `encode()` (método de `str`), 45
- `encode()` (método de `xmlrpc.client.Binary`), 1323
- `encode()` (método de `xmlrpc.client.DateTime`), 1323
- `encode_7or8bit()` (en el módulo `email.encoders`), 1092
- `encode_base64()` (en el módulo `email.encoders`), 1092
- `encode_noop()` (en el módulo `email.encoders`), 1092
- `encode_quopri()` (en el módulo `email.encoders`), 1091
- `encode_rfc2231()` (en el módulo `email.utils`), 1094
- `encodebytes()` (en el módulo `base64`), 1129
- `EncodedFile()` (en el módulo `codecs`), 160
- `encodePriority()` (método de `logging.handlers.SysLogHandler`), 699
- `encodestring()` (en el módulo `base64`), 1129
- `encodestring()` (en el módulo `quopri`), 1133
- `encoding`
 - `base64`, 1127
 - `quoted-printable`, 1133
- `encoding` (atributo de `curses.window`), 714
- `encoding` (atributo de `io.TextIOBase`), 619
- `encoding` (atributo de `UnicodeError`), 94
- `ENCODING` (en el módulo `tarfile`), 497
- `ENCODING` (en el módulo `token`), 1829
- `encodings_map` (atributo de `mimetypes.MimeTypes`), 1126
- `encodings_map` (en el módulo `mimetypes`), 1125
- `encodings.idna` (módulo), 174
- `encodings.mbcx` (módulo), 175
- `encodings.utf_8_sig` (módulo), 175
- `end` (atributo de `UnicodeError`), 94
- `end()` (método de `re.Match`), 123
- `end()` (método de `xml.etree.ElementTree.TreeBuilder`), 1158
- `END_ASYNC_FOR` (opcode), 1847
- `end_col_offset` (atributo de `ast.AST`), 1818
- `end_fill()` (en el módulo `turtle`), 1399
- `END_FINALLY` (opcode), 1849
- `end_headers()` (método de `http.server.BaseHTTPRequestHandler`), 1305
- `end_lineno` (atributo de `ast.AST`), 1818

- `end_ns()` (método *xml.etree.ElementTree.TreeBuilder*), 1158
- `end_paragraph()` (método de *formatter.formatter*), 1858
- `end_poly()` (en el módulo *turtle*), 1404
- `EndCdataSectionHandler()` (método *xml.parsers.expat.xmlparser*), 1193
- `EndDoctypeDeclHandler()` (método *xml.parsers.expat.xmlparser*), 1192
- `endDocument()` (método *xml.sax.handler.ContentHandler*), 1181
- `endElement()` (método *xml.sax.handler.ContentHandler*), 1182
- `EndElementHandler()` (método *xml.parsers.expat.xmlparser*), 1193
- `endElementNS()` (método *xml.sax.handler.ContentHandler*), 1182
- `endheaders()` (método *http.client.HTTPConnection*), 1254
- `ENDMARKER` (en el módulo *token*), 1827
- `EndNamespaceDeclHandler()` (método *xml.parsers.expat.xmlparser*), 1193
- `endpos` (atributo de *re.Match*), 124
- `endPrefixMapping()` (método *xml.sax.handler.ContentHandler*), 1182
- `endswith()` (método de *bytearray*), 57
- `endswith()` (método de *bytes*), 57
- `endswith()` (método de *str*), 45
- `endwin()` (en el módulo *curses*), 707
- `ENETDOWN` (en el módulo *errno*), 734
- `ENETRESET` (en el módulo *errno*), 735
- `ENETUNREACH` (en el módulo *errno*), 735
- `ENFILE` (en el módulo *errno*), 731
- `ENOANO` (en el módulo *errno*), 733
- `ENOBUFFS` (en el módulo *errno*), 735
- `ENOCSSI` (en el módulo *errno*), 732
- `ENODATA` (en el módulo *errno*), 733
- `ENODEV` (en el módulo *errno*), 731
- `ENOENT` (en el módulo *errno*), 730
- `ENOEXEC` (en el módulo *errno*), 730
- `ENOLCK` (en el módulo *errno*), 732
- `ENOLINK` (en el módulo *errno*), 733
- `ENOMEM` (en el módulo *errno*), 731
- `ENOMSG` (en el módulo *errno*), 732
- `ENONET` (en el módulo *errno*), 733
- `ENOPKG` (en el módulo *errno*), 733
- `ENOPROTOOPT` (en el módulo *errno*), 734
- `ENOSPC` (en el módulo *errno*), 731
- `ENOSR` (en el módulo *errno*), 733
- `ENOSTR` (en el módulo *errno*), 733
- `ENOSYS` (en el módulo *errno*), 732
- `ENOTBLK` (en el módulo *errno*), 731
- `ENOTCONN` (en el módulo *errno*), 735
- `ENOTDIR` (en el módulo *errno*), 731
- `ENOTEMPTY` (en el módulo *errno*), 732
- `ENOTNAM` (en el módulo *errno*), 735
- `ENOTSOCK` (en el módulo *errno*), 734
- `ENOTTY` (en el módulo *errno*), 731
- `de ENOTUNIQ` (en el módulo *errno*), 733
- `enqueue()` (método *de logging.handlers.QueueHandler*), 703
- `enqueue_sentinel()` (método *de logging.handlers.QueueListener*), 704
- `de ensure_directories()` (método *de venv.EnvBuilder*), 1679
- `de ensure_future()` (en el módulo *asyncio*), 918
- `ensurepip` (módulo), 1674
- `de enter()` (método de *sched.scheduler*), 855
- `de enter_async_context()` (método de *contextlib.AsyncExitStack*), 1735
- `de enter_context()` (método de *contextlib.ExitStack*), 1734
- `de enterabs()` (método de *sched.scheduler*), 854
- `de entities` (atributo de *xml.dom.DocumentType*), 1165
- `EntityDeclHandler()` (método *de xml.parsers.expat.xmlparser*), 1193
- `entitydefs` (en el módulo *html.entities*), 1140
- `EntityResolver` (clase en *xml.sax.handler*), 1179
- `de entorno virtual`, 1947
- `entrada de ruta`, 1944
- `Enum` (clase en *enum*), 267
- `de enum` (módulo), 266
- `enum_certificates()` (en el módulo *ssl*), 978
- `enum_crls()` (en el módulo *ssl*), 979
- `enumerate()` (en el módulo *threading*), 770
- `enumerate()` (función incorporada), 10
- `EnumKey()` (en el módulo *winreg*), 1870
- `EnumValue()` (en el módulo *winreg*), 1870
- `EnvBuilder` (clase en *venv*), 1678
- `environ` (en el módulo *os*), 560
- `environ` (en el módulo *posix*), 1882
- `environb` (en el módulo *os*), 561
- `environment variables`
deleting, 566
setting, 564
- `EnvironmentError`, 95
- `Environments`
virtual, 1675
- `EnvironmentVarGuard` (clase en *test.support*), 1626
- `ENXIO` (en el módulo *errno*), 730
- `eof` (atributo de *bz2.BZ2Decompressor*), 478
- `eof` (atributo de *lzma.LZMADecompressor*), 482
- `eof` (atributo de *shlex.shlex*), 1426
- `eof` (atributo de *ssl.MemoryBIO*), 1004
- `eof` (atributo de *zlib.Decompress*), 472
- `eof_received()` (método *de asyncio.BufferedProtocol*), 928
- `eof_received()` (método de *asyncio.Protocol*), 927
- `EOFError`, 90
- `EOPNOTSUPP` (en el módulo *errno*), 734
- `EOVERFLOW` (en el módulo *errno*), 733
- `EPERM` (en el módulo *errno*), 730
- `EPFNOSUPPORT` (en el módulo *errno*), 734
- `epilogue` (atributo de *email.message.EmailMessage*), 1046

- `epilogue` (atributo de `email.message.Message`), 1083
- `EPIPE` (en el módulo `errno`), 732
- `epoch`, 622
- `epoll()` (en el módulo `select`), 1007
- `EpollSelector` (clase en `selectors`), 1016
- `EPROTO` (en el módulo `errno`), 733
- `EPROTONOSUPPORT` (en el módulo `errno`), 734
- `EPROTOTYPE` (en el módulo `errno`), 734
- `eq()` (en el módulo `operator`), 365
- `EQUAL` (en el módulo `token`), 1828
- `EQUAL` (en el módulo `token`), 1828
- `ERA` (en el módulo `locale`), 1379
- `ERA_D_FMT` (en el módulo `locale`), 1379
- `ERA_D_T_FMT` (en el módulo `locale`), 1379
- `ERA_T_FMT` (en el módulo `locale`), 1379
- `ERANGE` (en el módulo `errno`), 732
- `erase()` (método de `curses.window`), 714
- `erasechar()` (en el módulo `curses`), 707
- `EREMCHG` (en el módulo `errno`), 734
- `EREMOTE` (en el módulo `errno`), 733
- `EREMOTEIO` (en el módulo `errno`), 735
- `ERESTART` (en el módulo `errno`), 734
- `erf()` (en el módulo `math`), 293
- `erfc()` (en el módulo `math`), 293
- `EROFS` (en el módulo `errno`), 731
- `ERR` (en el módulo `curses`), 718
- `errcheck` (atributo de `ctypes._FuncPtr`), 757
- `errcode` (atributo de `xmlrpc.client.ProtocolError`), 1325
- `errmsg` (atributo de `xmlrpc.client.ProtocolError`), 1325
- `errno`
 - módulo, 92
- `errno` (atributo de `OSError`), 92
- `errno` (módulo), 730
- `error`, 120, 154, 442, 443, 445, 469, 559, 664, 706, 862, 953, 1007, 1189, 1347, 1893, 1897
- `Error`, 258, 415, 460, 515, 534, 539, 1122, 1130, 1132, 1134, 1199, 1353, 1356, 1376
- `error()` (en el módulo `logging`), 677
- `error()` (método de `argparse.ArgumentParser`), 661
- `error()` (método de `logging.Logger`), 668
- `error()` (método de `urllib.request.OpenerDirector`), 1226
- `error()` (método de `xml.sax.handler.ErrorHandler`), 1183
- `error_body` (atributo de `wsgi-ref.handlers.BaseHandler`), 1217
- `error_content_type` (atributo de `http.server.BaseHTTPRequestHandler`), 1303
- `error_headers` (atributo de `wsgi-ref.handlers.BaseHandler`), 1217
- `error_leader()` (método de `shlex.shlex`), 1425
- `error_message_format` (atributo de `http.server.BaseHTTPRequestHandler`), 1303
- `error_output()` (método de `wsgi-ref.handlers.BaseHandler`), 1217
- `error_perm`, 1259
- `error_proto`, 1259, 1263
- `error_received()` (método de `asyncio.DatagramProtocol`), 929
- `error_reply`, 1258
- `error_status` (atributo de `wsgi-ref.handlers.BaseHandler`), 1217
- `error_temp`, 1258
- `ErrorByteIndex` (atributo de `xml.parsers.expat.xmlparser`), 1192
- `ErrorCode` (atributo de `xml.parsers.expat.xmlparser`), 1192
- `errorcode` (en el módulo `errno`), 730
- `ErrorColumnNumber` (atributo de `xml.parsers.expat.xmlparser`), 1192
- `ErrorHandler` (clase en `xml.sax.handler`), 1179
- `errorlevel` (atributo de `tarfile.TarFile`), 500
- `ErrorLineNumber` (atributo de `xml.parsers.expat.xmlparser`), 1192
- `Errors`
 - logging, 665
- `errors` (atributo de `io.TextIOBase`), 619
- `errors` (atributo de `unittest.TestLoader`), 1539
- `errors` (atributo de `unittest.TestResult`), 1541
- `ErrorString()` (en el módulo `xml.parsers.expat`), 1189
- `ERRORTOKEN` (en el módulo `token`), 1829
- `escape` (atributo de `shlex.shlex`), 1425
- `escape()` (en el módulo `glob`), 408
- `escape()` (en el módulo `html`), 1135
- `escape()` (en el módulo `re`), 119
- `escape()` (en el módulo `xml.sax.saxutils`), 1184
- `escapechar` (atributo de `csv.Dialect`), 515
- `escapedquotes` (atributo de `shlex.shlex`), 1426
- `ESHUTDOWN` (en el módulo `errno`), 735
- `ESOCKTNOSUPPORT` (en el módulo `errno`), 734
- `espacio de nombres`, 1943
- `especificador de módulo`, 1943
- `ESPIPE` (en el módulo `errno`), 731
- `ESRCH` (en el módulo `errno`), 730
- `ESRMNT` (en el módulo `errno`), 733
- `ESTALE` (en el módulo `errno`), 735
- `ESTRPIPE` (en el módulo `errno`), 734
- `ETIME` (en el módulo `errno`), 733
- `ETIMEDOUT` (en el módulo `errno`), 735
- `Etiny()` (método de `decimal.Context`), 311
- `ETOOMANYREFS` (en el módulo `errno`), 735
- `Etop()` (método de `decimal.Context`), 311
- `ETXTBSY` (en el módulo `errno`), 731
- `EUCLEAN` (en el módulo `errno`), 735
- `EUNATCH` (en el módulo `errno`), 732
- `EUSERS` (en el módulo `errno`), 734
- `eval`
 - función incorporada, 84, 261, 1815
- `eval()` (función incorporada), 10
- `Event` (clase en `asyncio`), 886
- `Event` (clase en `multiprocessing`), 798
- `Event` (clase en `threading`), 779

- event scheduling, 854
- event() (método de *msilib.Control*), 1865
- Event() (método de *multiprocessing.managers.SyncManager*), 804
- events (atributo de *selectors.SelectorKey*), 1014
- events (widgets), 1438
- EWOLDBLOCK (en el módulo *errno*), 732
- EX_CANTCREAT (en el módulo *os*), 598
- EX_CONFIG (en el módulo *os*), 598
- EX_DATAERR (en el módulo *os*), 597
- EX_IOERR (en el módulo *os*), 598
- EX_NOHOST (en el módulo *os*), 597
- EX_NOINPUT (en el módulo *os*), 597
- EX_NOPERM (en el módulo *os*), 598
- EX_NOTFOUND (en el módulo *os*), 598
- EX_NOUSER (en el módulo *os*), 597
- EX_OK (en el módulo *os*), 597
- EX_OSERR (en el módulo *os*), 598
- EX_OSFILE (en el módulo *os*), 598
- EX_PROTOCOL (en el módulo *os*), 598
- EX_SOFTWARE (en el módulo *os*), 598
- EX_TEMPFAIL (en el módulo *os*), 598
- EX_UNAVAILABLE (en el módulo *os*), 597
- EX_USAGE (en el módulo *os*), 597
- exact
 - tokenize command line option, 1832
- example (atributo de *doctest.DocTestFailure*), 1518
- example (atributo de *doctest.UnexpectedException*), 1518
- Example (clase en *doctest*), 1512
- examples (atributo de *doctest.DocTest*), 1512
- exc_info (atributo de *doctest.UnexpectedException*), 1518
- exc_info() (en el módulo *sys*), 1695
- exc_msg (atributo de *doctest.Example*), 1512
- exc_type (atributo de *traceback.TracebackException*), 1749
- excel (clase en *csv*), 514
- excel_tab (clase en *csv*), 514
- except
 - sentencia, 89
- except (2to3 fixer), 1609
- excepthook() (en el módulo *sys*), 1694
- excepthook() (en el módulo *threading*), 769
- excepthook() (in module *sys*), 1209
- Exception, 90
- EXCEPTION (en el módulo *tkinter*), 1440
- exception() (en el módulo *logging*), 678
- exception() (método de *asyncio.Future*), 920
- exception() (método de *asyncio.Task*), 877
- exception() (método de *concurrent.futures.Future*), 834
- exception() (método de *logging.Logger*), 668
- exceptions
 - in CGI scripts, 1208
- EXDEV (en el módulo *errno*), 731
- exec
 - función incorporada, 11, 84, 1815
- exec (2to3 fixer), 1609
- exec() (función incorporada), 11
- exec_module() (método de *importlib.abc.InspectLoader*), 1796
- exec_module() (método de *importlib.abc.Loader*), 1793
- exec_module() (método de *importlib.abc.SourceLoader*), 1797
- exec_module() (método de *importlib.machinery.ExtensionFileLoader*), 1802
- exec_prefix (en el módulo *sys*), 1695
- execfile (2to3 fixer), 1609
- execl() (en el módulo *os*), 596
- execle() (en el módulo *os*), 596
- execlp() (en el módulo *os*), 596
- execlpe() (en el módulo *os*), 596
- executable (en el módulo *sys*), 1695
- Executable Zip Files, 1684
- Execute() (método de *msilib.View*), 1863
- execute() (método de *sqlite3.Connection*), 450
- execute() (método de *sqlite3.Cursor*), 456
- executemany() (método de *sqlite3.Connection*), 450
- executemany() (método de *sqlite3.Cursor*), 456
- executescript() (método de *sqlite3.Connection*), 450
- executescript() (método de *sqlite3.Cursor*), 457
- ExecutionLoader (clase en *importlib.abc*), 1796
- Executor (clase en *concurrent.futures*), 830
- execv() (en el módulo *os*), 596
- execve() (en el módulo *os*), 596
- execvp() (en el módulo *os*), 596
- execvpe() (en el módulo *os*), 596
- ExFileSelectBox (clase en *tkinter.tix*), 1461
- EXFULL (en el módulo *errno*), 732
- exists() (en el módulo *os.path*), 391
- exists() (método de *pathlib.Path*), 383
- exists() (método de *tkinter.ttk.Treeview*), 1453
- exists() (método de *zipfile.Path*), 490
- exit (variable incorporada), 28
- exit() (en el módulo *_thread*), 862
- exit() (en el módulo *sys*), 1695
- exit() (método de *argparse.ArgumentParser*), 661
- exitcode (atributo de *multiprocessing.Process*), 789
- exitfunc (2to3 fixer), 1609
- exitonclick() (en el módulo *turtle*), 1412
- ExitStack (clase en *contextlib*), 1734
- exp() (en el módulo *cmath*), 296
- exp() (en el módulo *math*), 291
- exp() (método de *decimal.Context*), 312
- exp() (método de *decimal.Decimal*), 305
- expand() (método de *re.Match*), 122
- expand_tabs (atributo de *textwrap.TextWrapper*), 142
- ExpandEnvironmentStrings() (en el módulo *winreg*), 1871
- expandNode() (método de *xml.dom.pulldom.DOMEventStream*), 1177

- `expandtabs()` (método de `bytearray`), 62
 - `expandtabs()` (método de `bytes`), 62
 - `expandtabs()` (método de `str`), 45
 - `expanduser()` (en el módulo `os.path`), 391
 - `expanduser()` (método de `pathlib.Path`), 383
 - `expandvars()` (en el módulo `os.path`), 391
 - `Expat`, 1189
 - `ExpatError`, 1189
 - `expect()` (método de `telnetlib.Telnet`), 1290
 - `expected` (atributo de `asyncio.IncompleteReadError`), 896
 - `expectedFailure()` (en el módulo `unittest`), 1526
 - `expectedFailures` (atributo de `unittest.TestResult`), 1541
 - `expires` (atributo de `http.cookiejar.Cookie`), 1318
 - `exploded` (atributo de `ipaddress.IPv4Address`), 1334
 - `exploded` (atributo de `ipaddress.IPv4Network`), 1339
 - `exploded` (atributo de `ipaddress.IPv6Address`), 1336
 - `exploded` (atributo de `ipaddress.IPv6Network`), 1342
 - `expm1()` (en el módulo `math`), 291
 - `expovariate()` (en el módulo `random`), 329
 - `expr()` (en el módulo `parser`), 1814
 - expresión, 1938
 - expresión generadora, 1940
 - `expunge()` (método de `imaplib.IMAP4`), 1267
 - `extend()` (método de `array.array`), 245
 - `extend()` (método de `collections.deque`), 223
 - `extend()` (método de `xml.etree.ElementTree.Element`), 1154
 - `extend()` (sequence method), 40
 - `extend_path()` (en el módulo `pkgutil`), 1783
 - `EXTENDED_ARG` (opcode), 1854
 - `ExtendedContext` (clase en decimal), 309
 - `ExtendedInterpolation` (clase en `configparser`), 522
 - `extendleft()` (método de `collections.deque`), 223
 - `EXTENSION_SUFFIXES` (en el módulo `importlib.machinery`), 1799
 - `ExtensionFileLoader` (clase en `importlib.machinery`), 1802
 - `extensions_map` (atributo de `http.server.SimpleHTTPRequestHandler`), 1306
 - External Data Representation, 423, 536
 - `external_attr` (atributo de `zipfile.ZipInfo`), 493
 - `ExternalClashError`, 1122
 - `ExternalEntityParserCreate()` (método de `xml.parsers.expat.xmlparser`), 1190
 - `ExternalEntityRefHandler()` (método de `xml.parsers.expat.xmlparser`), 1194
 - `extra` (atributo de `zipfile.ZipInfo`), 492
 - `--extract <tarfile> [output_dir]`
 - tarfile command line option, 507
 - `--extract <zipfile> <output_dir>`
 - zipfile command line option, 494
 - `extract()` (método de clase de `traceback.StackSummary`), 1750
 - `extract()` (método de `tarfile.TarFile`), 499
 - `extract()` (método de `zipfile.ZipFile`), 488
 - `extract_cookies()` (método de `http.cookiejar.CookieJar`), 1313
 - `extract_stack()` (en el módulo `traceback`), 1748
 - `extract_tb()` (en el módulo `traceback`), 1748
 - `extract_version` (atributo de `zipfile.ZipInfo`), 493
 - `extractall()` (método de `tarfile.TarFile`), 499
 - `extractall()` (método de `zipfile.ZipFile`), 488
 - `ExtractError`, 496
 - `extractfile()` (método de `tarfile.TarFile`), 500
 - `extraction_filter` (atributo de `tarfile.TarFile`), 500
 - `extsep` (en el módulo `os`), 608
- ## F
- `-f`
 - compileall command line option, 1838
 - trace command line option, 1660
 - unittest command line option, 1522
 - `f-string`, 1939
 - `f_contiguous` (atributo de `memoryview`), 75
 - `F_LOCK` (en el módulo `os`), 568
 - `F_OK` (en el módulo `os`), 576
 - `F_TEST` (en el módulo `os`), 568
 - `F_TLOCK` (en el módulo `os`), 568
 - `F_ULOCK` (en el módulo `os`), 568
 - `fabs()` (en el módulo `math`), 289
 - `factorial()` (en el módulo `math`), 289
 - `factory()` (método de clase de `importlib.util.LazyLoader`), 1806
 - `fail()` (método de `unittest.TestCase`), 1535
 - `FAIL_FAST` (en el módulo `doctest`), 1505
 - `--failfast`
 - unittest command line option, 1522
 - `failfast` (atributo de `unittest.TestResult`), 1542
 - `failureException` (atributo de `unittest.TestCase`), 1535
 - `failures` (atributo de `unittest.TestResult`), 1541
 - `FakePath` (clase en `test.support`), 1627
 - `false`, 29
 - `False`, 29, 85
 - `False` (Built-in object), 29
 - `False` (variable incorporada), 27
 - `family` (atributo de `socket.socket`), 969
 - `FancyURLopener` (clase en `urllib.request`), 1236
 - `--fast`
 - gzip command line option, 475
 - `fast` (atributo de `pickle.Pickler`), 425
 - `FastChildWatcher` (clase en `asyncio`), 937
 - `fatalError()` (método de `xml.sax.handler.ErrorHandler`), 1183
 - `Fault` (clase en `xmlrpc.client`), 1324
 - `faultCode` (atributo de `xmlrpc.client.Fault`), 1324
 - `faulthandler` (módulo), 1638
 - `faultString` (atributo de `xmlrpc.client.Fault`), 1324
 - `fchdir()` (en el módulo `os`), 578
 - `fchmod()` (en el módulo `os`), 567

- `fchown()` (en el módulo `os`), 567
- `FCICreate()` (en el módulo `msilib`), 1861
- `fcntl` (módulo), 1889
- `fcntl()` (en el módulo `fcntl`), 1890
- `fd` (atributo de `selectors.SelectorKey`), 1014
- `fd()` (en el módulo `turtle`), 1390
- `fd_count()` (en el módulo `test.support`), 1617
- `fdatasync()` (en el módulo `os`), 567
- `fdopen()` (en el módulo `os`), 566
- `Feature` (clase en `msilib`), 1865
- `feature_external_ges` (en el módulo `xml.sax.handler`), 1180
- `feature_external_pes` (en el módulo `xml.sax.handler`), 1180
- `feature_namespace_prefixes` (en el módulo `xml.sax.handler`), 1180
- `feature_namespaces` (en el módulo `xml.sax.handler`), 1179
- `feature_string_interning` (en el módulo `xml.sax.handler`), 1180
- `feature_validation` (en el módulo `xml.sax.handler`), 1180
- `feed()` (método de `email.parser.BytesFeedParser`), 1047
- `feed()` (método de `html.parser.HTMLParser`), 1137
- `feed()` (método de `xml.etree.ElementTree.XMLParser`), 1158
- `feed()` (método de `xml.etree.ElementTree.XMLPullParser`), 1160
- `feed()` (método de `xml.sax.xmlreader.IncrementalParser`), 1187
- `FeedParser` (clase en `email.parser`), 1047
- `fetch()` (método de `imaplib.IMAP4`), 1268
- `Fetch()` (método de `msilib.View`), 1863
- `fetchall()` (método de `sqlite3.Cursor`), 458
- `fetchmany()` (método de `sqlite3.Cursor`), 458
- `fetchone()` (método de `sqlite3.Cursor`), 458
- `fflags` (atributo de `select.kevent`), 1013
- `Field` (clase en `dataclasses`), 1724
- `field()` (en el módulo `dataclasses`), 1723
- `field_size_limit()` (en el módulo `csv`), 513
- `fieldnames` (atributo de `csv.csvreader`), 516
- `fields` (atributo de `uuid.UUID`), 1292
- `fields()` (en el módulo `dataclasses`), 1725
- `file`
 - byte-code, 1836, 1928
 - configuration, 518
 - copying, 410
 - debugger configuration, 1642
 - gzip command line option, 475
 - .ini, 518
 - large files, 1881
 - `mime.types`, 1125
 - modes, 17
 - path configuration, 1774
 - .pdbrc, 1642
 - plist, 539
 - temporary, 403
- `file ...`
 - compileall command line option, 1838
- `file` (atributo de `pyclbr.Class`), 1836
- `file` (atributo de `pyclbr.Function`), 1835
- `file control`
 - UNIX, 1889
- `file name`
 - temporary, 403
- `file object`
 - io module, 610
 - `open()` built-in function, 16
- `--file=<file>`
 - trace command line option, 1660
- `FILE_ATTRIBUTE_ARCHIVE` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_COMPRESSED` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_DEVICE` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_DIRECTORY` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_ENCRYPTED` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_HIDDEN` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_INTEGRITY_STREAM` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_NO_SCRUB_DATA` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_NORMAL` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_NOT_CONTENT_INDEXED` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_OFFLINE` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_READONLY` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_REPARSE_POINT` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_SPARSE_FILE` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_SYSTEM` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_TEMPORARY` (en el módulo `stat`), 401
- `FILE_ATTRIBUTE_VIRTUAL` (en el módulo `stat`), 401
- `file_dispatcher` (clase en `asyncore`), 1020
- `file_open()` (método de `urllib.request.FileHandler`), 1231
- `file_size` (atributo de `zipfile.ZipInfo`), 493
- `file_wrapper` (clase en `asyncore`), 1020
- `filecmp` (módulo), 401
- `fileConfig()` (en el módulo `logging.config`), 682
- `FileCookieJar` (clase en `http.cookiejar`), 1312
- `FileEntry` (clase en `tkinter.tix`), 1461
- `FileExistsError`, 95
- `FileFinder` (clase en `importlib.machinery`), 1801
- `FileHandler` (clase en `logging`), 693
- `FileHandler` (clase en `urllib.request`), 1224

- `FileInput` (clase en `fileinput`), 395
- `fileinput` (módulo), 394
- `FileIO` (clase en `io`), 616
- `filelineno()` (en el módulo `fileinput`), 395
- `FileLoader` (clase en `importlib.abc`), 1796
- `filemode()` (en el módulo `stat`), 398
- `filename` (atributo de `doctest.DocTest`), 1512
- `filename` (atributo de `http.cookiejar.FileCookieJar`), 1314
- `filename` (atributo de `OSError`), 92
- `filename` (atributo de `SyntaxError`), 93
- `filename` (atributo de `traceback.TracebackException`), 1749
- `filename` (atributo de `tracemalloc.Frame`), 1668
- `filename` (atributo de `zipfile.ZipFile`), 490
- `filename` (atributo de `zipfile.ZipInfo`), 492
- `filename()` (en el módulo `fileinput`), 395
- `filename2` (atributo de `OSError`), 92
- `filename_only` (en el módulo `tabnanny`), 1834
- `filename_pattern` (atributo de `tracemalloc.Filter`), 1668
- `filenames`
 - `pathname expansion`, 407
 - `wildcard expansion`, 409
- `fileno()` (en el módulo `fileinput`), 395
- `fileno()` (método de `http.client.HTTPResponse`), 1255
- `fileno()` (método de `io.IOBase`), 613
- `fileno()` (método de `multiprocessing.connection.Connection`), 796
- `fileno()` (método de `ossaudiodev.oss_audio_device`), 1362
- `fileno()` (método de `ossaudiodev.oss_mixer_device`), 1365
- `fileno()` (método de `select.devpoll`), 1009
- `fileno()` (método de `select.epoll`), 1010
- `fileno()` (método de `select.kqueue`), 1012
- `fileno()` (método de `selectors.DevpollSelector`), 1016
- `fileno()` (método de `selectors.EpollSelector`), 1016
- `fileno()` (método de `selectors.KqueueSelector`), 1016
- `fileno()` (método de `socketserver.BaseServer`), 1296
- `fileno()` (método de `socket.socket`), 963
- `fileno()` (método de `telnetlib.Telnet`), 1290
- `FileNotFoundError`, 95
- `fileobj` (atributo de `selectors.SelectorKey`), 1014
- `FileSelectBox` (clase en `tkinter.tix`), 1461
- `FileType` (clase en `argparse`), 657
- `FileWrapper` (clase en `wsgiref.util`), 1211
- `fill()` (en el módulo `textwrap`), 140
- `fill()` (método de `textwrap.TextWrapper`), 143
- `fillcolor()` (en el módulo `turtle`), 1398
- `filling()` (en el módulo `turtle`), 1399
- `filter` (2to3 fixer), 1609
- `filter` (atributo de `select.kevent`), 1012
- `Filter` (clase en `logging`), 672
- `Filter` (clase en `tracemalloc`), 1667
- `--filter <filtername>`
 - `tarfile` command line option, 507
- `filter()` (en el módulo `curses`), 707
- `filter()` (en el módulo `fnmatch`), 409
- `filter()` (función incorporada), 11
- `filter()` (método de `logging.Filter`), 672
- `filter()` (método de `logging.Handler`), 670
- `filter()` (método de `logging.Logger`), 668
- `FILTER_DIR` (en el módulo `unittest.mock`), 1582
- `filter_traces()` (método de `tracemalloc.Snapshot`), 1668
- `FilterError`, 496
- `filterfalse()` (en el módulo `itertools`), 348
- `filterwarnings()` (en el módulo `warnings`), 1720
- `Final` (en el módulo `typing`), 1495
- `final()` (en el módulo `typing`), 1492
- `finalize` (clase en `weakref`), 249
- `find()` (en el módulo `gettext`), 1369
- `find()` (método de `bytearray`), 58
- `find()` (método de `bytes`), 58
- `find()` (método de `doctest.DocTestFinder`), 1513
- `find()` (método de `mmap.mmap`), 1033
- `find()` (método de `str`), 46
- `find()` (método de `xml.etree.ElementTree.Element`), 1154
- `find()` (método de `xml.etree.ElementTree.ElementTree`), 1156
- `find_class()` (método de `pickle.Unpickler`), 426
- `find_class()` (`pickle` protocol), 436
- `find_library()` (en el módulo `ctypes.util`), 761
- `find_loader()` (en el módulo `importlib`), 1790
- `find_loader()` (en el módulo `pkgutil`), 1784
- `find_loader()` (método de `importlib.abc.PathEntryFinder`), 1793
- `find_loader()` (método de `importlib.machinery.FileFinder`), 1801
- `find_longest_match()` (método de `difflib.SequenceMatcher`), 135
- `find_module()` (en el módulo `imp`), 1928
- `find_module()` (método de clase de `importlib.machinery.PathFinder`), 1800
- `find_module()` (método de `imp.NullImporter`), 1932
- `find_module()` (método de `importlib.abc.Finder`), 1792
- `find_module()` (método de `importlib.abc.MetaPathFinder`), 1792
- `find_module()` (método de `importlib.abc.PathEntryFinder`), 1793
- `find_module()` (método de `zipimport.zipimporter`), 1782
- `find_msvcrt()` (en el módulo `ctypes.util`), 761
- `find_spec()` (en el módulo `importlib.util`), 1804
- `find_spec()` (método de clase de `importlib.machinery.PathFinder`), 1800
- `find_spec()` (método de `importlib.abc.MetaPathFinder`), 1792
- `find_spec()` (método de `importlib.abc.PathEntryFinder`), 1793
- `find_spec()` (método de `importlib.machinery.FileFinder`), 1801
- `find_unused_port()` (en el módulo `test.support`),

- 1624
- `find_user_password()` (método de `urllib.request.HTTPPasswordMgr`), 1229
- `find_user_password()` (método de `urllib.request.HTTPPasswordMgrWithPriorAuth`), 1230
- `findall()` (en el módulo `re`), 118
- `findall()` (método de `re.Pattern`), 121
- `findall()` (método de `xml.etree.ElementTree.Element`), 1154
- `findall()` (método de `xml.etree.ElementTree.ElementTree`), 1156
- `findCaller()` (método de `logging.Logger`), 668
- `Finder` (clase en `importlib.abc`), 1792
- `findfactor()` (en el módulo `audioop`), 1348
- `findfile()` (en el módulo `test.support`), 1617
- `findfit()` (en el módulo `audioop`), 1348
- `finditer()` (en el módulo `re`), 118
- `finditer()` (método de `re.Pattern`), 121
- `findlabels()` (en el módulo `dis`), 1844
- `findlinestarts()` (en el módulo `dis`), 1844
- `findmatch()` (en el módulo `mailcap`), 1105
- `findmax()` (en el módulo `audioop`), 1348
- `findtext()` (método de `xml.etree.ElementTree.Element`), 1154
- `findtext()` (método de `xml.etree.ElementTree.ElementTree`), 1156
- `finish()` (método de `socketserver.BaseRequestHandler`), 1298
- `finish_request()` (método de `socketserver.BaseServer`), 1297
- `firstChild` (atributo de `xml.dom.Node`), 1164
- `firstkey()` (método de `dbm.gnu.gdbm`), 444
- `firstweekday()` (en el módulo `calendar`), 216
- `fix_missing_locations()` (en el módulo `ast`), 1822
- `fix_sentence_endings` (atributo de `textwrap.TextWrapper`), 142
- `Flag` (clase en `enum`), 267
- `flag_bits` (atributo de `zipfile.ZipInfo`), 493
- `flags` (atributo de `re.Pattern`), 121
- `flags` (atributo de `select.kevent`), 1012
- `flags` (en el módulo `sys`), 1696
- `flash()` (en el módulo `curses`), 707
- `flatten()` (método de `email.generator.BytesGenerator`), 1051
- `flatten()` (método de `email.generator.Generator`), 1052
- `flattening` objects, 421
- `float` función incorporada, 31
- `float` (clase incorporada), 11
- `float_info` (en el módulo `sys`), 1696
- `float_repr_style` (en el módulo `sys`), 1697
- `floating point` literals, 31
- `objeto`, 31
- `FloatingPointError`, 90
- `FloatOperation` (clase en `decimal`), 316
- `flock()` (en el módulo `fcntl`), 1890
- `floor()` (en el módulo `math`), 289
- `floor()` (in module `math`), 32
- `floordiv()` (en el módulo `operator`), 366
- `flush()` (método de `bz2.BZ2Compressor`), 477
- `flush()` (método de `formatter.writer`), 1859
- `flush()` (método de `io.BufferedWriter`), 618
- `flush()` (método de `io.IOBase`), 613
- `flush()` (método de `logging.Handler`), 670
- `flush()` (método de `logging.handlers.BufferingHandler`), 701
- `flush()` (método de `logging.handlers.MemoryHandler`), 702
- `flush()` (método de `logging.StreamHandler`), 692
- `flush()` (método de `lzma.LZMACompressor`), 482
- `flush()` (método de `mailbox.Mailbox`), 1109
- `flush()` (método de `mailbox.Maildir`), 1111
- `flush()` (método de `mailbox.MH`), 1113
- `flush()` (método de `mmap.mmap`), 1033
- `flush()` (método de `xml.etree.ElementTree.XMLParser`), 1159
- `flush()` (método de `xml.etree.ElementTree.XMLPullParser`), 1160
- `flush()` (método de `zlib.Compress`), 471
- `flush()` (método de `zlib.Decompress`), 472
- `flush_headers()` (método de `http.server.BaseHTTPRequestHandler`), 1305
- `flush_softspace()` (método de `formatter.formatter`), 1858
- `flushinp()` (en el módulo `curses`), 707
- `FlushKey()` (en el módulo `winreg`), 1871
- `fma()` (método de `decimal.Context`), 312
- `fma()` (método de `decimal.Decimal`), 305
- `fmean()` (en el módulo `statistics`), 334
- `fmod()` (en el módulo `math`), 289
- `FMT_BINARY` (en el módulo `plistlib`), 541
- `FMT_XML` (en el módulo `plistlib`), 541
- `fnmatch` (módulo), 409
- `fnmatch()` (en el módulo `fnmatch`), 409
- `fnmatchcase()` (en el módulo `fnmatch`), 409
- `focus()` (método de `tkinter.ttk.Treeview`), 1453
- `fold` (atributo de `datetime.datetime`), 190
- `fold` (atributo de `datetime.time`), 198
- `fold()` (método de `email.headerregistry.BaseHeader`), 1062
- `fold()` (método de `email.policy.Compat32`), 1059
- `fold()` (método de `email.policy.EmailPolicy`), 1058
- `fold()` (método de `email.policy.Policy`), 1057
- `fold_binary()` (método de `email.policy.Compat32`), 1060
- `fold_binary()` (método de `email.policy.EmailPolicy`), 1058
- `fold_binary()` (método de `email.policy.Policy`), 1057

- `FOR_ITER` (*opcode*), 1852
- `forget()` (*en el módulo test.support*), 1616
- `forget()` (*método de tkinter.ttk.Notebook*), 1447
- `fork()` (*en el módulo os*), 598
- `fork()` (*en el módulo pty*), 1888
- `ForkingMixIn` (*clase en socketserver*), 1295
- `ForkingTCPServer` (*clase en socketserver*), 1296
- `ForkingUDPServer` (*clase en socketserver*), 1296
- `forkpty()` (*en el módulo os*), 599
- `Form` (*clase en tkinter.tix*), 1462
- `format` (*atributo de memoryview*), 74
- `format` (*atributo de multiprocessing.shared_memory.ShareableList*), 828
- `format` (*atributo de struct.Struct*), 158
- `format()` (*en el módulo locale*), 1380
- `format()` (*función incorporada*), 12
- `format()` (*método de logging.Formatter*), 671
- `format()` (*método de logging.Handler*), 670
- `format()` (*método de pprint.PrettyPrinter*), 261
- `format()` (*método de str*), 46
- `format()` (*método de string.Formatter*), 100
- `format()` (*método de traceback.StackSummary*), 1750
- `format()` (*método de traceback.TracebackException*), 1750
- `format()` (*método de tracemalloc.Traceback*), 1670
- `format_datetime()` (*en el módulo email.utils*), 1094
- `format_exc()` (*en el módulo traceback*), 1748
- `format_exception()` (*en el módulo traceback*), 1748
- `format_exception_only()` (*en el módulo traceback*), 1748
- `format_exception_only()` (*método de traceback.TracebackException*), 1750
- `format_field()` (*método de string.Formatter*), 101
- `format_help()` (*método de argparse.ArgumentParser*), 660
- `format_list()` (*en el módulo traceback*), 1748
- `format_map()` (*método de str*), 46
- `format_stack()` (*en el módulo traceback*), 1749
- `format_stack_entry()` (*método de bdb.Bdb*), 1637
- `format_string()` (*en el módulo locale*), 1380
- `format_tb()` (*en el módulo traceback*), 1748
- `format_usage()` (*método de argparse.ArgumentParser*), 660
- `FORMAT_VALUE` (*opcode*), 1854
- `formataddr()` (*en el módulo email.utils*), 1093
- `formatargspec()` (*en el módulo inspect*), 1768
- `formatargvalues()` (*en el módulo inspect*), 1768
- `formatdate()` (*en el módulo email.utils*), 1094
- `FormatError`, 1122
- `FormatError()` (*en el módulo ctypes*), 761
- `FormatException()` (*método de logging.Formatter*), 672
- `formatmonth()` (*método de calendar.HTMLCalendar*), 214
- `formatmonth()` (*método de calendar.TextCalendar*), 214
- `formatStack()` (*método de logging.Formatter*), 672
- `Formatter` (*clase en logging*), 671
- `Formatter` (*clase en string*), 100
- `formatter` (*módulo*), 1857
- `formatTime()` (*método de logging.Formatter*), 671
- `formatting`
 - `bytearray` (%), 66
 - `bytes` (%), 66
- `formatting, string` (%), 52
- `formatwarning()` (*en el módulo warnings*), 1720
- `formatyear()` (*método de calendar.HTMLCalendar*), 214
- `formatyear()` (*método de calendar.TextCalendar*), 214
- `formatyearpage()` (*método de calendar.HTMLCalendar*), 214
- `Fortran contiguous`, 1937
- `forward()` (*en el módulo turtle*), 1390
- `ForwardRef` (*clase en typing*), 1491
- `found_terminator()` (*método de asyncio.async_chat*), 1022
- `fpathconf()` (*en el módulo os*), 567
- `fqdn` (*atributo de smtpd.SMTPChannel*), 1287
- `Fraction` (*clase en fractions*), 324
- `fractions` (*módulo*), 324
- `frame` (*atributo de tkinter.scrolledtext.ScrolledText*), 1463
- `Frame` (*clase en tracemalloc*), 1668
- `FrameSummary` (*clase en traceback*), 1751
- `FrameType` (*en el módulo types*), 256
- `freeze()` (*en el módulo gc*), 1756
- `freeze_support()` (*en el módulo multiprocessing*), 794
- `frexp()` (*en el módulo math*), 289
- `from_address()` (*método de ctypes._CData*), 763
- `from_buffer()` (*método de ctypes._CData*), 762
- `from_buffer_copy()` (*método de ctypes._CData*), 763
- `from_bytes()` (*método de clase de int*), 33
- `from_callable()` (*método de clase de inspect.Signature*), 1764
- `from_decimal()` (*método de fractions.Fraction*), 325
- `from_exception()` (*método de clase de traceback.TracebackException*), 1750
- `from_file()` (*método de clase de zipfile.ZipInfo*), 492
- `from_float()` (*método de decimal.Decimal*), 305
- `from_float()` (*método de fractions.Fraction*), 325
- `from_iterable()` (*método de clase de itertools.chain*), 346
- `from_list()` (*método de clase de traceback.StackSummary*), 1750
- `from_param()` (*método de ctypes._CData*), 763
- `from_samples()` (*método de clase de statistics.NormalDist*), 339
- `from_traceback()` (*método de clase de dis.Bytecode*), 1842

- `frombuf()` (método de clase de `tarfile.TarInfo`), 501
 - `frombytes()` (método de `array.array`), 245
 - `fromfd()` (en el módulo `socket`), 958
 - `fromfd()` (método de `select.epoll`), 1010
 - `fromfd()` (método de `select.kqueue`), 1012
 - `fromfile()` (método de `array.array`), 245
 - `fromhex()` (método de clase de `bytearray`), 56
 - `fromhex()` (método de clase de `bytes`), 55
 - `fromhex()` (método de clase de `float`), 34
 - `fromisocalendar()` (método de clase de `datetime.date`), 183
 - `fromisocalendar()` (método de clase de `datetime.datetime`), 189
 - `fromisoformat()` (método de clase de `datetime.date`), 183
 - `fromisoformat()` (método de clase de `datetime.datetime`), 189
 - `fromisoformat()` (método de clase de `datetime.time`), 198
 - `fromkeys()` (método de clase de `dict`), 79
 - `fromkeys()` (método de `collections.Counter`), 221
 - `fromlist()` (método de `array.array`), 245
 - `fromordinal()` (método de clase de `datetime.date`), 183
 - `fromordinal()` (método de clase de `datetime.datetime`), 189
 - `fromshare()` (en el módulo `socket`), 958
 - `fromstring()` (en el módulo `xml.etree.ElementTree`), 1150
 - `fromstring()` (método de `array.array`), 245
 - `fromstringlist()` (en el módulo `xml.etree.ElementTree`), 1150
 - `fromtarfile()` (método de clase de `tarfile.TarInfo`), 501
 - `fromtimestamp()` (método de clase de `datetime.date`), 183
 - `fromtimestamp()` (método de clase de `datetime.datetime`), 188
 - `fromunicode()` (método de `array.array`), 245
 - `fromutc()` (método de `datetime.timezone`), 208
 - `fromutc()` (método de `datetime.tzinfo`), 202
 - `FrozenImporter` (clase en `importlib.machinery`), 1800
 - `FrozenInstanceError`, 1729
 - `FrozenSet` (clase en `typing`), 1487
 - `frozenset` (clase incorporada), 75
 - `fs_is_case_insensitive()` (en el módulo `test.support`), 1625
 - `FS_NONASCII` (en el módulo `test.support`), 1615
 - `fsdecode()` (en el módulo `os`), 561
 - `fsencode()` (en el módulo `os`), 561
 - `fspath()` (en el módulo `os`), 561
 - `fstat()` (en el módulo `os`), 568
 - `fstatvfs()` (en el módulo `os`), 568
 - `fsum()` (en el módulo `math`), 289
 - `fsync()` (en el módulo `os`), 568
 - `FTP`, 1237
 - `ftplib` (standard module), 1257
 - `protocol`, 1237, 1257
 - `FTP` (clase en `ftplib`), 1257
 - `ftp_open()` (método de `urllib.request.FTPHandler`), 1231
 - `FTP_TLS` (clase en `ftplib`), 1258
 - `FTPHandler` (clase en `urllib.request`), 1224
 - `ftplib` (módulo), 1257
 - `ftruncate()` (en el módulo `os`), 568
 - `Full`, 856
 - `full()` (método de `asyncio.Queue`), 894
 - `full()` (método de `multiprocessing.Queue`), 792
 - `full()` (método de `queue.Queue`), 856
 - `full_url` (atributo de `urllib.request.Request`), 1225
 - `fullmatch()` (en el módulo `re`), 117
 - `fullmatch()` (método de `re.Pattern`), 121
 - `fully_trusted_filter()` (en el módulo `tarfile`), 504
 - `func` (atributo de `functools.partial`), 365
 - `funcattrs` (2to3 fixer), 1609
 - función, 1939
 - función clave, 1941
 - función corrutina, 1937
 - función genérica, 1940
 - función incorporada
 - `compile`, 84, 255, 1816
 - `complex`, 31
 - `eval`, 84, 261, 1815
 - `exec`, 11, 84, 1815
 - `float`, 31
 - `hash`, 39
 - `int`, 31
 - `len`, 38, 77
 - `max`, 38
 - `min`, 38
 - `slice`, 1854
 - `type`, 84
 - `Function` (clase en `symtable`), 1825
 - `FunctionTestCase` (clase en `unittest`), 1537
 - `FunctionType` (en el módulo `types`), 254
 - `functools` (módulo), 357
 - `funny_files` (atributo de `filecmp.dircmp`), 403
 - `future` (2to3 fixer), 1609
 - `Future` (clase en `asyncio`), 919
 - `Future` (clase en `concurrent.futures`), 834
 - `FutureWarning`, 96
 - `fwalk()` (en el módulo `os`), 593
- ## G
- `-g`
 - `trace` command line option, 1660
 - `G.722`, 1352
 - `gaierror`, 953
 - `gamma()` (en el módulo `math`), 293
 - `gammavariate()` (en el módulo `random`), 329
 - `gancho` a entrada de ruta, 1944
 - `garbage` (en el módulo `gc`), 1757
 - `gather()` (en el módulo `asyncio`), 871
 - `gather()` (método de `curses.textpad.Textbox`), 723

- `gauss()` (en el módulo *random*), 329
- `gc` (módulo), 1754
- `gc_collect()` (en el módulo *test.support*), 1620
- `gcd()` (en el módulo *fractions*), 326
- `gcd()` (en el módulo *math*), 290
- `ge()` (en el módulo *operator*), 365
- `gen_uuid()` (en el módulo *msilib*), 1862
- `generador`, 1940
- `generador asincrónico`, 1936
- `generate_tokens()` (en el módulo *tokenize*), 1830
- `generator`, 1939
- `Generator` (clase en *collections.abc*), 235
- `Generator` (clase en *email.generator*), 1051
- `Generator` (clase en *typing*), 1488
- `generator expression`, 1940
- `GeneratorExit`, 91
- `GeneratorType` (en el módulo *types*), 254
- `Generic` (clase en *typing*), 1484
- `generic_visit()` (método de *ast.NodeVisitor*), 1823
- `genops()` (en el módulo *pickletools*), 1856
- `geometric_mean()` (en el módulo *statistics*), 334
- `get()` (en el módulo *webbrowser*), 1200
- `get()` (método de *asyncio.Queue*), 894
- `get()` (método de *configparser.ConfigParser*), 532
- `get()` (método de *contextvars.Context*), 861
- `get()` (método de *contextvars.ContextVar*), 859
- `get()` (método de *dict*), 79
- `get()` (método de *email.message.EmailMessage*), 1041
- `get()` (método de *email.message.Message*), 1079
- `get()` (método de *mailbox.Mailbox*), 1108
- `get()` (método de *multiprocessing.pool.AsyncResult*), 811
- `get()` (método de *multiprocessing.Queue*), 792
- `get()` (método de *multiprocessing.SimpleQueue*), 793
- `get()` (método de *ossaudiodev.oss_mixer_device*), 1365
- `get()` (método de *queue.Queue*), 857
- `get()` (método de *queue.SimpleQueue*), 858
- `get()` (método de *tkinter.ttk.Combobox*), 1444
- `get()` (método de *tkinter.ttk.Spinbox*), 1445
- `get()` (método de *types.MappingProxyType*), 257
- `get()` (método de *xml.etree.ElementTree.Element*), 1154
- `GET_AITER` (opcode), 1847
- `get_all()` (método de *email.message.EmailMessage*), 1041
- `get_all()` (método de *email.message.Message*), 1079
- `get_all()` (método de *wsgiref.headers.Headers*), 1212
- `get_all_breaks()` (método de *bdb.Bdb*), 1636
- `get_all_start_methods()` (en el módulo *multiprocessing*), 794
- `GET_ANEXT` (opcode), 1847
- `get_app()` (método de *wsgi-ref.simple_server.WSGIServer*), 1213
- `get_archive_formats()` (en el módulo *shutil*), 417
- `get_args()` (en el módulo *typing*), 1491
- `get_asyncgen_hooks()` (en el módulo *sys*), 1700
- `get_attribute()` (en el módulo *test.support*), 1623
- `GET_AWAITABLE` (opcode), 1847
- `get_begidx()` (en el módulo *readline*), 149
- `get_blocking()` (en el módulo *os*), 568
- `get_body()` (método de *email.message.EmailMessage*), 1044
- `get_body_encoding()` (método de *email.charset.Charset*), 1090
- `get_boundary()` (método de *email.message.EmailMessage*), 1042
- `get_boundary()` (método de *email.message.Message*), 1082
- `get_bpbynumber()` (método de *bdb.Bdb*), 1636
- `get_break()` (método de *bdb.Bdb*), 1636
- `get_breaks()` (método de *bdb.Bdb*), 1636
- `get_buffer()` (método de *asyncio.BufferedProtocol*), 928
- `get_buffer()` (método de *xdr.lib.Packer*), 537
- `get_buffer()` (método de *xdr.lib.Unpacker*), 538
- `get_bytes()` (método de *mailbox.Mailbox*), 1108
- `get_ca_certs()` (método de *ssl.SSLContext*), 991
- `get_cache_token()` (en el módulo *abc*), 1745
- `get_channel_binding()` (método de *ssl.SSLSocket*), 988
- `get_charset()` (método de *email.message.Message*), 1078
- `get_charsets()` (método de *email.message.EmailMessage*), 1043
- `get_charsets()` (método de *email.message.Message*), 1082
- `get_child_watcher()` (en el módulo *asyncio*), 936
- `get_child_watcher()` (método de *asyncio.AbstractEventLoopPolicy*), 935
- `get_children()` (método de *symtable.SymbolTable*), 1825
- `get_children()` (método de *tkinter.ttk.Treeview*), 1452
- `get_ciphers()` (método de *ssl.SSLContext*), 991
- `get_clock_info()` (en el módulo *time*), 624
- `get_close_matches()` (en el módulo *difflib*), 132
- `get_code()` (método de *importlib.abc.InspectLoader*), 1795
- `get_code()` (método de *importlib.abc.SourceLoader*), 1797
- `get_code()` (método de *importlib.machinery.ExtensionFileLoader*), 1802
- `get_code()` (método de *importlib.machinery.SourcelessFileLoader*), 1802
- `get_code()` (método de *zipimport.zipimporter*), 1782
- `get_completer()` (en el módulo *readline*), 149
- `get_completer_delims()` (en el módulo *readline*), 150
- `get_completion_type()` (en el módulo *readline*), 149
- `get_config_h_filename()` (en el módulo *sysconfig*), 1713

- `get_config_var()` (en el módulo `sysconfig`), 1711
`get_config_vars()` (en el módulo `sysconfig`), 1711
`get_content()` (en el módulo `email.contentmanager`), 1068
`get_content()` (método de `email.contentmanager.ContentManager`), 1067
`get_content()` (método de `email.message.EmailMessage`), 1044
`get_content_charset()` (método de `email.message.EmailMessage`), 1043
`get_content_charset()` (método de `email.message.Message`), 1082
`get_content_disposition()` (método de `email.message.EmailMessage`), 1043
`get_content_disposition()` (método de `email.message.Message`), 1082
`get_content_maintype()` (método de `email.message.EmailMessage`), 1042
`get_content_maintype()` (método de `email.message.Message`), 1080
`get_content_subtype()` (método de `email.message.EmailMessage`), 1042
`get_content_subtype()` (método de `email.message.Message`), 1080
`get_content_type()` (método de `email.message.EmailMessage`), 1041
`get_content_type()` (método de `email.message.Message`), 1080
`get_context()` (en el módulo `multiprocessing`), 795
`get_coro()` (método de `asyncio.Task`), 877
`get_coroutine_origin_tracking_depth()` (en el módulo `sys`), 1700
`get_count()` (en el módulo `gc`), 1755
`get_current_history_length()` (en el módulo `readline`), 148
`get_data()` (en el módulo `pkgutil`), 1785
`get_data()` (método de `importlib.abc.FileLoader`), 1796
`get_data()` (método de `importlib.abc.ResourceLoader`), 1795
`get_data()` (método de `zipimport.zipimporter`), 1782
`get_date()` (método de `mailbox.MaildirMessage`), 1116
`get_debug()` (en el módulo `gc`), 1755
`get_debug()` (método de `asyncio.loop`), 912
`get_default()` (método de `argparse.ArgumentParser`), 659
`get_default_domain()` (en el módulo `nis`), 1897
`get_default_type()` (método de `email.message.EmailMessage`), 1042
`get_default_type()` (método de `email.message.Message`), 1080
`get_default_verify_paths()` (en el módulo `ssl`), 978
`get_dialect()` (en el módulo `csv`), 513
`get_docstring()` (en el módulo `ast`), 1822
`get_doctest()` (método de `doctest.DocTestParser`), 1514
`get_endidx()` (en el módulo `readline`), 149
`get_environ()` (método de `wsgiref.simple_server.WSGIRequestHandler`), 1213
`get_errno()` (en el módulo `ctypes`), 761
`get_event_loop()` (en el módulo `asyncio`), 897
`get_event_loop()` (método de `asyncio.AbstractEventLoopPolicy`), 935
`get_event_loop_policy()` (en el módulo `asyncio`), 935
`get_examples()` (método de `doctest.DocTestParser`), 1514
`get_exception_handler()` (método de `asyncio.loop`), 911
`get_exec_path()` (en el módulo `os`), 562
`get_extra_info()` (método de `asyncio.BaseTransport`), 923
`get_extra_info()` (método de `asyncio.StreamWriter`), 882
`get_field()` (método de `string.Formatter`), 100
`get_file()` (método de `mailbox.Babyl`), 1114
`get_file()` (método de `mailbox.Mailbox`), 1108
`get_file()` (método de `mailbox.Maildir`), 1111
`get_file()` (método de `mailbox.mbox`), 1111
`get_file()` (método de `mailbox.MH`), 1113
`get_file()` (método de `mailbox.MMDF`), 1114
`get_file_breaks()` (método de `bdb.Bdb`), 1636
`get_filename()` (método de `email.message.EmailMessage`), 1042
`get_filename()` (método de `email.message.Message`), 1082
`get_filename()` (método de `importlib.abc.ExecutionLoader`), 1796
`get_filename()` (método de `importlib.abc.FileLoader`), 1796
`get_filename()` (método de `importlib.machinery.ExtensionFileLoader`), 1802
`get_filename()` (método de `zipimport.zipimporter`), 1782
`get_flags()` (método de `mailbox.MaildirMessage`), 1116
`get_flags()` (método de `mailbox.mboxMessage`), 1117
`get_flags()` (método de `mailbox.MMDFMessage`), 1121
`get_folder()` (método de `mailbox.Maildir`), 1110
`get_folder()` (método de `mailbox.MH`), 1112
`get_frees()` (método de `symtable.Function`), 1825
`get_freeze_count()` (en el módulo `gc`), 1756
`get_from()` (método de `mailbox.mboxMessage`), 1117
`get_from()` (método de `mailbox.MMDFMessage`), 1121
`get_full_url()` (método de `urllib.request.Request`), 1226
`get_globals()` (método de `symtable.Function`), 1825
`get_grouped_opcodes()` (método de `dif-`

- flib.SequenceMatcher*), 136
- `get_handle_inheritable()` (en el módulo *os*), 575
- `get_header()` (método de *urllib.request.Request*), 1226
- `get_history_item()` (en el módulo *readline*), 148
- `get_history_length()` (en el módulo *readline*), 148
- `get_id()` (método de *symtable.SymbolTable*), 1824
- `get_ident()` (en el módulo *_thread*), 863
- `get_ident()` (en el módulo *threading*), 770
- `get_identifiers()` (método de *symtable.SymbolTable*), 1824
- `get_importer()` (en el módulo *pkgutil*), 1784
- `get_info()` (método de *mailbox.MaildirMessage*), 1116
- `get_inheritable()` (en el módulo *os*), 575
- `get_inheritable()` (método de *socket.socket*), 963
- `get_instructions()` (en el módulo *dis*), 1844
- `get_int_max_str_digits()` (en el módulo *sys*), 1698
- `get_interpreter()` (en el módulo *zipapp*), 1685
- `GET_ITER` (opcode), 1845
- `get_key()` (método de *selectors.BaseSelector*), 1016
- `get_labels()` (método de *mailbox.Babyl*), 1113
- `get_labels()` (método de *mailbox.BabylMessage*), 1119
- `get_last_error()` (en el módulo *ctypes*), 761
- `get_line_buffer()` (en el módulo *readline*), 148
- `get_lineno()` (método de *symtable.SymbolTable*), 1824
- `get_loader()` (en el módulo *pkgutil*), 1784
- `get_locals()` (método de *symtable.Function*), 1825
- `get_logger()` (en el módulo *multiprocessing*), 815
- `get_loop()` (método de *asyncio.Future*), 920
- `get_loop()` (método de *asyncio.Server*), 914
- `get_magic()` (en el módulo *imp*), 1928
- `get_makefile_filename()` (en el módulo *sysconfig*), 1713
- `get_map()` (método de *selectors.BaseSelector*), 1016
- `get_matching_blocks()` (método de *flib.SequenceMatcher*), 135
- `get_message()` (método de *mailbox.Mailbox*), 1108
- `get_method()` (método de *urllib.request.Request*), 1225
- `get_methods()` (método de *symtable.Class*), 1825
- `get_mixed_type_key()` (en el módulo *ipaddress*), 1345
- `get_name()` (método de *asyncio.Task*), 877
- `get_name()` (método de *symtable.Symbol*), 1825
- `get_name()` (método de *symtable.SymbolTable*), 1824
- `get_namespace()` (método de *symtable.Symbol*), 1826
- `get_namespaces()` (método de *symtable.Symbol*), 1826
- `get_native_id()` (en el módulo *_thread*), 863
- `get_native_id()` (en el módulo *threading*), 770
- `get_nonlocals()` (método de *symtable.Function*), 1825
- `get_nonstandard_attr()` (método de *http.cookiejar.Cookie*), 1318
- `get_nowait()` (método de *asyncio.Queue*), 894
- `get_nowait()` (método de *multiprocessing.Queue*), 793
- `get_nowait()` (método de *queue.Queue*), 857
- `get_nowait()` (método de *queue.SimpleQueue*), 858
- `get_object_traceback()` (en el módulo *tracemalloc*), 1666
- `get_objects()` (en el módulo *gc*), 1755
- `get_opcodes()` (método de *flib.SequenceMatcher*), 135
- `get_option()` (método de *optparse.OptionParser*), 1919
- `get_option_group()` (método de *optparse.OptionParser*), 1910
- `get_origin()` (en el módulo *typing*), 1491
- `get_original_stdout()` (en el módulo *test.support*), 1618
- `get_osfhandle()` (en el módulo *msvcrt*), 1867
- `get_output_charset()` (método de *email.charset.Charset*), 1090
- `get_param()` (método de *email.message.Message*), 1081
- `get_parameters()` (método de *symtable.Function*), 1825
- `get_params()` (método de *email.message.Message*), 1080
- `get_path()` (en el módulo *sysconfig*), 1712
- `get_path_names()` (en el módulo *sysconfig*), 1712
- `get_paths()` (en el módulo *sysconfig*), 1712
- `get_payload()` (método de *email.message.Message*), 1077
- `get_pid()` (método de *asyncio.SubprocessTransport*), 925
- `get_pipe_transport()` (método de *asyncio.SubprocessTransport*), 925
- `get_platform()` (en el módulo *sysconfig*), 1712
- `get_poly()` (en el módulo *turtle*), 1404
- `get_position()` (método de *xdrlib.Unpacker*), 538
- `get_protocol()` (método de *asyncio.BaseTransport*), 924
- `get_python_version()` (en el módulo *sysconfig*), 1712
- `get_recsrc()` (método de *ossaudio-dev.oss_mixer_device*), 1365
- `get_referents()` (en el módulo *gc*), 1756
- `get_referrers()` (en el módulo *gc*), 1755
- `get_request()` (método de *socketserver.BaseServer*), 1297
- `get_returncode()` (método de *asyncio.SubprocessTransport*), 925
- `get_running_loop()` (en el módulo *asyncio*), 897
- `get_scheme()` (método de *wsgiref.handlers.BaseHandler*), 1216
- `get_scheme_names()` (en el módulo *sysconfig*), 1712

- [get_sequences\(\)](#) (método de *mailbox.MH*), 1112
[get_sequences\(\)](#) (método de *mailbox.MHMessage*), 1118
[get_server\(\)](#) (método de *multiprocessing.managers.BaseManager*), 803
[get_server_certificate\(\)](#) (en el módulo *ssl*), 978
[get_shapepoly\(\)](#) (en el módulo *turtle*), 1403
[get_socket\(\)](#) (método de *telnetlib.Telnet*), 1290
[get_source\(\)](#) (método de *importlib.abc.InspectLoader*), 1795
[get_source\(\)](#) (método de *importlib.abc.SourceLoader*), 1797
[get_source\(\)](#) (método de *importlib.machinery.ExtensionFileLoader*), 1802
[get_source\(\)](#) (método de *importlib.machinery.SourcelessFileLoader*), 1802
[get_source\(\)](#) (método de *zipimport.zipimporter*), 1782
[get_source_segment\(\)](#) (en el módulo *ast*), 1822
[get_stack\(\)](#) (método de *asyncio.Task*), 877
[get_stack\(\)](#) (método de *bdb.Bdb*), 1637
[get_start_method\(\)](#) (en el módulo *multiprocessing*), 795
[get_starttag_text\(\)](#) (método de *html.parser.HTMLParser*), 1137
[get_stats\(\)](#) (en el módulo *gc*), 1755
[get_stderr\(\)](#) (método de *wsgiref.handlers.BaseHandler*), 1216
[get_stderr\(\)](#) (método de *wsgiref.simple_server.WSGIRequestHandler*), 1213
[get_stdin\(\)](#) (método de *wsgiref.handlers.BaseHandler*), 1216
[get_string\(\)](#) (método de *mailbox.Mailbox*), 1108
[get_subdir\(\)](#) (método de *mailbox.MaildirMessage*), 1115
[get_suffixes\(\)](#) (en el módulo *imp*), 1928
[get_symbols\(\)](#) (método de *symtable.SymbolTable*), 1825
[get_tag\(\)](#) (en el módulo *imp*), 1930
[get_task_factory\(\)](#) (método de *asyncio.loop*), 901
[get_terminal_size\(\)](#) (en el módulo *os*), 574
[get_terminal_size\(\)](#) (en el módulo *shutil*), 420
[get_terminator\(\)](#) (método de *asyncio.chat.async_chat*), 1022
[get_threshold\(\)](#) (en el módulo *gc*), 1755
[get_token\(\)](#) (método de *shlex.shlex*), 1424
[get_traceback_limit\(\)](#) (en el módulo *tracemalloc*), 1666
[get_traced_memory\(\)](#) (en el módulo *tracemalloc*), 1666
[get_tracemalloc_memory\(\)](#) (en el módulo *tracemalloc*), 1666
[get_type\(\)](#) (método de *symtable.SymbolTable*), 1824
[get_type_hints\(\)](#) (en el módulo *typing*), 1491
[get_unixfrom\(\)](#) (método de *email.message.EmailMessage*), 1040
[get_unixfrom\(\)](#) (método de *email.message.Message*), 1077
[get_unpack_formats\(\)](#) (en el módulo *shutil*), 419
[get_usage\(\)](#) (método de *optparse.OptionParser*), 1920
[get_value\(\)](#) (método de *string.Formatter*), 100
[get_version\(\)](#) (método de *optparse.OptionParser*), 1910
[get_visible\(\)](#) (método de *mailbox.BabylMessage*), 1120
[get_wch\(\)](#) (método de *curses.window*), 714
[get_write_buffer_limits\(\)](#) (método de *asyncio.WriteTransport*), 924
[get_write_buffer_size\(\)](#) (método de *asyncio.WriteTransport*), 924
[GET_YIELD_FROM_ITER](#) (opcode), 1846
[getacl\(\)](#) (método de *imaplib.IMAP4*), 1268
[getaddresses\(\)](#) (en el módulo *email.utils*), 1093
[getaddrinfo\(\)](#) (en el módulo *socket*), 958
[getaddrinfo\(\)](#) (método de *asyncio.loop*), 909
[getallocatedblocks\(\)](#) (en el módulo *sys*), 1697
[getandroidapilevel\(\)](#) (en el módulo *sys*), 1698
[getannotation\(\)](#) (método de *imaplib.IMAP4*), 1268
[getargspec\(\)](#) (en el módulo *inspect*), 1767
[getargvalues\(\)](#) (en el módulo *inspect*), 1767
[getatime\(\)](#) (en el módulo *os.path*), 391
[getattr\(\)](#) (función incorporada), 12
[getattr_static\(\)](#) (en el módulo *inspect*), 1770
[getAttribute\(\)](#) (método de *xml.dom.Element*), 1167
[getAttributeNode\(\)](#) (método de *xml.dom.Element*), 1167
[getAttributeNodeNS\(\)](#) (método de *xml.dom.Element*), 1167
[getAttributeNS\(\)](#) (método de *xml.dom.Element*), 1167
[GetBase\(\)](#) (método de *xml.parsers.expat.xmlparser*), 1190
[getbegyx\(\)](#) (método de *curses.window*), 714
[getbkgd\(\)](#) (método de *curses.window*), 714
[getblocking\(\)](#) (método de *socket.socket*), 964
[getboolean\(\)](#) (método de *configparser.ConfigParser*), 532
[getbuffer\(\)](#) (método de *io.BytesIO*), 617
[getByteStream\(\)](#) (método de *xml.sax.xmlreader.InputSource*), 1188
[getcallargs\(\)](#) (en el módulo *inspect*), 1768
[getcanvas\(\)](#) (en el módulo *turtle*), 1410
[getcapabilities\(\)](#) (método de *nnplib.NNTP*), 1274
[getcaps\(\)](#) (en el módulo *mailcap*), 1106
[getch\(\)](#) (en el módulo *msvcrt*), 1868
[getch\(\)](#) (método de *curses.window*), 714
[getCharacterStream\(\)](#) (método de

- xml.sax.xmlreader.InputSource*), 1188
- `getche()` (en el módulo *msvcrt*), 1868
- `getcheckinterval()` (en el módulo *sys*), 1698
- `getChild()` (método de *logging.Logger*), 667
- `getchildren()` (método de *xml.etree.ElementTree.Element*), 1154
- `getclasstree()` (en el módulo *inspect*), 1767
- `getclosuresvars()` (en el módulo *inspect*), 1769
- `GetColumnInfo()` (método de *msilib.View*), 1863
- `getColumnNumber()` (método de *xml.sax.xmlreader.Locator*), 1187
- `getcomments()` (en el módulo *inspect*), 1762
- `getcompname()` (método de *aifc.aifc*), 1351
- `getcompname()` (método de *sunau.AU_read*), 1354
- `getcompname()` (método de *wave.Wave_read*), 1356
- `getcomptype()` (método de *aifc.aifc*), 1351
- `getcomptype()` (método de *sunau.AU_read*), 1354
- `getcomptype()` (método de *wave.Wave_read*), 1356
- `getContentHandler()` (método de *xml.sax.xmlreader.XMLReader*), 1186
- `getcontext()` (en el módulo *decimal*), 309
- `getcoroutinelocals()` (en el módulo *inspect*), 1772
- `getcoroutinestate()` (en el módulo *inspect*), 1771
- `getctime()` (en el módulo *os.path*), 391
- `getcwd()` (en el módulo *os*), 578
- `getcwdb()` (en el módulo *os*), 579
- `getcwdu(2to3 fixer)`, 1609
- `getdecoder()` (en el módulo *codecs*), 159
- `getdefaultencoding()` (en el módulo *sys*), 1698
- `getdefaultlocale()` (en el módulo *locale*), 1379
- `getdefaulttimeout()` (en el módulo *socket*), 961
- `getdlopenflags()` (en el módulo *sys*), 1698
- `getdoc()` (en el módulo *inspect*), 1762
- `getDOMImplementation()` (en el módulo *xml.dom*), 1162
- `getDTDHandler()` (método de *xml.sax.xmlreader.XMLReader*), 1186
- `getEffectiveLevel()` (método de *logging.Logger*), 666
- `getegid()` (en el módulo *os*), 562
- `getElementsByTagName()` (método de *xml.dom.Document*), 1166
- `getElementsByTagName()` (método de *xml.dom.Element*), 1167
- `getElementsByTagNameNS()` (método de *xml.dom.Document*), 1166
- `getElementsByTagNameNS()` (método de *xml.dom.Element*), 1167
- `getencoder()` (en el módulo *codecs*), 159
- `getEncoding()` (método de *xml.sax.xmlreader.InputSource*), 1187
- `getEntityResolver()` (método de *xml.sax.xmlreader.XMLReader*), 1186
- `getenv()` (en el módulo *os*), 561
- `getenvb()` (en el módulo *os*), 562
- `getErrorHandler()` (método de *xml.sax.xmlreader.XMLReader*), 1186
- `geteuid()` (en el módulo *os*), 562
- `getEvent()` (método de *xml.dom.pulldom.DOMEventStream*), 1177
- `getEventCategory()` (método de *logging.handlers.NTEventLogHandler*), 700
- `getEventType()` (método de *logging.handlers.NTEventLogHandler*), 700
- `getException()` (método de *xml.sax.SAXException*), 1179
- `getFeature()` (método de *xml.sax.xmlreader.XMLReader*), 1186
- `GetFieldCount()` (método de *msilib.Record*), 1864
- `getfile()` (en el módulo *inspect*), 1762
- `getfilesystemencodeerrors()` (en el módulo *sys*), 1698
- `getfilesystemencoding()` (en el módulo *sys*), 1698
- `getfirst()` (método de *cgi.FieldStorage*), 1205
- `getfloat()` (método de *configparser.ConfigParser*), 532
- `getfmts()` (método de *ossaudio-dev.oss_audio_device*), 1363
- `getfqdn()` (en el módulo *socket*), 959
- `getframeinfo()` (en el módulo *inspect*), 1769
- `getframerate()` (método de *aifc.aifc*), 1351
- `getframerate()` (método de *sunau.AU_read*), 1354
- `getframerate()` (método de *wave.Wave_read*), 1356
- `getfullargspec()` (en el módulo *inspect*), 1767
- `getgeneratorlocals()` (en el módulo *inspect*), 1771
- `getgeneratorstate()` (en el módulo *inspect*), 1771
- `getgid()` (en el módulo *os*), 562
- `getgrall()` (en el módulo *grp*), 1884
- `getgrgid()` (en el módulo *grp*), 1884
- `getgrnam()` (en el módulo *grp*), 1884
- `getgrouplist()` (en el módulo *os*), 562
- `getgroups()` (en el módulo *os*), 562
- `getheader()` (método de *http.client.HTTPResponse*), 1255
- `getheaders()` (método de *http.client.HTTPResponse*), 1255
- `gethostbyaddr()` (en el módulo *socket*), 959
- `gethostbyaddr()` (in module *socket*), 565
- `gethostbyname()` (en el módulo *socket*), 959
- `gethostbyname_ex()` (en el módulo *socket*), 959
- `gethostname()` (en el módulo *socket*), 959
- `gethostname()` (in module *socket*), 565
- `getincrementaldecoder()` (en el módulo *codecs*), 159
- `getincrementalencoder()` (en el módulo *codecs*), 159
- `getinfo()` (método de *zipfile.ZipFile*), 487
- `getinnerframes()` (en el módulo *inspect*), 1770
- `GetInputContext()` (método de *xml.parsers.expat.xmlparser*), 1190

- `getint()` (método de `configparser.ConfigParser`), 532
`GetInteger()` (método de `msilib.Record`), 1864
`getitem()` (en el módulo `operator`), 367
`getiterator()` (método de `xml.etree.ElementTree.Element`), 1155
`getiterator()` (método de `xml.etree.ElementTree.ElementTree`), 1156
`getitimer()` (en el módulo `signal`), 1028
`getkey()` (método de `curses.window`), 714
`GetLastError()` (en el módulo `ctypes`), 761
`getLength()` (método de `xml.sax.xmlreader.Attributes`), 1188
`getLevelName()` (en el módulo `logging`), 678
`getline()` (en el módulo `linecache`), 410
`getLineNumber()` (método de `xml.sax.xmlreader Locator`), 1187
`getList()` (método de `cgi.FieldStorage`), 1205
`getloadavg()` (en el módulo `os`), 608
`getlocale()` (en el módulo `locale`), 1379
`getLogger()` (en el módulo `logging`), 676
`getLoggerClass()` (en el módulo `logging`), 676
`getlogin()` (en el módulo `os`), 562
`getLogRecordFactory()` (en el módulo `logging`), 676
`getmark()` (método de `aifc.aifc`), 1351
`getmark()` (método de `sunau.AU_read`), 1354
`getmark()` (método de `wave.Wave_read`), 1356
`getmarkers()` (método de `aifc.aifc`), 1351
`getmarkers()` (método de `sunau.AU_read`), 1354
`getmarkers()` (método de `wave.Wave_read`), 1356
`getmaxyx()` (método de `curses.window`), 714
`getmember()` (método de `tarfile.TarFile`), 498
`getmembers()` (en el módulo `inspect`), 1759
`getmembers()` (método de `tarfile.TarFile`), 498
`getMessage()` (método de `logging.LogRecord`), 673
`getMessage()` (método de `xml.sax.SAXException`), 1179
`getMessageID()` (método de `logging.handlers.NTEventLogHandler`), 700
`getmodule()` (en el módulo `inspect`), 1762
`getmodulename()` (en el módulo `inspect`), 1759
`getmouse()` (en el módulo `curses`), 707
`getmro()` (en el módulo `inspect`), 1768
`getmtime()` (en el módulo `os.path`), 391
`getname()` (método de `chunk.Chunk`), 1358
`getName()` (método de `threading.Thread`), 773
`getNameByQName()` (método de `xml.sax.xmlreader.AttributesNS`), 1188
`getnameinfo()` (en el módulo `socket`), 959
`getnameinfo()` (método de `asyncio.loop`), 909
`getnames()` (método de `tarfile.TarFile`), 498
`getNames()` (método de `xml.sax.xmlreader.Attributes`), 1188
`getnchannels()` (método de `aifc.aifc`), 1351
`getnchannels()` (método de `sunau.AU_read`), 1354
`getnchannels()` (método de `wave.Wave_read`), 1356
`getnframes()` (método de `aifc.aifc`), 1351
`getnframes()` (método de `sunau.AU_read`), 1354
`getnframes()` (método de `wave.Wave_read`), 1356
`getnode`, 1292
`getnode()` (en el módulo `uuid`), 1292
`getopt` (módulo), 663
`getopt()` (en el módulo `getopt`), 663
`GetoptError`, 663
`getouterframes()` (en el módulo `inspect`), 1770
`getoutput()` (en el módulo `subprocess`), 853
`getpagesize()` (en el módulo `resource`), 1896
`getparams()` (método de `aifc.aifc`), 1351
`getparams()` (método de `sunau.AU_read`), 1354
`getparams()` (método de `wave.Wave_read`), 1356
`getparyx()` (método de `curses.window`), 714
`getpass` (módulo), 704
`getpass()` (en el módulo `getpass`), 704
`GetPassWarning`, 705
`getpeercert()` (método de `ssl.SSLSocket`), 987
`getpeername()` (método de `socket.socket`), 964
`getpen()` (en el módulo `turtle`), 1404
`getpgid()` (en el módulo `os`), 562
`getpgrp()` (en el módulo `os`), 563
`getpid()` (en el módulo `os`), 563
`getpos()` (método de `html.parser.HTMLParser`), 1137
`getppid()` (en el módulo `os`), 563
`getpreferredencoding()` (en el módulo `locale`), 1380
`getpriority()` (en el módulo `os`), 563
`getprofile()` (en el módulo `sys`), 1699
`GetProperty()` (método de `msilib.SummaryInformation`), 1863
`GetProperty()` (método de `xml.sax.xmlreader.XMLReader`), 1186
`GetPropertyCount()` (método de `msilib.SummaryInformation`), 1863
`getprotobyname()` (en el módulo `socket`), 959
`getproxies()` (en el módulo `urllib.request`), 1221
`getPublicId()` (método de `xml.sax.xmlreader.InputSource`), 1187
`getPublicId()` (método de `xml.sax.xmlreader.Locator`), 1187
`getpwall()` (en el módulo `pwd`), 1883
`getpwnam()` (en el módulo `pwd`), 1883
`getpwuid()` (en el módulo `pwd`), 1882
`getQNameByName()` (método de `xml.sax.xmlreader.AttributesNS`), 1188
`getQNames()` (método de `xml.sax.xmlreader.AttributesNS`), 1188
`getquota()` (método de `imaplib.IMAP4`), 1268
`getquotaroot()` (método de `imaplib.IMAP4`), 1268
`getrandbits()` (en el módulo `random`), 327
`getrandom()` (en el módulo `os`), 609
`getreader()` (en el módulo `codecs`), 160
`getrecursionlimit()` (en el módulo `sys`), 1699
`getrefcount()` (en el módulo `sys`), 1698
`GetReparseDeferralEnabled()` (método de `xml.parsers.expat.xmlparser`), 1191
`getresgid()` (en el módulo `os`), 563

`getresponse()` (método *http.client.HTTPConnection*), 1253
`getresuid()` (en el módulo *os*), 563
`getrlimit()` (en el módulo *resource*), 1893
`getroot()` (método *xml.etree.ElementTree.ElementTree*), 1156
`getrusage()` (en el módulo *resource*), 1895
`getsamplwidth()` (en el módulo *audioop*), 1348
`getsampwidth()` (método de *aifc.aifc*), 1351
`getsampwidth()` (método de *sunau.AU_read*), 1354
`getsampwidth()` (método de *wave.Wave_read*), 1356
`getscreen()` (en el módulo *turtle*), 1405
`getservbyname()` (en el módulo *socket*), 960
`getservbyport()` (en el módulo *socket*), 960
`GetSetDescriptorType` (en el módulo *types*), 256
`getshapes()` (en el módulo *turtle*), 1411
`getsid()` (en el módulo *os*), 565
`getsignal()` (en el módulo *signal*), 1027
`getsitpackages()` (en el módulo *site*), 1775
`getsize()` (en el módulo *os.path*), 391
`getsize()` (método de *chunk.Chunk*), 1358
`getsizeof()` (en el módulo *sys*), 1699
`getsockname()` (método de *socket.socket*), 964
`getsockopt()` (método de *socket.socket*), 964
`getsource()` (en el módulo *inspect*), 1762
`getsourcefile()` (en el módulo *inspect*), 1762
`getsourcelines()` (en el módulo *inspect*), 1762
`getspall()` (en el módulo *spwd*), 1883
`getspnam()` (en el módulo *spwd*), 1883
`getstate()` (en el módulo *random*), 327
`getstate()` (método de *codecs.IncrementalDecoder*), 165
`getstate()` (método de *codecs.IncrementalEncoder*), 164
`getstatusoutput()` (en el módulo *subprocess*), 853
`getstr()` (método de *curses.window*), 714
`GetString()` (método de *msilib.Record*), 1864
`getSubject()` (método de *logging.handlers.SMTPHandler*), 701
`GetSummaryInformation()` (método de *msilib.Database*), 1862
`getswitchinterval()` (en el módulo *sys*), 1699
`getSystemId()` (método de *xml.sax.xmlreader.InputSource*), 1187
`getSystemId()` (método de *xml.sax.xmlreader.Locator*), 1187
`getsyx()` (en el módulo *curses*), 707
`gettarinfo()` (método de *tarfile.TarFile*), 500
`gettempdir()` (en el módulo *tempfile*), 405
`gettempdirb()` (en el módulo *tempfile*), 406
`gettempprefix()` (en el módulo *tempfile*), 406
`gettempprefixb()` (en el módulo *tempfile*), 406
`getTestCaseNames()` (método de *unittest.TestLoader*), 1540
`gettext` (módulo), 1367
`gettext()` (en el módulo *gettext*), 1368
`gettext()` (en el módulo *locale*), 1383
`gettext()` (método de *gettext.GNUTranslations*), 1372
`gettext()` (método de *gettext.NullTranslations*), 1370
`gettimeout()` (método de *socket.socket*), 964
`gettrace()` (en el módulo *sys*), 1699
`getturtle()` (en el módulo *turtle*), 1404
`getType()` (método de *xml.sax.xmlreader.Attributes*), 1188
`getuid()` (en el módulo *os*), 563
`geturl()` (método de *urllib.parse.urllib.parse.SplitResult*), 1243
`getuser()` (en el módulo *getpass*), 705
`getuserbase()` (en el módulo *site*), 1775
`getusersitepackages()` (en el módulo *site*), 1775
`getvalue()` (método de *io.BytesIO*), 617
`getvalue()` (método de *io.StringIO*), 621
`getValue()` (método de *xml.sax.xmlreader.Attributes*), 1188
`getValueByQName()` (método de *xml.sax.xmlreader.AttributesNS*), 1188
`getwch()` (en el módulo *msvcrt*), 1868
`getwche()` (en el módulo *msvcrt*), 1868
`getweakrefcount()` (en el módulo *weakref*), 248
`getweakrefs()` (en el módulo *weakref*), 248
`getwelcome()` (método de *ftplib.FTP*), 1259
`getwelcome()` (método de *nnplib.NNTP*), 1274
`getwelcome()` (método de *poplib.POP3*), 1263
`getwin()` (en el módulo *curses*), 707
`getwindowsversion()` (en el módulo *sys*), 1699
`getwriter()` (en el módulo *codecs*), 160
`getxattr()` (en el módulo *os*), 595
`getyx()` (método de *curses.window*), 714
`gid` (atributo de *tarfile.TarInfo*), 502
`GIL`, 1940
`glob`
 módulo, 409
`glob` (módulo), 407
`glob()` (en el módulo *glob*), 408
`glob()` (método de *msilib.Directory*), 1865
`glob()` (método de *pathlib.Path*), 384
`globals()` (función incorporada), 13
`globs` (atributo de *doctest.DocTest*), 1512
`gmtime()` (en el módulo *time*), 625
`gname` (atributo de *tarfile.TarInfo*), 502
`GNOME`, 1373
`GNU_FORMAT` (en el módulo *tarfile*), 497
`gnu_getopt()` (en el módulo *getopt*), 663
`GNUTranslations` (clase en *gettext*), 1372
`got` (atributo de *doctest.DocTestFailure*), 1518
`goto()` (en el módulo *turtle*), 1391
Graphical User Interface, 1429
`GREATER` (en el módulo *token*), 1827
`GREATEREQUAL` (en el módulo *token*), 1828
Greenwich Mean Time, 622
`GRND_NONBLOCK` (en el módulo *os*), 610
`GRND_RANDOM` (en el módulo *os*), 610
`Group` (clase en *email.headerregistry*), 1066
`group()` (método de *nnplib.NNTP*), 1275
`group()` (método de *pathlib.Path*), 384

- `group()` (método de *re.Match*), 122
`groupby()` (en el módulo *itertools*), 349
`groupdict()` (método de *re.Match*), 123
`groupindex` (atributo de *re.Pattern*), 121
`groups` (atributo de *email.headerregistry.AddressHeader*), 1063
`groups` (atributo de *re.Pattern*), 121
`groups()` (método de *re.Match*), 123
`grp` (módulo), 1884
`gt()` (en el módulo *operator*), 365
`guess_all_extensions()` (en el módulo *mimetypes*), 1124
`guess_all_extensions()` (método de *mimetypes.MimeTypes*), 1126
`guess_extension()` (en el módulo *mimetypes*), 1124
`guess_extension()` (método de *mimetypes.MimeTypes*), 1126
`guess_scheme()` (en el módulo *wsgiref.util*), 1210
`guess_type()` (en el módulo *mimetypes*), 1124
`guess_type()` (método de *mimetypes.MimeTypes*), 1126
GUI, 1429
`gzip` (módulo), 473
`gzip` command line option
 `--best`, 475
 `-d`, 475
 `--decompress`, 475
 `--fast`, 475
 `file`, 475
 `-h`, 475
 `--help`, 475
`GzipFile` (clase en *gzip*), 473
- ## H
- `-h`
 `gzip` command line option, 475
 `json.tool` command line option, 1105
 `timeit` command line option, 1657
 `tokenize` command line option, 1832
 `zipapp` command line option, 1685
`halfdelay()` (en el módulo *curses*), 708
`Handle` (clase en *asyncio*), 913
`handle()` (método de *http.server.BaseHTTPRequestHandler*), 1304
`handle()` (método de *logging.Handler*), 670
`handle()` (método de *logging.handlers.QueueListener*), 704
`handle()` (método de *logging.Logger*), 669
`handle()` (método de *logging.NullHandler*), 693
`handle()` (método de *socketserver.BaseRequestHandler*), 1298
`handle()` (método de *wsgiref.simple_server.WSGIRequestHandler*), 1213
`handle_accept()` (método de *asyncore.dispatcher*), 1018
`handle_accepted()` (método de *asyncore.dispatcher*), 1019
`handle_charref()` (método de *html.parser.HTMLParser*), 1137
`handle_close()` (método de *asyncore.dispatcher*), 1018
`handle_comment()` (método de *html.parser.HTMLParser*), 1138
`handle_connect()` (método de *asyncore.dispatcher*), 1018
`handle_data()` (método de *html.parser.HTMLParser*), 1137
`handle_decl()` (método de *html.parser.HTMLParser*), 1138
`handle_defect()` (método de *email.policy.Policy*), 1055
`handle_endtag()` (método de *html.parser.HTMLParser*), 1137
`handle_entityref()` (método de *html.parser.HTMLParser*), 1137
`handle_error()` (método de *asyncore.dispatcher*), 1018
`handle_error()` (método de *socketserver.BaseServer*), 1297
`handle_expect_100()` (método de *http.server.BaseHTTPRequestHandler*), 1304
`handle_expt()` (método de *asyncore.dispatcher*), 1018
`handle_one_request()` (método de *http.server.BaseHTTPRequestHandler*), 1304
`handle_pi()` (método de *html.parser.HTMLParser*), 1138
`handle_read()` (método de *asyncore.dispatcher*), 1018
`handle_request()` (método de *socketserver.BaseServer*), 1296
`handle_request()` (método de *xmlrpc.server.CGIXMLRPCRequestHandler*), 1332
`handle_startendtag()` (método de *html.parser.HTMLParser*), 1137
`handle_starttag()` (método de *html.parser.HTMLParser*), 1137
`handle_timeout()` (método de *socketserver.BaseServer*), 1298
`handle_write()` (método de *asyncore.dispatcher*), 1018
`handleError()` (método de *logging.Handler*), 670
`handleError()` (método de *logging.handlers.SocketHandler*), 697
`Handler` (clase en *logging*), 669
`handler()` (en el módulo *cgiib*), 1209
`harmonic_mean()` (en el módulo *statistics*), 334
`HAS_ALPN` (en el módulo *ssl*), 983
`has_children()` (método de *syntable.SymbolTable*), 1824

- `has_colors()` (en el módulo `curses`), 707
- `has_dualstack_ipv6()` (en el módulo `socket`), 957
- `HAS_ECDH` (en el módulo `ssl`), 983
- `has_exec()` (método de `syntable.SymbolTable`), 1824
- `has_extn()` (método de `smtplib.SMTP`), 1281
- `has_header()` (método de `csv.Sniffer`), 514
- `has_header()` (método de `urllib.request.Request`), 1225
- `has_ic()` (en el módulo `curses`), 707
- `has_il()` (en el módulo `curses`), 707
- `has_ipv6` (en el módulo `socket`), 956
- `has_key` (2to3 fixer), 1609
- `has_key()` (en el módulo `curses`), 708
- `has_location` (atributo de `importlib.machinery.ModuleSpec`), 1803
- `HAS_NEVER_CHECK_COMMON_NAME` (en el módulo `ssl`), 983
- `has_nonstandard_attr()` (método de `http.cookiejar.Cookie`), 1318
- `HAS_NPN` (en el módulo `ssl`), 983
- `has_option()` (método de `configparser.ConfigParser`), 531
- `has_option()` (método de `optparse.OptionParser`), 1919
- `has_section()` (método de `configparser.ConfigParser`), 531
- `HAS_SNI` (en el módulo `ssl`), 983
- `HAS_SSLv2` (en el módulo `ssl`), 983
- `HAS_SSLv3` (en el módulo `ssl`), 983
- `has_ticket` (atributo de `ssl.SSLSession`), 1004
- `HAS_TLSv1` (en el módulo `ssl`), 983
- `HAS_TLSv1_1` (en el módulo `ssl`), 984
- `HAS_TLSv1_2` (en el módulo `ssl`), 984
- `HAS_TLSv1_3` (en el módulo `ssl`), 984
- `hasattr()` (función incorporada), 13
- `hasAttribute()` (método de `xml.dom.Element`), 1167
- `hasAttributeNS()` (método de `xml.dom.Element`), 1167
- `hasAttributes()` (método de `xml.dom.Node`), 1164
- `hasChildNodes()` (método de `xml.dom.Node`), 1164
- `hascompare` (en el módulo `dis`), 1855
- `hasconst` (en el módulo `dis`), 1854
- `hasFeature()` (método de `xml.dom.DOMImplementation`), 1163
- `hasfree` (en el módulo `dis`), 1854
- `hash`
 - función incorporada, 39
- `hash()` (función incorporada), 13
- `hash-based pyc`, 1940
- `hash_info` (en el módulo `sys`), 1700
- `hashable`, 1940
- `Hashable` (clase en `collections.abc`), 234
- `Hashable` (clase en `typing`), 1486
- `hasHandlers()` (método de `logging.Logger`), 669
- `hash.block_size` (en el módulo `hashlib`), 545
- `hash.digest_size` (en el módulo `hashlib`), 545
- `hashlib` (módulo), 543
- `hasjabs` (en el módulo `dis`), 1854
- `hasjrel` (en el módulo `dis`), 1854
- `haslocal` (en el módulo `dis`), 1855
- `hasname` (en el módulo `dis`), 1854
- `HAVE_ARGUMENT` (opcode), 1854
- `HAVE_CONTEXTVAR` (en el módulo `decimal`), 314
- `HAVE_DOCSTRINGS` (en el módulo `test.support`), 1616
- `HAVE_THREADS` (en el módulo `decimal`), 314
- `HCI_DATA_DIR` (en el módulo `socket`), 956
- `HCI_FILTER` (en el módulo `socket`), 956
- `HCI_TIME_STAMP` (en el módulo `socket`), 956
- `head()` (método de `nnplib.NNTP`), 1277
- `Header` (clase en `email.header`), 1087
- `header_encode()` (método de `email.charset.Charset`), 1090
- `header_encode_lines()` (método de `email.charset.Charset`), 1090
- `header_encoding` (atributo de `email.charset.Charset`), 1089
- `header_factory` (atributo de `email.policy.EmailPolicy`), 1057
- `header_fetch_parse()` (método de `email.policy.Compat32`), 1059
- `header_fetch_parse()` (método de `email.policy.EmailPolicy`), 1058
- `header_fetch_parse()` (método de `email.policy.Policy`), 1056
- `header_items()` (método de `urllib.request.Request`), 1226
- `header_max_count()` (método de `email.policy.EmailPolicy`), 1058
- `header_max_count()` (método de `email.policy.Policy`), 1056
- `header_offset` (atributo de `zipfile.ZipInfo`), 493
- `header_source_parse()` (método de `email.policy.Compat32`), 1059
- `header_source_parse()` (método de `email.policy.EmailPolicy`), 1058
- `header_source_parse()` (método de `email.policy.Policy`), 1056
- `header_store_parse()` (método de `email.policy.Compat32`), 1059
- `header_store_parse()` (método de `email.policy.EmailPolicy`), 1058
- `header_store_parse()` (método de `email.policy.Policy`), 1056
- `HeaderError`, 496
- `HeaderParseError`, 1060
- `HeaderParser` (clase en `email.parser`), 1049
- `HeaderRegistry` (clase en `email.headerregistry`), 1065
- `headers`
 - MIME, 1124, 1202
- `headers` (atributo de `http.server.BaseHTTPRequestHandler`), 1303
- `headers` (atributo de `urllib.error.HTTPError`), 1247
- `headers` (atributo de `xmlrpc.client.ProtocolError`),

- 1325
- Headers (*clase en `wsgiref.headers`*), 1211
- HeaderWriteError, 1060
- heading() (*en el módulo `turtle`*), 1395
- heading() (*método de `tkinter.ttk.Treeview`*), 1453
- heapify() (*en el módulo `heapq`*), 238
- heapmin() (*en el módulo `msvcrt`*), 1868
- heappop() (*en el módulo `heapq`*), 237
- heappush() (*en el módulo `heapq`*), 237
- heappushpop() (*en el módulo `heapq`*), 238
- heapq (*módulo*), 237
- heapreplace() (*en el módulo `heapq`*), 238
- helo() (*método de `smtpplib.SMTP`*), 1281
- help
- online, 1496
- help
- gzip command line option, 475
 - json.tool command line option, 1105
 - timeit command line option, 1657
 - tokenize command line option, 1832
 - trace command line option, 1659
 - zipapp command line option, 1685
- help (*atributo de `optparse.Option`*), 1914
- help (*`pdb` command*), 1642
- help() (*función incorporada*), 13
- help() (*método de `nnplib.NNTP`*), 1276
- herror, 953
- hex (*atributo de `uuid.UUID`*), 1292
- hex() (*función incorporada*), 13
- hex() (*método de `bytearray`*), 56
- hex() (*método de `bytes`*), 55
- hex() (*método de `float`*), 34
- hex() (*método de `memoryview`*), 71
- hexadecimal
- literals, 31
- hexbin() (*en el módulo `binhex`*), 1130
- hexdigest() (*método de `hashlib.hash`*), 545
- hexdigest() (*método de `hashlib.shake`*), 545
- hexdigest() (*método de `hmac.HMAC`*), 554
- hexdigits (*en el módulo `string`*), 99
- hexlify() (*en el módulo `binascii`*), 1132
- hexversion (*en el módulo `sys`*), 1701
- hidden() (*método de `curses.panel.Panel`*), 726
- hide() (*método de `curses.panel.Panel`*), 726
- hide() (*método de `tkinter.ttk.Notebook`*), 1447
- hide_cookie2 (*atributo de `http.cookiejar.CookiePolicy`*), 1316
- hideturtle() (*en el módulo `turtle`*), 1400
- HierarchyRequestErr, 1169
- HIGH_PRIORITY_CLASS (*en el módulo `subprocess`*), 847
- HIGHEST_PROTOCOL (*en el módulo `pickle`*), 423
- HKEY_CLASSES_ROOT (*en el módulo `winreg`*), 1874
- HKEY_CURRENT_CONFIG (*en el módulo `winreg`*), 1875
- HKEY_CURRENT_USER (*en el módulo `winreg`*), 1874
- HKEY_DYN_DATA (*en el módulo `winreg`*), 1875
- HKEY_LOCAL_MACHINE (*en el módulo `winreg`*), 1874
- HKEY_PERFORMANCE_DATA (*en el módulo `winreg`*), 1874
- HKEY_USERS (*en el módulo `winreg`*), 1874
- hline() (*método de `curses.window`*), 715
- HList (*clase en `tkinter.tix`*), 1461
- hls_to_rgb() (*en el módulo `coloursys`*), 1359
- hmac (*módulo*), 553
- HOME, 391
- home() (*en el módulo `turtle`*), 1392
- home() (*método de clase de `pathlib.Path`*), 383
- HOMEDRIVE, 391
- HOMEPATH, 391
- hook_compressed() (*en el módulo `fileinput`*), 396
- hook_encoded() (*en el módulo `fileinput`*), 396
- host (*atributo de `urllib.request.Request`*), 1225
- hostmask (*atributo de `ipaddress.IPv4Network`*), 1339
- hostmask (*atributo de `ipaddress.IPv6Network`*), 1342
- hostname_checks_common_name (*atributo de `ssl.SSLContext`*), 996
- hosts (*atributo de `netrc.netrc`*), 536
- hosts() (*método de `ipaddress.IPv4Network`*), 1340
- hosts() (*método de `ipaddress.IPv6Network`*), 1342
- hour (*atributo de `datetime.datetime`*), 190
- hour (*atributo de `datetime.time`*), 198
- HRESULT (*clase en `ctypes`*), 765
- hStdError (*atributo de `subprocess.STARTUPINFO`*), 846
- hStdInput (*atributo de `subprocess.STARTUPINFO`*), 846
- hStdOutput (*atributo de `subprocess.STARTUPINFO`*), 846
- hsv_to_rgb() (*en el módulo `coloursys`*), 1359
- ht() (*en el módulo `turtle`*), 1400
- HTML, 1136, 1237
- html (*módulo*), 1135
- html() (*en el módulo `cgitb`*), 1209
- html5 (*en el módulo `html.entities`*), 1140
- HTMLCalendar (*clase en `calendar`*), 214
- HtmlDiff (*clase en `difflib`*), 130
- html.entities (*módulo*), 1140
- HTMLParser (*clase en `html.parser`*), 1136
- html.parser (*módulo*), 1136
- htonl() (*en el módulo `socket`*), 960
- htons() (*en el módulo `socket`*), 960
- HTTP
- http (*standard module*), 1248
 - http.client (*standard module*), 1250
 - protocol, 1202, 1237, 1248, 1250, 1302
- HTTP (*en el módulo `email.policy`*), 1059
- http (*módulo*), 1248
- http_error_301() (*método de `urllib.request.HTTPRedirectHandler`*), 1229
- http_error_302() (*método de `urllib.request.HTTPRedirectHandler`*), 1229
- http_error_303() (*método de `urllib.request.HTTPRedirectHandler`*), 1229
- http_error_307() (*método de `urllib.request.HTTPRedirectHandler`*), 1229

`http_error_401()` (método de `urllib.request.HTTPBasicAuthHandler`), 1230
`http_error_401()` (método de `urllib.request.HTTPDigestAuthHandler`), 1231
`http_error_407()` (método de `urllib.request.ProxyBasicAuthHandler`), 1230
`http_error_407()` (método de `urllib.request.ProxyDigestAuthHandler`), 1231
`http_error_auth_reqed()` (método de `urllib.request.AbstractBasicAuthHandler`), 1230
`http_error_auth_reqed()` (método de `urllib.request.AbstractDigestAuthHandler`), 1230
`http_error_default()` (método de `urllib.request.BaseHandler`), 1228
`http_open()` (método de `urllib.request.HTTPHandler`), 1231
`HTTP_PORT` (en el módulo `http.client`), 1252
`http_proxy`, 1220, 1233
`http_response()` (método de `urllib.request.HTTPErrorProcessor`), 1232
`http_version` (atributo de `wsgiref.handlers.BaseHandler`), 1218
`HTTPBasicAuthHandler` (clase en `urllib.request`), 1223
`http.client` (módulo), 1250
`HTTPConnection` (clase en `http.client`), 1250
`http.cookiejar` (módulo), 1311
`HTTPCookieProcessor` (clase en `urllib.request`), 1223
`http.cookies` (módulo), 1308
`httplib`, 1302
`HTTPDefaultErrorHandler` (clase en `urllib.request`), 1223
`HTTPDigestAuthHandler` (clase en `urllib.request`), 1224
`HTTPError`, 1246
`HTTPErrorProcessor` (clase en `urllib.request`), 1224
`HTTPException`, 1252
`HTTPHandler` (clase en `logging.handlers`), 702
`HTTPHandler` (clase en `urllib.request`), 1224
`HTTPPasswordMgr` (clase en `urllib.request`), 1223
`HTTPPasswordMgrWithDefaultRealm` (clase en `urllib.request`), 1223
`HTTPPasswordMgrWithPriorAuth` (clase en `urllib.request`), 1223
`HTTPRedirectHandler` (clase en `urllib.request`), 1223
`HTTPResponse` (clase en `http.client`), 1251
`https_open()` (método de `urllib.request.HTTPSHandler`), 1231
`HTTPS_PORT` (en el módulo `http.client`), 1252
`https_response()` (método de `urllib.request.HTTPErrorProcessor`), 1232
`HTTPSConnection` (clase en `http.client`), 1251
`http.server`
 security, 1308
`HTTPServer` (clase en `http.server`), 1302
`http.server` (módulo), 1302
`HTTPSHandler` (clase en `urllib.request`), 1224
`HTTPStatus` (clase en `http`), 1248
`hypot()` (en el módulo `math`), 292

I
(en el módulo `re`), 116
-i list
 compileall command line option, 1838
I/O control
 buffering, 19, 964
 POSIX, 1886
 tty, 1886
 UNIX, 1889
`iadd()` (en el módulo `operator`), 370
`iand()` (en el módulo `operator`), 370
`iconcat()` (en el módulo `operator`), 370
`id` (atributo de `ssl.SSLSession`), 1004
`id()` (función incorporada), 13
`id()` (método de `unittest.TestCase`), 1535
`idcok()` (método de `curses.window`), 715
`ident` (atributo de `select.kevent`), 1012
`ident` (atributo de `threading.Thread`), 773
`identchars` (atributo de `cmd.Cmd`), 1420
`identify()` (método de `tkinter.ttk.Notebook`), 1447
`identify()` (método de `tkinter.ttk.Treeview`), 1453
`identify()` (método de `tkinter.ttk.Widget`), 1443
`identify_column()` (método de `tkinter.ttk.Treeview`), 1453
`identify_element()` (método de `tkinter.ttk.Treeview`), 1453
`identify_region()` (método de `tkinter.ttk.Treeview`), 1453
`identify_row()` (método de `tkinter.ttk.Treeview`), 1453
idioms (2to3 fixer), 1609
IDLE, 1464, 1940
`IDLE_PRIORITY_CLASS` (en el módulo `subprocess`), 847
`IDLESTARTUP`, 1471
`idlok()` (método de `curses.window`), 715
`if`
 sentencia, 29
`if_indextoname()` (en el módulo `socket`), 962
`if_nameindex()` (en el módulo `socket`), 961
`if_nametoindex()` (en el módulo `socket`), 962
`ifloordiv()` (en el módulo `operator`), 371
`iglob()` (en el módulo `glob`), 408
`ignorableWhitespace()` (método de `xml.sax.handler.ContentHandler`), 1182
`ignore` (`pdb` command), 1643
`ignore_errors()` (en el módulo `codecs`), 163
`IGNORE_EXCEPTION_DETAIL` (en el módulo `doctest`), 1504
`ignore_patterns()` (en el módulo `shutil`), 413
`IGNORECASE` (en el módulo `re`), 116

- ignore-dir=<dir>
 - trace command line option, 1660
- ignore-module=<mod>
 - trace command line option, 1660
- ihave() (método de *nnplib.NNTP*), 1277
- IISCGIHandler (clase en *wsgiref.handlers*), 1215
- ilshift() (en el módulo *operator*), 371
- imag (atributo de *numbers.Complex*), 285
- imap() (método de *multiprocessing.pool.Pool*), 810
- IMAP4
 - protocol, 1265
- IMAP4 (clase en *imaplib*), 1265
- IMAP4_SSL
 - protocol, 1265
- IMAP4_SSL (clase en *imaplib*), 1265
- IMAP4_stream
 - protocol, 1265
- IMAP4_stream (clase en *imaplib*), 1266
- IMAP4.abort, 1265
- IMAP4.error, 1265
- IMAP4.readonly, 1265
- imap_unordered() (método de *multiprocessing.pool.Pool*), 810
- imaplib (módulo), 1265
- imatmul() (en el módulo *operator*), 371
- imgchr (módulo), 1360
- immedok() (método de *curses.window*), 715
- immutable
 - sequence types, 39
- imod() (en el módulo *operator*), 371
- imp
 - módulo, 25
- imp (módulo), 1928
- ImpImporter (clase en *pkgutil*), 1783
- impl_detail() (en el módulo *test.support*), 1621
- implementation (en el módulo *sys*), 1701
- ImpLoader (clase en *pkgutil*), 1784
- import
 - sentencia, 25, 1773, 1928
- import (2to3 fixer), 1609
- import_fresh_module() (en el módulo *test.support*), 1622
- IMPORT_FROM (opcode), 1851
- import_module() (en el módulo *importlib*), 1790
- import_module() (en el módulo *test.support*), 1622
- IMPORT_NAME (opcode), 1851
- IMPORT_STAR (opcode), 1848
- importador, 1941
- importar, 1941
- ImportError, 91
- importlib (módulo), 1789
- importlib.abc (módulo), 1792
- importlib.machinery (módulo), 1799
- importlib.metadata (módulo), 1808
- importlib.resources (módulo), 1798
- importlib.util (módulo), 1803
- imports (2to3 fixer), 1609
- imports2 (2to3 fixer), 1609
- ImportWarning, 96
- ImproperConnectionState, 1252
- imul() (en el módulo *operator*), 371
- in
 - operador, 30, 38
- in_dll() (método de *ctypes._CDATA*), 763
- in_table_a1() (en el módulo *stringprep*), 146
- in_table_b1() (en el módulo *stringprep*), 146
- in_table_c3() (en el módulo *stringprep*), 146
- in_table_c4() (en el módulo *stringprep*), 146
- in_table_c5() (en el módulo *stringprep*), 146
- in_table_c6() (en el módulo *stringprep*), 146
- in_table_c7() (en el módulo *stringprep*), 146
- in_table_c8() (en el módulo *stringprep*), 146
- in_table_c9() (en el módulo *stringprep*), 146
- in_table_c11() (en el módulo *stringprep*), 146
- in_table_c11_c12() (en el módulo *stringprep*), 146
- in_table_c12() (en el módulo *stringprep*), 146
- in_table_c21() (en el módulo *stringprep*), 146
- in_table_c21_c22() (en el módulo *stringprep*), 146
- in_table_c22() (en el módulo *stringprep*), 146
- in_table_d1() (en el módulo *stringprep*), 147
- in_table_d2() (en el módulo *stringprep*), 147
- in_transaction (atributo de *sqlite3.Connection*), 450
- inch() (método de *curses.window*), 715
- inclusive (atributo de *tracemalloc.DomainFilter*), 1667
- inclusive (atributo de *tracemalloc.Filter*), 1667
- Incomplete, 1132
- IncompleteRead, 1252
- IncompleteReadError, 896
- increment_lineno() (en el módulo *ast*), 1822
- incrementaldecoder (atributo de *codecs.CodecInfo*), 159
- IncrementalDecoder (clase en *codecs*), 164
- incrementalencoder (atributo de *codecs.CodecInfo*), 159
- IncrementalEncoder (clase en *codecs*), 164
- IncrementalNewlineDecoder (clase en *io*), 621
- IncrementalParser (clase en *xml.sax.xmlreader*), 1185
- indent (atributo de *doctest.Example*), 1512
- INDENT (en el módulo *token*), 1827
- indent() (en el módulo *textwrap*), 141
- IndentationError, 93
- indentlevel=<num>
 - pickletools command line option, 1855
- index() (en el módulo *operator*), 366
- index() (método de *array.array*), 245
- index() (método de *bytearray*), 58
- index() (método de *bytes*), 58
- index() (método de *collections.deque*), 223
- index() (método de *multiprocessing.shared_memory.ShareableList*), 828

- `index()` (método de `str`), 47
- `index()` (método de `tkinter.ttk.Notebook`), 1447
- `index()` (método de `tkinter.ttk.Treeview`), 1453
- `index()` (sequence method), 38
- `IndexError`, 91
- `indexOf()` (en el módulo `operator`), 367
- `IndexSizeErr`, 1169
- indicador de tipo, 1947
- `inet_aton()` (en el módulo `socket`), 960
- `inet_ntoa()` (en el módulo `socket`), 960
- `inet_ntop()` (en el módulo `socket`), 961
- `inet_pton()` (en el módulo `socket`), 960
- `Inexact` (clase en `decimal`), 315
- `inf` (en el módulo `cmath`), 297
- `inf` (en el módulo `math`), 294
- `infile`
 - `json.tool` command line option, 1105
- `infile` (atributo de `shlex.shlex`), 1426
- `Infinity`, 11
- `infj` (en el módulo `cmath`), 297
- `--info`
 - `zipapp` command line option, 1684
- `info()` (en el módulo `logging`), 677
- `info()` (método de `dis.Bytecode`), 1842
- `info()` (método de `gettext.NullTranslations`), 1371
- `info()` (método de `logging.Logger`), 668
- `infolist()` (método de `zipfile.ZipFile`), 487
- `.ini`
 - file, 518
- `ini` file, 518
- `init()` (en el módulo `mimetypes`), 1124
- `init_color()` (en el módulo `curses`), 708
- `init_database()` (en el módulo `msilib`), 1862
- `init_pair()` (en el módulo `curses`), 708
- `inited` (en el módulo `mimetypes`), 1125
- `initgroups()` (en el módulo `os`), 563
- `initial_indent` (atributo de `textwrap.TextWrapper`), 142
- `initscr()` (en el módulo `curses`), 708
- `immutable`, 1940
- `inode()` (método de `os.DirEntry`), 585
- `INPLACE_ADD` (opcode), 1847
- `INPLACE_AND` (opcode), 1847
- `INPLACE_FLOOR_DIVIDE` (opcode), 1847
- `INPLACE_LSHIFT` (opcode), 1847
- `INPLACE_MATRIX_MULTIPLY` (opcode), 1846
- `INPLACE_MODULO` (opcode), 1847
- `INPLACE_MULTIPLY` (opcode), 1846
- `INPLACE_OR` (opcode), 1847
- `INPLACE_POWER` (opcode), 1846
- `INPLACE_RSHIFT` (opcode), 1847
- `INPLACE_SUBTRACT` (opcode), 1847
- `INPLACE_TRUE_DIVIDE` (opcode), 1847
- `INPLACE_XOR` (opcode), 1847
- `input` (2to3 fixer), 1609
- `input()` (en el módulo `fileinput`), 395
- `input()` (función incorporada), 14
- `input_charset` (atributo de `email.charset.Charset`), 1089
- `input_codec` (atributo de `email.charset.Charset`), 1090
- `InputOnly` (clase en `tkinter.tix`), 1462
- `InputSource` (clase en `xml.sax.xmlreader`), 1185
- `insch()` (método de `curses.window`), 715
- `insdelln()` (método de `curses.window`), 715
- `insert()` (método de `array.array`), 245
- `insert()` (método de `collections.deque`), 223
- `insert()` (método de `tkinter.ttk.Notebook`), 1447
- `insert()` (método de `tkinter.ttk.Treeview`), 1453
- `insert()` (método de `xml.etree.ElementTree.Element`), 1155
- `insert()` (sequence method), 40
- `insert_text()` (en el módulo `readline`), 148
- `insertBefore()` (método de `xml.dom.Node`), 1164
- `insertln()` (método de `curses.window`), 715
- `insnstr()` (método de `curses.window`), 715
- `insort()` (en el módulo `bisect`), 242
- `insort_left()` (en el módulo `bisect`), 242
- `insort_right()` (en el módulo `bisect`), 242
- `inspect` (módulo), 1758
- `inspect` command line option
 - `--details`, 1773
- `InspectLoader` (clase en `importlib.abc`), 1795
- `insstr()` (método de `curses.window`), 715
- `install()` (en el módulo `gettext`), 1370
- `install()` (método de `gettext.NullTranslations`), 1371
- `install_opener()` (en el módulo `urllib.request`), 1221
- `install_scripts()` (método de `venv.EnvBuilder`), 1679
- `installHandler()` (en el módulo `unittest`), 1547
- `instate()` (método de `tkinter.ttk.Widget`), 1443
- `instr()` (método de `curses.window`), 715
- `instream` (atributo de `shlex.shlex`), 1426
- `Instruction` (clase en `dis`), 1844
- `Instruction.arg` (en el módulo `dis`), 1845
- `Instruction.argrepr` (en el módulo `dis`), 1845
- `Instruction.argval` (en el módulo `dis`), 1845
- `Instruction.is_jump_target` (en el módulo `dis`), 1845
- `Instruction.offset` (en el módulo `dis`), 1845
- `Instruction.opcode` (en el módulo `dis`), 1844
- `Instruction.opname` (en el módulo `dis`), 1844
- `Instruction.starts_line` (en el módulo `dis`), 1845
- `int`
 - función incorporada, 31
- `int` (atributo de `uuid.UUID`), 1292
- `int` (clase incorporada), 14
- `Int2AP()` (en el módulo `imaplib`), 1266
- `int_info` (en el módulo `sys`), 1701
- `integer`
 - literals, 31
 - objeto, 31
 - types, operations on, 32

- Integral (*clase en numbers*), 286
- Integrated Development Environment, 1464
- IntegrityError, 460
- Intel/DVI ADPCM, 1347
- IntEnum (*clase en enum*), 267
- interact (*pdb command*), 1645
- interact() (*en el módulo code*), 1777
- interact() (*método de code.InteractiveConsole*), 1779
- interact() (*método de telnetlib.Telnet*), 1290
- InteractiveConsole (*clase en code*), 1777
- InteractiveInterpreter (*clase en code*), 1777
- interactivo, 1941
- intern (2to3 fixer), 1609
- intern() (*en el módulo sys*), 1702
- internal_attr (*atributo de zipfile.ZipInfo*), 493
- InternalDate2tuple() (*en el módulo imaplib*), 1266
- internalSubset (*atributo de xml.dom.DocumentType*), 1165
- Internet, 1199
- interpolation
- bytearray(%), 66
 - bytes(%), 66
- interpolation, string(%), 52
- InterpolationDepthError, 535
- InterpolationError, 535
- InterpolationMissingOptionError, 535
- InterpolationSyntaxError, 535
- interpretado, 1941
- interpreter prompts, 1704
- interpreter_requires_environment() (*en el módulo test.support.script_helper*), 1627
- interrupt() (*método de sqlite3.Connection*), 452
- interrupt_main() (*en el módulo _thread*), 862
- InterruptedError, 95
- intersection() (*método de frozenset*), 76
- intersection_update() (*método de frozenset*), 77
- IntFlag (*clase en enum*), 267
- intro (*atributo de cmd.Cmd*), 1420
- InuseAttributeErr, 1169
- inv() (*en el módulo operator*), 366
- inv_cdf() (*método de statistics.NormalDist*), 340
- InvalidAccessErr, 1169
- invalidate_caches() (*en el módulo importlib*), 1790
- invalidate_caches() (*método de clase de importlib.machinery.PathFinder*), 1800
- invalidate_caches() (*método de importlib.abc.MetaPathFinder*), 1792
- invalidate_caches() (*método de importlib.abc.PathEntryFinder*), 1793
- invalidate_caches() (*método de importlib.machinery.FileFinder*), 1801
- invalidation-mode [timestamp|checked-hash|unchecked-hash]
- compileall command line option, 1839
- InvalidCharacterErr, 1169
- InvalidModificationErr, 1169
- InvalidOperation (*clase en decimal*), 316
- InvalidStateErr, 1169
- InvalidStateError, 836, 896
- InvalidURL, 1252
- invert() (*en el módulo operator*), 366
- IO (*clase en typing*), 1489
- io (*módulo*), 610
- IO_REPARSE_TAG_APPEXECLINK (*en el módulo stat*), 401
- IO_REPARSE_TAG_MOUNT_POINT (*en el módulo stat*), 401
- IO_REPARSE_TAG_SYMLINK (*en el módulo stat*), 401
- IOBase (*clase en io*), 613
- ioctl() (*en el módulo fcntl*), 1890
- ioctl() (*método de socket.socket*), 964
- IOCTL_VM_SOCKETS_GET_LOCAL_CID (*en el módulo socket*), 955
- IOError, 95
- ior() (*en el módulo operator*), 371
- io.StringIO
- objeto, 44
- ip (*atributo de ipaddress.IPv4Interface*), 1344
- ip (*atributo de ipaddress.IPv6Interface*), 1344
- ip_address() (*en el módulo ipaddress*), 1333
- ip_interface() (*en el módulo ipaddress*), 1334
- ip_network() (*en el módulo ipaddress*), 1333
- ipaddress (*módulo*), 1333
- ipow() (*en el módulo operator*), 371
- ipv4_mapped (*atributo de ipaddress.IPv6Address*), 1337
- IPv4Address (*clase en ipaddress*), 1334
- IPv4Interface (*clase en ipaddress*), 1344
- IPv4Network (*clase en ipaddress*), 1338
- IPV6_ENABLED (*en el módulo test.support*), 1615
- IPv6Address (*clase en ipaddress*), 1336
- IPv6Interface (*clase en ipaddress*), 1344
- IPv6Network (*clase en ipaddress*), 1341
- irshift() (*en el módulo operator*), 371
- is
- operador, 30
- is not
- operador, 30
- is_() (*en el módulo operator*), 366
- is_absolute() (*método de pathlib.PurePath*), 380
- is_active() (*método de asyncio.AbstractChildWatcher*), 937
- is_alive() (*método de multiprocessing.Process*), 789
- is_alive() (*método de threading.Thread*), 773
- is_android (*en el módulo test.support*), 1615
- is_annotated() (*método de symtable.Symbol*), 1825
- is_assigned() (*método de symtable.Symbol*), 1826

- `is_attachment()` (método de `email.message.EmailMessage`), 1043
- `is_authenticated()` (método de `urllib.request.HTTPPasswordMgrWithPriorAuth`), 1230
- `is_block_device()` (método de `pathlib.Path`), 385
- `is_blocked()` (método de `http.cookiejar.DefaultCookiePolicy`), 1316
- `is_canonical()` (método de `decimal.Context`), 312
- `is_canonical()` (método de `decimal.Decimal`), 305
- `is_char_device()` (método de `pathlib.Path`), 385
- `IS_CHARACTER_JUNK()` (en el módulo `difflib`), 134
- `is_check_supported()` (en el módulo `lzma`), 483
- `is_closed()` (método de `asyncio.loop`), 899
- `is_closing()` (método de `asyncio.BaseTransport`), 923
- `is_closing()` (método de `asyncio.StreamWriter`), 882
- `is_dataclass()` (en el módulo `dataclasses`), 1726
- `is_declared_global()` (método de `symtable.Symbol`), 1825
- `is_dir()` (método de `os.DirEntry`), 585
- `is_dir()` (método de `pathlib.Path`), 384
- `is_dir()` (método de `zipfile.Path`), 490
- `is_dir()` (método de `zipfile.ZipInfo`), 492
- `is_enabled()` (en el módulo `faulthandler`), 1638
- `is_expired()` (método de `http.cookiejar.Cookie`), 1319
- `is_fifo()` (método de `pathlib.Path`), 385
- `is_file()` (método de `os.DirEntry`), 585
- `is_file()` (método de `pathlib.Path`), 384
- `is_file()` (método de `zipfile.Path`), 490
- `is_finalizing()` (en el módulo `sys`), 1702
- `is_finite()` (método de `decimal.Context`), 312
- `is_finite()` (método de `decimal.Decimal`), 305
- `is_free()` (método de `symtable.Symbol`), 1825
- `is_global` (atributo de `ipaddress.IPv4Address`), 1335
- `is_global` (atributo de `ipaddress.IPv6Address`), 1336
- `is_global()` (método de `symtable.Symbol`), 1825
- `is_hop_by_hop()` (en el módulo `wsgiref.util`), 1211
- `is_imported()` (método de `symtable.Symbol`), 1825
- `is_infinite()` (método de `decimal.Context`), 312
- `is_infinite()` (método de `decimal.Decimal`), 305
- `is_integer()` (método de `float`), 34
- `is_jython` (en el módulo `test.support`), 1615
- `IS_LINE_JUNK()` (en el módulo `difflib`), 134
- `is_linetouched()` (método de `curses.window`), 715
- `is_link_local` (atributo de `ipaddress.IPv4Address`), 1336
- `is_link_local` (atributo de `ipaddress.IPv4Network`), 1339
- `is_link_local` (atributo de `ipaddress.IPv6Address`), 1336
- `is_link_local` (atributo de `ipaddress.IPv6Network`), 1342
- `is_local()` (método de `symtable.Symbol`), 1825
- `is_loopback` (atributo de `ipaddress.IPv4Address`), 1336
- `is_loopback` (atributo de `ipaddress.IPv4Network`), 1339
- `is_loopback` (atributo de `ipaddress.IPv6Address`), 1336
- `is_loopback` (atributo de `ipaddress.IPv6Network`), 1342
- `is_mount()` (método de `pathlib.Path`), 384
- `is_multicast` (atributo de `ipaddress.IPv4Address`), 1335
- `is_multicast` (atributo de `ipaddress.IPv4Network`), 1339
- `is_multicast` (atributo de `ipaddress.IPv6Address`), 1336
- `is_multicast` (atributo de `ipaddress.IPv6Network`), 1342
- `is_multipart()` (método de `email.message.EmailMessage`), 1040
- `is_multipart()` (método de `email.message.Message`), 1077
- `is_namespace()` (método de `symtable.Symbol`), 1826
- `is_nan()` (método de `decimal.Context`), 312
- `is_nan()` (método de `decimal.Decimal`), 305
- `is_nested()` (método de `symtable.SymbolTable`), 1824
- `is_nonlocal()` (método de `symtable.Symbol`), 1825
- `is_normal()` (método de `decimal.Context`), 312
- `is_normal()` (método de `decimal.Decimal`), 305
- `is_normalized()` (en el módulo `unicodedata`), 145
- `is_not()` (en el módulo `operator`), 366
- `is_not_allowed()` (método de `http.cookiejar.DefaultCookiePolicy`), 1317
- `is_optimized()` (método de `symtable.SymbolTable`), 1824
- `is_package()` (método de `importlib.abc.InspectLoader`), 1796
- `is_package()` (método de `importlib.abc.SourceLoader`), 1797
- `is_package()` (método de `importlib.machinery.ExtensionFileLoader`), 1802
- `is_package()` (método de `importlib.machinery.SourceFileLoader`), 1801
- `is_package()` (método de `importlib.machinery.SourcelessFileLoader`), 1802
- `is_package()` (método de `zipimport.zipimporter`), 1782
- `is_parameter()` (método de `symtable.Symbol`), 1825
- `is_private` (atributo de `ipaddress.IPv4Address`), 1335
- `is_private` (atributo de `ipaddress.IPv4Network`), 1339
- `is_private` (atributo de `ipaddress.IPv6Address`), 1336
- `is_private` (atributo de `ipaddress.IPv6Network`), 1342

- `is_python_build()` (en el módulo `sysconfig`), 1713
`is_qnan()` (método de `decimal.Context`), 312
`is_qnan()` (método de `decimal.Decimal`), 305
`is_reading()` (método de `asyncio.ReadTransport`), 924
`is_referenced()` (método de `symtable.Symbol`), 1825
`is_reserved` (atributo de `ipaddress.IPv4Address`), 1336
`is_reserved` (atributo de `ipaddress.IPv4Network`), 1339
`is_reserved` (atributo de `ipaddress.IPv6Address`), 1336
`is_reserved` (atributo de `ipaddress.IPv6Network`), 1342
`is_reserved()` (método de `pathlib.PurePath`), 380
`is_resource()` (en el módulo `importlib.resources`), 1799
`is_resource()` (método de `importlib.abc.ResourceReader`), 1795
`is_resource_enabled()` (en el módulo `test.support`), 1616
`is_running()` (método de `asyncio.loop`), 899
`is_safe` (atributo de `uuid.UUID`), 1292
`is_serving()` (método de `asyncio.Server`), 915
`is_set()` (método de `asyncio.Event`), 886
`is_set()` (método de `threading.Event`), 779
`is_signed()` (método de `decimal.Context`), 312
`is_signed()` (método de `decimal.Decimal`), 305
`is_site_local` (atributo de `ipaddress.IPv6Address`), 1336
`is_site_local` (atributo de `ipaddress.IPv6Network`), 1342
`is_snan()` (método de `decimal.Context`), 312
`is_snan()` (método de `decimal.Decimal`), 306
`is_socket()` (método de `pathlib.Path`), 384
`is_subnormal()` (método de `decimal.Context`), 312
`is_subnormal()` (método de `decimal.Decimal`), 306
`is_symlink()` (método de `os.DirEntry`), 586
`is_symlink()` (método de `pathlib.Path`), 384
`is_tarfile()` (en el módulo `tarfile`), 496
`is_term_resized()` (en el módulo `curses`), 708
`is_tracing()` (en el módulo `tracemalloc`), 1666
`is_tracked()` (en el módulo `gc`), 1756
`is_unspecified` (atributo de `ipaddress.IPv4Address`), 1336
`is_unspecified` (atributo de `ipaddress.IPv4Network`), 1339
`is_unspecified` (atributo de `ipaddress.IPv6Address`), 1336
`is_unspecified` (atributo de `ipaddress.IPv6Network`), 1342
`is_wintouched()` (método de `curses.window`), 715
`is_zero()` (método de `decimal.Context`), 312
`is_zero()` (método de `decimal.Decimal`), 306
`is_zipfile()` (en el módulo `zipfile`), 486
`isabs()` (en el módulo `os.path`), 392
`isabstract()` (en el módulo `inspect`), 1761
`IsADirectoryError`, 96
`isalnum()` (en el módulo `curses.ascii`), 724
`isalnum()` (método de `bytearray`), 62
`isalnum()` (método de `bytes`), 62
`isalnum()` (método de `str`), 47
`isalpha()` (en el módulo `curses.ascii`), 724
`isalpha()` (método de `bytearray`), 62
`isalpha()` (método de `bytes`), 62
`isalpha()` (método de `str`), 47
`isascii()` (en el módulo `curses.ascii`), 724
`isascii()` (método de `bytearray`), 63
`isascii()` (método de `bytes`), 63
`isascii()` (método de `str`), 47
`isasyncgen()` (en el módulo `inspect`), 1761
`isasyncgenfunction()` (en el módulo `inspect`), 1760
`isatty()` (en el módulo `os`), 568
`isatty()` (método de `chunk.Chunk`), 1358
`isatty()` (método de `io.IOBBase`), 613
`isawaitable()` (en el módulo `inspect`), 1760
`isblank()` (en el módulo `curses.ascii`), 725
`isblk()` (método de `tarfile.TarInfo`), 503
`isbuiltin()` (en el módulo `inspect`), 1761
`ischr()` (método de `tarfile.TarInfo`), 503
`isclass()` (en el módulo `inspect`), 1760
`isclose()` (en el módulo `cmath`), 297
`isclose()` (en el módulo `math`), 290
`isctrl()` (en el módulo `curses.ascii`), 725
`iscode()` (en el módulo `inspect`), 1761
`iscoroutine()` (en el módulo `asyncio`), 878
`iscoroutine()` (en el módulo `inspect`), 1760
`iscoroutinefunction()` (en el módulo `asyncio`), 878
`iscoroutinefunction()` (en el módulo `inspect`), 1760
`isctrl()` (en el módulo `curses.ascii`), 725
`isDaemon()` (método de `threading.Thread`), 773
`isdatadescriptor()` (en el módulo `inspect`), 1761
`isdecimal()` (método de `str`), 47
`isdev()` (método de `tarfile.TarInfo`), 503
`isdigit()` (en el módulo `curses.ascii`), 725
`isdigit()` (método de `bytearray`), 63
`isdigit()` (método de `bytes`), 63
`isdigit()` (método de `str`), 47
`isdir()` (en el módulo `os.path`), 392
`isdir()` (método de `tarfile.TarInfo`), 502
`isdisjoint()` (método de `frozenset`), 76
`isdown()` (en el módulo `turtle`), 1397
`iselement()` (en el módulo `xml.etree.ElementTree`), 1150
`isenabled()` (en el módulo `gc`), 1754
`isEnabledFor()` (método de `logging.Logger`), 666
`isendwin()` (en el módulo `curses`), 708
`ISEOF()` (en el módulo `token`), 1827
`isexpr()` (en el módulo `parser`), 1816
`isexpr()` (método de `parser.ST`), 1816
`isfifo()` (método de `tarfile.TarInfo`), 503
`isfile()` (en el módulo `os.path`), 392

- `isfile()` (método de `tarfile.TarInfo`), 502
- `isfinite()` (en el módulo `cmath`), 297
- `isfinite()` (en el módulo `math`), 290
- `isfirstline()` (en el módulo `fileinput`), 395
- `isframe()` (en el módulo `inspect`), 1761
- `isfunction()` (en el módulo `inspect`), 1760
- `isfuture()` (en el módulo `asyncio`), 918
- `isgenerator()` (en el módulo `inspect`), 1760
- `isgeneratorfunction()` (en el módulo `inspect`), 1760
- `isgetsetdescriptor()` (en el módulo `inspect`), 1761
- `isgraph()` (en el módulo `curses.ascii`), 725
- `isidentifier()` (método de `str`), 47
- `isinf()` (en el módulo `cmath`), 297
- `isinf()` (en el módulo `math`), 290
- `isinstance(2to3 fixer)`, 1610
- `isinstance()` (función incorporada), 14
- `iskeyword()` (en el módulo `keyword`), 1830
- `isleap()` (en el módulo `calendar`), 216
- `islice()` (en el módulo `itertools`), 349
- `islink()` (en el módulo `os.path`), 392
- `islnk()` (método de `tarfile.TarInfo`), 503
- `islower()` (en el módulo `curses.ascii`), 725
- `islower()` (método de `bytearray`), 63
- `islower()` (método de `bytes`), 63
- `islower()` (método de `str`), 47
- `ismemberdescriptor()` (en el módulo `inspect`), 1761
- `ismeta()` (en el módulo `curses.ascii`), 725
- `ismethod()` (en el módulo `inspect`), 1760
- `ismethoddescriptor()` (en el módulo `inspect`), 1761
- `ismodule()` (en el módulo `inspect`), 1760
- `ismount()` (en el módulo `os.path`), 392
- `isnan()` (en el módulo `cmath`), 297
- `isnan()` (en el módulo `math`), 290
- `ISNONTERMINAL()` (en el módulo `token`), 1827
- `isnumeric()` (método de `str`), 47
- `isocalendar()` (método de `datetime.date`), 185
- `isocalendar()` (método de `datetime.datetime`), 194
- `isoformat()` (método de `datetime.date`), 185
- `isoformat()` (método de `datetime.datetime`), 194
- `isoformat()` (método de `datetime.time`), 199
- `IsolatedAsyncioTestCase` (clase en `unittest`), 1536
- `isolation_level` (atributo de `sqlite3.Connection`), 450
- `isowekday()` (método de `datetime.date`), 185
- `isowekday()` (método de `datetime.datetime`), 194
- `isprint()` (en el módulo `curses.ascii`), 725
- `isprintable()` (método de `str`), 48
- `ispunct()` (en el módulo `curses.ascii`), 725
- `isqrt()` (en el módulo `math`), 290
- `isreadable()` (en el módulo `pprint`), 261
- `isreadable()` (método de `pprint.PrettyPrinter`), 261
- `isrecursive()` (en el módulo `pprint`), 261
- `isrecursive()` (método de `pprint.PrettyPrinter`), 261
- `isreg()` (método de `tarfile.TarInfo`), 502
- `isReservedKey()` (método de `http.cookies.Morsel`), 1310
- `isroutine()` (en el módulo `inspect`), 1761
- `isSameNode()` (método de `xml.dom.Node`), 1164
- `isspace()` (en el módulo `curses.ascii`), 725
- `isspace()` (método de `bytearray`), 63
- `isspace()` (método de `bytes`), 63
- `isspace()` (método de `str`), 48
- `isstdin()` (en el módulo `fileinput`), 395
- `issubclass()` (función incorporada), 15
- `issubset()` (método de `frozenset`), 76
- `issuite()` (en el módulo `parser`), 1816
- `issuite()` (método de `parser.ST`), 1816
- `issuperset()` (método de `frozenset`), 76
- `issym()` (método de `tarfile.TarInfo`), 503
- `ISTERMINAL()` (en el módulo `token`), 1827
- `istitle()` (método de `bytearray`), 63
- `istitle()` (método de `bytes`), 63
- `istitle()` (método de `str`), 48
- `itraceback()` (en el módulo `inspect`), 1761
- `isub()` (en el módulo `operator`), 371
- `isupper()` (en el módulo `curses.ascii`), 725
- `isupper()` (método de `bytearray`), 63
- `isupper()` (método de `bytes`), 63
- `isupper()` (método de `str`), 48
- `isvisible()` (en el módulo `turtle`), 1400
- `isxdigit()` (en el módulo `curses.ascii`), 725
- `item()` (método de `tkinter.ttk.Treeview`), 1454
- `item()` (método de `xml.dom.NamedNodeMap`), 1168
- `item()` (método de `xml.dom.NodeList`), 1165
- `itemgetter()` (en el módulo `operator`), 368
- `items()` (método de `configparser.ConfigParser`), 532
- `items()` (método de `contextvars.Context`), 861
- `items()` (método de `dict`), 79
- `items()` (método de `email.message.EmailMessage`), 1041
- `items()` (método de `email.message.Message`), 1079
- `items()` (método de `mailbox.Mailbox`), 1108
- `items()` (método de `types.MappingProxyType`), 257
- `items()` (método de `xml.etree.ElementTree.Element`), 1154
- `itemsiz` (atributo de `array.array`), 244
- `itemsiz` (atributo de `memoryview`), 74
- `ItemsView` (clase en `collections.abc`), 235
- `ItemsView` (clase en `typing`), 1487
- `iter()` (función incorporada), 15
- `iter()` (método de `xml.etree.ElementTree.Element`), 1155
- `iter()` (método de `xml.etree.ElementTree.ElementTree`), 1156
- `iter_attachments()` (método de `email.message.EmailMessage`), 1044
- `iter_child_nodes()` (en el módulo `ast`), 1822
- `iter_fields()` (en el módulo `ast`), 1822
- `iter_importers()` (en el módulo `pkgutil`), 1784
- `iter_modules()` (en el módulo `pkgutil`), 1784

- `iter_parts()` (método `email.message.EmailMessage`), 1044
- `iter_unpack()` (en el módulo `struct`), 154
- `iter_unpack()` (método de `struct.Struct`), 158
- `iterable`, 1941
- `Iterable` (clase en `collections.abc`), 234
- `Iterable` (clase en `typing`), 1485
- `iterable` asincrónico, 1936
- `iterador`, 1941
- `iterador` asincrónico, 1936
- `iterador` generador, 1940
- `iterador` generador asincrónico, 1936
- `Iterator` (clase en `collections.abc`), 235
- `Iterator` (clase en `typing`), 1485
- `iterator` protocol, 37
- `iterdecode()` (en el módulo `codecs`), 161
- `iterdir()` (método de `pathlib.Path`), 385
- `iterdir()` (método de `zipfile.Path`), 490
- `iterdump()` (método de `sqlite3.Connection`), 455
- `iterencode()` (en el módulo `codecs`), 160
- `iterencode()` (método de `json.JSONEncoder`), 1102
- `iterfind()` (método `xml.etree.ElementTree.Element`), 1155
- `iterfind()` (método `xml.etree.ElementTree.ElementTree`), 1156
- `iteritems()` (método de `mailbox.Mailbox`), 1108
- `iterkeys()` (método de `mailbox.Mailbox`), 1108
- `itermonthdates()` (método de `calendar.Calendar`), 213
- `itermonthdays()` (método de `calendar.Calendar`), 213
- `itermonthdays2()` (método de `calendar.Calendar`), 213
- `itermonthdays3()` (método de `calendar.Calendar`), 213
- `itermonthdays4()` (método de `calendar.Calendar`), 213
- `iterparse()` (en el módulo `xml.etree.ElementTree`), 1150
- `itertext()` (método `xml.etree.ElementTree.Element`), 1155
- `itertools` (2to3 fixer), 1610
- `itertools` (módulo), 343
- `itertools_imports` (2to3 fixer), 1610
- `itervalues()` (método de `mailbox.Mailbox`), 1108
- `iterweekdays()` (método de `calendar.Calendar`), 213
- `ITIMER_PROF` (en el módulo `signal`), 1026
- `ITIMER_REAL` (en el módulo `signal`), 1026
- `ITIMER_VIRTUAL` (en el módulo `signal`), 1026
- `ItimerError`, 1026
- `itruediv()` (en el módulo `operator`), 371
- `ixor()` (en el módulo `operator`), 371
- J**
- `-j N`
compileall command line option, 1839
- `de Jansen, Jack`, 1133
- `java_ver()` (en el módulo `platform`), 729
- `join()` (en el módulo `os.path`), 392
- `join()` (en el módulo `shlex`), 1423
- `join()` (método de `asyncio.Queue`), 894
- `join()` (método de `bytearray`), 58
- `join()` (método de `bytes`), 58
- `join()` (método de `multiprocessing.JoinableQueue`), 794
- `join()` (método de `multiprocessing.pool.Pool`), 810
- `join()` (método de `multiprocessing.Process`), 789
- `join()` (método de `queue.Queue`), 857
- `join()` (método de `str`), 48
- `join()` (método de `threading.Thread`), 772
- `join_thread()` (en el módulo `test.support`), 1623
- `join_thread()` (método de `multiprocessing.Queue`), 793
- `JoinableQueue` (clase en `multiprocessing`), 793
- `joinpath()` (método de `pathlib.PurePath`), 380
- `js_output()` (método de `http.cookies.BaseCookie`), 1309
- `js_output()` (método de `http.cookies.Morsel`), 1310
- `json` (módulo), 1096
- `JSONDecodeError`, 1102
- `JSONDecoder` (clase en `json`), 1100
- `JSONEncoder` (clase en `json`), 1100
- `--json-lines`
json.tool command line option, 1105
- `json.tool` (módulo), 1104
- `json.tool` command line option
-h, 1105
--help, 1105
infile, 1105
--json-lines, 1105
outfile, 1105
--sort-keys, 1105
- `jump` (`pdb` command), 1644
- `JUMP_ABSOLUTE` (opcode), 1852
- `JUMP_FORWARD` (opcode), 1851
- `JUMP_IF_FALSE_OR_POP` (opcode), 1851
- `JUMP_IF_TRUE_OR_POP` (opcode), 1851
- K**
- `-k`
unittest command line option, 1522
- `kbhit()` (en el módulo `msvcrt`), 1868
- `KDEDIR`, 1201
- `kevent()` (en el módulo `select`), 1008
- `key` (atributo de `http.cookies.Morsel`), 1310
- `KEY_ALL_ACCESS` (en el módulo `winreg`), 1875
- `KEY_CREATE_LINK` (en el módulo `winreg`), 1875
- `KEY_CREATE_SUB_KEY` (en el módulo `winreg`), 1875
- `KEY_ENUMERATE_SUB_KEYS` (en el módulo `winreg`), 1875
- `KEY_EXECUTE` (en el módulo `winreg`), 1875
- `KEY_NOTIFY` (en el módulo `winreg`), 1875
- `KEY_QUERY_VALUE` (en el módulo `winreg`), 1875
- `KEY_READ` (en el módulo `winreg`), 1875

KEY_SET_VALUE (en el módulo *winreg*), 1875
KEY_WOW64_32KEY (en el módulo *winreg*), 1875
KEY_WOW64_64KEY (en el módulo *winreg*), 1875
KEY_WRITE (en el módulo *winreg*), 1875
KeyboardInterrupt, 91
KeyError, 91
keylog_filename (atributo de *ssl.SSLContext*), 995
keyname() (en el módulo *curses*), 708
keypad() (método de *curses.window*), 716
keyrefs() (método de *weakref.WeakKeyDictionary*), 248
keys() (método de *contextvars.Context*), 861
keys() (método de *dict*), 79
keys() (método de *email.message.EmailMessage*), 1041
keys() (método de *email.message.Message*), 1079
keys() (método de *mailbox.Mailbox*), 1108
keys() (método de *sqlite3.Row*), 459
keys() (método de *types.MappingProxyType*), 257
keys() (método de *xml.etree.ElementTree.Element*), 1154
KeysView (clase en *collections.abc*), 235
KeysView (clase en *typing*), 1487
keyword (módulo), 1830
keywords (atributo de *functools.partial*), 365
kill() (en el módulo *os*), 599
kill() (método de *asyncio.subprocess.Process*), 892
kill() (método de *asyncio.SubprocessTransport*), 925
kill() (método de *multiprocessing.Process*), 790
kill() (método de *subprocess.Popen*), 845
kill_python() (en el módulo *test.support.script_helper*), 1628
killchar() (en el módulo *curses*), 708
killpg() (en el módulo *os*), 599
kind (atributo de *inspect.Parameter*), 1764
knownfiles (en el módulo *mimetypes*), 1125
kqueue() (en el módulo *select*), 1008
KqueueSelector (clase en *selectors*), 1016
kwargs (atributo de *inspect.BoundArguments*), 1766
kwlist (en el módulo *keyword*), 1830

L
-l
 compileall command line option, 1838
 pickletools command line option, 1855
 trace command line option, 1660
L (en el módulo *re*), 116
-l <tarfile>
 tarfile command line option, 507
-l <zipfile>
 zipfile command line option, 494
LabelEntry (clase en *tkinter.tix*), 1460
LabelFrame (clase en *tkinter.tix*), 1460
lambda, 1942
LambdaType (en el módulo *types*), 254
LANG, 1367, 1369, 1377, 1379
language
 C, 31, 32
LANGUAGE, 1367, 1369
large files, 1881
LARGEST (en el módulo *test.support*), 1616
LargeZipFile, 485
last() (método de *nnplib.NNTP*), 1276
last_accepted (atributo de *multiprocessing.connection.Listener*), 813
last_traceback (en el módulo *sys*), 1702
last_type (en el módulo *sys*), 1702
last_value (en el módulo *sys*), 1702
lastChild (atributo de *xml.dom.Node*), 1164
lastcmd (atributo de *cmd.Cmd*), 1420
lastgroup (atributo de *re.Match*), 124
lastindex (atributo de *re.Match*), 124
lastResort (en el módulo *logging*), 680
lastrowid (atributo de *sqlite3.Cursor*), 458
layout() (método de *tkinter.ttk.Style*), 1456
lazycache() (en el módulo *linecache*), 410
LazyLoader (clase en *importlib.util*), 1806
LBRACE (en el módulo *token*), 1828
LBYL, 1942
LC_ALL, 1367, 1369
LC_ALL (en el módulo *locale*), 1381
LC_COLLATE (en el módulo *locale*), 1381
LC_CTYPE (en el módulo *locale*), 1381
LC_MESSAGES, 1367, 1369
LC_MESSAGES (en el módulo *locale*), 1381
LC_MONETARY (en el módulo *locale*), 1381
LC_NUMERIC (en el módulo *locale*), 1381
LC_TIME (en el módulo *locale*), 1381
lchflags() (en el módulo *os*), 579
lchmod() (en el módulo *os*), 579
lchmod() (método de *pathlib.Path*), 385
lchown() (en el módulo *os*), 579
ldexp() (en el módulo *math*), 290
ldgettext() (en el módulo *gettext*), 1368
ldngettext() (en el módulo *gettext*), 1368
le() (en el módulo *operator*), 365
leapdays() (en el módulo *calendar*), 216
leaveok() (método de *curses.window*), 716
left (atributo de *filecmp.dircmp*), 402
left() (en el módulo *turtle*), 1390
left_list (atributo de *filecmp.dircmp*), 402
left_only (atributo de *filecmp.dircmp*), 402
LEFTSHIFT (en el módulo *token*), 1828
LEFTSHIFTEQUAL (en el módulo *token*), 1828
len
 función incorporada, 38, 77
len() (función incorporada), 15
length (atributo de *xml.dom.NamedNodeMap*), 1168
length (atributo de *xml.dom.NodeList*), 1165
length_hint() (en el módulo *operator*), 367
LESS (en el módulo *token*), 1827
LESSEQUAL (en el módulo *token*), 1828
lexists() (en el módulo *os.path*), 391

- `lgamma()` (en el módulo `math`), 294
- `gettext()` (en el módulo `gettext`), 1368
- `gettext()` (método de `gettext.GNUTranslations`), 1372
- `gettext()` (método de `gettext.NullTranslations`), 1370
- `lib2to3` (módulo), 1612
- `libc_ver()` (en el módulo `platform`), 730
- `library` (atributo de `ssl.SSLError`), 975
- `library` (en el módulo `dbm.ndbm`), 445
- `LibraryLoader` (clase en `ctypes`), 756
- `license` (variable incorporada), 28
- `LifoQueue` (clase en `asyncio`), 895
- `LifoQueue` (clase en `queue`), 856
- light-weight processes, 862
- `limit_denominator()` (método de `Fractions.Fraction`), 325
- `LimitOverrunError`, 896
- `lin2adpcm()` (en el módulo `audioop`), 1348
- `lin2alaw()` (en el módulo `audioop`), 1348
- `lin2lin()` (en el módulo `audioop`), 1348
- `lin2ulaw()` (en el módulo `audioop`), 1349
- `line()` (método de `msilib.Dialog`), 1866
- `line_buffering` (atributo de `io.TextIOWrapper`), 620
- `line_num` (atributo de `csv.csvreader`), 516
- line-buffered I/O, 19
- `linecache` (módulo), 410
- `lineno` (atributo de `ast.AST`), 1818
- `lineno` (atributo de `doctest.DocTest`), 1512
- `lineno` (atributo de `doctest.Example`), 1512
- `lineno` (atributo de `json.JSONDecodeError`), 1102
- `lineno` (atributo de `pyclbr.Class`), 1836
- `lineno` (atributo de `pyclbr.Function`), 1835
- `lineno` (atributo de `re.error`), 120
- `lineno` (atributo de `shlex.shlex`), 1426
- `lineno` (atributo de `SyntaxError`), 93
- `lineno` (atributo de `traceback.TracebackException`), 1749
- `lineno` (atributo de `tracemalloc.Filter`), 1667
- `lineno` (atributo de `tracemalloc.Frame`), 1668
- `lineno` (atributo de `xml.parsers.expat.ExpatError`), 1194
- `lineno()` (en el módulo `fileinput`), 395
- `LINES`, 707, 711
- `lines` (atributo de `os.terminal_size`), 575
- `linesep` (atributo de `email.policy.Policy`), 1055
- `linesep` (en el módulo `os`), 608
- `lineterminator` (atributo de `csv.Dialect`), 515
- `LineTooLong`, 1252
- `link()` (en el módulo `os`), 579
- `link_to()` (método de `pathlib.Path`), 388
- `linkname` (atributo de `tarfile.TarInfo`), 502
- `LinkOutsideDestinationError`, 497
- `list`
 - objeto, 40, 41
 - type, operations on, 40
- `List` (clase en `typing`), 1486
- `list` (clase incorporada), 41
- `list` (`pdb` command), 1644
- `--list <tarfile>`
 - tarfile command line option, 507
- `--list <zipfile>`
 - zipfile command line option, 494
- `list()` (método de `imaplib.IMAP4`), 1268
- `list()` (método de `multiprocessing.managers.SyncManager`), 805
- `list()` (método de `nntplib.NNTP`), 1275
- `list()` (método de `poplib.POP3`), 1264
- `list()` (método de `tarfile.TarFile`), 499
- `LIST_APPEND` (opcode), 1848
- `list_dialects()` (en el módulo `csv`), 513
- `list_folders()` (método de `mailbox.Maildir`), 1110
- `list_folders()` (método de `mailbox.MH`), 1112
- `lista`, 1942
- `listdir()` (en el módulo `os`), 579
- `listen()` (en el módulo `logging.config`), 682
- `listen()` (en el módulo `turtle`), 1408
- `listen()` (método de `asyncore.dispatcher`), 1019
- `listen()` (método de `socket.socket`), 964
- `Listener` (clase en `multiprocessing.connection`), 812
- `--listfuncs`
 - trace command line option, 1660
- `listMethods()` (método de `xmlrpc.client.ServerProxy.system`), 1322
- `ListNoteBook` (clase en `tkinter.tix`), 1461
- `listxattr()` (en el módulo `os`), 595
- `Literal` (en el módulo `typing`), 1494
- `literal_eval()` (en el módulo `ast`), 1821
- literals
 - binary, 31
 - complex number, 31
 - floating point, 31
 - hexadecimal, 31
 - integer, 31
 - numeric, 31
 - octal, 31
- `LittleEndianStructure` (clase en `ctypes`), 766
- `ljust()` (método de `bytearray`), 59
- `ljust()` (método de `bytes`), 59
- `ljust()` (método de `str`), 48
- `LK_LOCK` (en el módulo `msvcrt`), 1867
- `LK_NBLCK` (en el módulo `msvcrt`), 1867
- `LK_NBRLOCK` (en el módulo `msvcrt`), 1867
- `LK_RLCK` (en el módulo `msvcrt`), 1867
- `LK_UNLCK` (en el módulo `msvcrt`), 1867
- `ll` (`pdb` command), 1644
- `LMTP` (clase en `smtplib`), 1279
- `ln()` (método de `decimal.Context`), 312
- `ln()` (método de `decimal.Decimal`), 306
- `LNAME`, 705
- `lngettext()` (en el módulo `gettext`), 1368
- `lngettext()` (método de `gettext.GNUTranslations`), 1372
- `lngettext()` (método de `gettext.NullTranslations`), 1370

- `load()` (en el módulo `json`), 1099
- `load()` (en el módulo `marshal`), 441
- `load()` (en el módulo `pickle`), 424
- `load()` (en el módulo `plistlib`), 539
- `load()` (método de clase de `tracemalloc.Snapshot`), 1668
- `load()` (método de `http.cookiejar.FileCookieJar`), 1314
- `load()` (método de `http.cookies.BaseCookie`), 1309
- `load()` (método de `pickle.Unpickler`), 426
- `LOAD_ATTR` (opcode), 1851
- `LOAD_BUILD_CLASS` (opcode), 1849
- `load_cert_chain()` (método de `ssl.SSLContext`), 990
- `LOAD_CLASSDEREF` (opcode), 1852
- `LOAD_CLOSURE` (opcode), 1852
- `LOAD_CONST` (opcode), 1850
- `load_default_certs()` (método de `ssl.SSLContext`), 990
- `LOAD_DEREF` (opcode), 1852
- `load_dh_params()` (método de `ssl.SSLContext`), 993
- `load_extension()` (método de `sqlite3.Connection`), 454
- `LOAD_FAST` (opcode), 1852
- `LOAD_GLOBAL` (opcode), 1852
- `LOAD_METHOD` (opcode), 1853
- `load_module()` (en el módulo `imp`), 1929
- `load_module()` (método de `importlib.abc.FileLoader`), 1796
- `load_module()` (método de `importlib.abc.InspectLoader`), 1796
- `load_module()` (método de `importlib.abc.Loader`), 1793
- `load_module()` (método de `importlib.abc.SourceLoader`), 1797
- `load_module()` (método de `importlib.machinery.SourceFileLoader`), 1801
- `load_module()` (método de `importlib.machinery.SourcelessFileLoader`), 1802
- `load_module()` (método de `zipimport.zipimporter`), 1782
- `LOAD_NAME` (opcode), 1850
- `load_package_tests()` (en el módulo `test.support`), 1624
- `load_verify_locations()` (método de `ssl.SSLContext`), 991
- `loader` (atributo de `importlib.machinery.ModuleSpec`), 1803
- `Loader` (clase en `importlib.abc`), 1793
- `loader_state` (atributo de `importlib.machinery.ModuleSpec`), 1803
- `LoadError`, 1312
- `LoadKey()` (en el módulo `winreg`), 1871
- `LoadLibrary()` (método de `ctypes.LibraryLoader`), 756
- `loads()` (en el módulo `json`), 1099
- `loads()` (en el módulo `marshal`), 442
- `loads()` (en el módulo `pickle`), 424
- `loads()` (en el módulo `plistlib`), 540
- `loads()` (en el módulo `xmlrpc.client`), 1326
- `loadTestsFromModule()` (método de `unittest.TestLoader`), 1539
- `loadTestsFromName()` (método de `unittest.TestLoader`), 1539
- `loadTestsFromNames()` (método de `unittest.TestLoader`), 1540
- `loadTestsFromTestCase()` (método de `unittest.TestLoader`), 1539
- `local` (clase en `threading`), 771
- `localcontext()` (en el módulo `decimal`), 309
- `LOCALE` (en el módulo `re`), 116
- `locale` (módulo), 1376
- `localeconv()` (en el módulo `locale`), 1377
- `LocaleHTMLCalendar` (clase en `calendar`), 215
- `LocaleTextCalendar` (clase en `calendar`), 215
- `localName` (atributo de `xml.dom.Attr`), 1168
- `localName` (atributo de `xml.dom.Node`), 1164
- `--locals`
 - `unittest` command line option, 1522
- `locals()` (función incorporada), 15
- `localtime()` (en el módulo `email.utils`), 1092
- `localtime()` (en el módulo `time`), 625
- `Locator` (clase en `xml.sax.xmlreader`), 1185
- `Lock` (clase en `asyncio`), 885
- `Lock` (clase en `multiprocessing`), 798
- `Lock` (clase en `threading`), 774
- `lock()` (método de `mailbox.Babyl`), 1114
- `lock()` (método de `mailbox.Mailbox`), 1109
- `lock()` (método de `mailbox.Maildir`), 1111
- `lock()` (método de `mailbox.mbox`), 1111
- `lock()` (método de `mailbox.MH`), 1113
- `lock()` (método de `mailbox.MMDF`), 1114
- `Lock()` (método de `multiprocessing.managers.SyncManager`), 804
- `lock_held()` (en el módulo `imp`), 1930
- `locked()` (método de `_thread.lock`), 863
- `locked()` (método de `asyncio.Condition`), 887
- `locked()` (método de `asyncio.Lock`), 886
- `locked()` (método de `asyncio.Semaphore`), 888
- `locked()` (método de `threading.Lock`), 774
- `lockf()` (en el módulo `fcntl`), 1891
- `lockf()` (en el módulo `os`), 568
- `locking()` (en el módulo `msvcrt`), 1867
- `LockType` (en el módulo `_thread`), 862
- `log()` (en el módulo `cmath`), 296
- `log()` (en el módulo `logging`), 678
- `log()` (en el módulo `math`), 291
- `log()` (método de `logging.Logger`), 668
- `log1p()` (en el módulo `math`), 292
- `log2()` (en el módulo `math`), 292
- `log10()` (en el módulo `cmath`), 296
- `log10()` (en el módulo `math`), 292
- `log10()` (método de `decimal.Context`), 312
- `log10()` (método de `decimal.Decimal`), 306
- `log_date_time_string()` (método de `http.server.BaseHTTPRequestHandler`),

- 1305
 log_error() (método de *http.server.BaseHTTPRequestHandler*), 1305
 log_exception() (método de *wsgi-ref.handlers.BaseHandler*), 1217
 log_message() (método de *http.server.BaseHTTPRequestHandler*), 1305
 log_request() (método de *http.server.BaseHTTPRequestHandler*), 1305
 log_to_stderr() (en el módulo *multiprocessing*), 815
 logb() (método de *decimal.Context*), 312
 logb() (método de *decimal.Decimal*), 306
 Logger (clase en *logging*), 666
 LoggerAdapter (clase en *logging*), 676
 logging
 Errors, 665
 logging (módulo), 665
 logging.config (módulo), 681
 logging.handlers (módulo), 692
 logical_and() (método de *decimal.Context*), 312
 logical_and() (método de *decimal.Decimal*), 306
 logical_invert() (método de *decimal.Context*), 312
 logical_invert() (método de *decimal.Decimal*), 306
 logical_or() (método de *decimal.Context*), 312
 logical_or() (método de *decimal.Decimal*), 306
 logical_xor() (método de *decimal.Context*), 312
 logical_xor() (método de *decimal.Decimal*), 306
 login() (método de *ftplib.FTP*), 1259
 login() (método de *imaplib.IMAP4*), 1268
 login() (método de *nnplib.NNTP*), 1274
 login() (método de *smtpplib.SMTP*), 1281
 login_cram_md5() (método de *imaplib.IMAP4*), 1268
 LOGNAME, 562, 705
 lognormvariate() (en el módulo *random*), 329
 logout() (método de *imaplib.IMAP4*), 1268
 LogRecord (clase en *logging*), 673
 long (2to3 fixer), 1610
 longMessage (atributo de *unittest.TestCase*), 1535
 longname() (en el módulo *curses*), 708
 lookup() (en el módulo *codecs*), 159
 lookup() (en el módulo *unicodedata*), 144
 lookup() (método de *symtable.SymbolTable*), 1825
 lookup() (método de *tkinter.ttk.Style*), 1456
 lookup_error() (en el módulo *codecs*), 162
 LookupError, 90
 loop() (en el módulo *asyncore*), 1017
 lower() (método de *bytearray*), 64
 lower() (método de *bytes*), 64
 lower() (método de *str*), 48
 LPAR (en el módulo *token*), 1827
 lpAttributeList (atributo de *subprocess.STARTUPINFO*), 846
 lru_cache() (en el módulo *functools*), 358
 lseek() (en el módulo *os*), 569
 lshift() (en el módulo *operator*), 366
 LSQB (en el módulo *token*), 1827
 lstat() (en el módulo *os*), 580
 lstat() (método de *pathlib.Path*), 385
 lstrip() (método de *bytearray*), 60
 lstrip() (método de *bytes*), 60
 lstrip() (método de *str*), 48
 lsub() (método de *imaplib.IMAP4*), 1268
 lt() (en el módulo *operator*), 365
 lt() (en el módulo *turtle*), 1390
 LWPCookieJar (clase en *http.cookiejar*), 1315
 lzma (módulo), 479
 LZMACompressor (clase en *lzma*), 481
 LZMADecompressor (clase en *lzma*), 482
 LZMAError, 480
 LZMAFile (clase en *lzma*), 480
- ## M
- m
 pickletools command line option, 1855
 trace command line option, 1660
 M (en el módulo *re*), 116
 -m <mainfn>
 zipapp command line option, 1684
 mac_ver() (en el módulo *platform*), 730
 machine() (en el módulo *platform*), 727
 macros (atributo de *netrc.netrc*), 536
 MADV_AUTOSYNC (en el módulo *mmap*), 1034
 MADV_CORE (en el módulo *mmap*), 1034
 MADV_DODUMP (en el módulo *mmap*), 1034
 MADV_DOFORK (en el módulo *mmap*), 1034
 MADV_DONTDUMP (en el módulo *mmap*), 1034
 MADV_DONTFORK (en el módulo *mmap*), 1034
 MADV_DONTNEED (en el módulo *mmap*), 1034
 MADV_FREE (en el módulo *mmap*), 1034
 MADV_HUGEPAGE (en el módulo *mmap*), 1034
 MADV_HWPOISON (en el módulo *mmap*), 1034
 MADV_MERGEABLE (en el módulo *mmap*), 1034
 MADV_NOCORE (en el módulo *mmap*), 1034
 MADV_NOHUGEPAGE (en el módulo *mmap*), 1034
 MADV_NORMAL (en el módulo *mmap*), 1034
 MADV_NOSYNC (en el módulo *mmap*), 1034
 MADV_PROTECT (en el módulo *mmap*), 1034
 MADV_RANDOM (en el módulo *mmap*), 1034
 MADV_REMOVE (en el módulo *mmap*), 1034
 MADV_SEQUENTIAL (en el módulo *mmap*), 1034
 MADV_SOFT_OFFLINE (en el módulo *mmap*), 1034
 MADV_UNMERGEABLE (en el módulo *mmap*), 1034
 MADV_WILLNEED (en el módulo *mmap*), 1034
 madvise() (método de *mmap.mmap*), 1033
 magic
 method, 1942
 MAGIC_NUMBER (en el módulo *importlib.util*), 1803

- MagicMock (clase en `unittest.mock`), 1577
 Mailbox (clase en `mailbox`), 1107
 mailbox (módulo), 1106
 mailcap (módulo), 1105
 Maildir (clase en `mailbox`), 1110
 MaildirMessage (clase en `mailbox`), 1115
 mailfrom (atributo de `smtpd.SMTPChannel`), 1287
 MailmanProxy (clase en `smtpd`), 1286
 main() (en el módulo `py_compile`), 1837
 main() (en el módulo `site`), 1775
 main() (en el módulo `unittest`), 1544
 --main=<mainfn>
 zipapp command line option, 1684
 main_thread() (en el módulo `threading`), 770
 mainloop() (en el módulo `turtle`), 1409
 maintype (atributo de `email.headerregistry.ContentTypeHeader`), 1064
 major (atributo de `email.headerregistry.MIMEVersionHeader`), 1064
 major() (en el módulo `os`), 581
 make_alternative() (método de `email.message.EmailMessage`), 1045
 make_archive() (en el módulo `shutil`), 417
 make_bad_fd() (en el módulo `test.support`), 1622
 make_cookies() (método de `http.cookiejar.CookieJar`), 1313
 make_dataclass() (en el módulo `dataclasses`), 1725
 make_file() (método de `difflib.HtmlDiff`), 130
 MAKE_FUNCTION (opcode), 1853
 make_header() (en el módulo `email.header`), 1088
 make_legacy_pyc() (en el módulo `test.support`), 1616
 make_mixed() (método de `email.message.EmailMessage`), 1045
 make_msgid() (en el módulo `email.utils`), 1092
 make_parser() (en el módulo `xml.sax`), 1178
 make_pkg() (en el módulo `test.support.script_helper`), 1628
 make_related() (método de `email.message.EmailMessage`), 1045
 make_script() (en el módulo `test.support.script_helper`), 1628
 make_server() (en el módulo `wsgiref.simple_server`), 1212
 make_table() (método de `difflib.HtmlDiff`), 131
 make_zip_pkg() (en el módulo `test.support.script_helper`), 1628
 make_zip_script() (en el módulo `test.support.script_helper`), 1628
 makedev() (en el módulo `os`), 581
 makedirs() (en el módulo `os`), 580
 makeelement() (método de `xml.etree.ElementTree.Element`), 1155
 makefile() (método de `socket.socket`), 964
 makeLogRecord() (en el módulo `logging`), 679
 makePickle() (método de `logging.handlers.SocketHandler`), 697
 makeRecord() (método de `logging.Logger`), 669
 makeSocket() (método de `logging.handlers.DatagramHandler`), 698
 makeSocket() (método de `logging.handlers.SocketHandler`), 697
 maketrans() (método estático de `bytearray`), 58
 maketrans() (método estático de `bytes`), 58
 maketrans() (método estático de `str`), 48
 mangle_from_ (atributo de `email.policy.Compat32`), 1059
 mangle_from_ (atributo de `email.policy.Policy`), 1055
 map (2to3 fixer), 1610
 map() (función incorporada), 15
 map() (método de `concurrent.futures.Executor`), 830
 map() (método de `multiprocessing.pool.Pool`), 810
 map() (método de `tkinter.ttk.Style`), 1456
 MAP_ADD (opcode), 1848
 map_async() (método de `multiprocessing.pool.Pool`), 810
 map_table_b2() (en el módulo `stringprep`), 146
 map_table_b3() (en el módulo `stringprep`), 146
 map_to_type() (método de `email.headerregistry.HeaderRegistry`), 1065
 mapeado, 1942
 mapLogRecord() (método de `logging.handlers.HTTPHandler`), 702
 mapping
 objeto, 77
 types, operations on, 77
 Mapping (clase en `collections.abc`), 235
 Mapping (clase en `typing`), 1486
 mapping() (método de `msilib.Control`), 1866
 MappingProxyType (clase en `types`), 256
 MappingView (clase en `collections.abc`), 235
 MappingView (clase en `typing`), 1487
 mapPriority() (método de `logging.handlers.SysLogHandler`), 699
 maps (atributo de `collections.ChainMap`), 217
 maps() (en el módulo `nis`), 1897
 máquina virtual, 1947
 marshal (módulo), 441
 marshalling
 objects, 421
 masking
 operations, 32
 Match (clase en `typing`), 1489
 match() (en el módulo `nis`), 1897
 match() (en el módulo `re`), 117
 match() (método de `pathlib.PurePath`), 380
 match() (método de `re.Pattern`), 120
 match_hostname() (en el módulo `ssl`), 977
 match_test() (en el módulo `test.support`), 1617
 match_value() (método de `test.support.Matcher`), 1627
 Matcher (clase en `test.support`), 1626
 matches() (método de `test.support.Matcher`), 1627

- math
 módulo, 32, 298
 math (módulo), 288
 matmul() (en el módulo operator), 366
 max
 función incorporada, 38
 max (atributo de datetime.date), 183
 max (atributo de datetime.datetime), 190
 max (atributo de datetime.time), 198
 max (atributo de datetime.timedelta), 180
 max() (en el módulo audioop), 1349
 max() (función incorporada), 15
 max() (método de decimal.Context), 312
 max() (método de decimal.Decimal), 306
 max_count (atributo de email.headerregistry.BaseHeader), 1062
 MAX_EMAX (en el módulo decimal), 314
 MAX_INTERPOLATION_DEPTH (en el módulo config-parser), 534
 max_line_length (atributo de email.policy.Policy), 1054
 max_lines (atributo de textwrap.TextWrapper), 143
 max_mag() (método de decimal.Context), 313
 max_mag() (método de decimal.Decimal), 306
 max_memuse (en el módulo test.support), 1615
 MAX_PREC (en el módulo decimal), 314
 max_prefixlen (atributo de ipaddress.IPv4Address), 1334
 max_prefixlen (atributo de ipaddress.IPv4Network), 1339
 max_prefixlen (atributo de ipaddress.IPv6Address), 1336
 max_prefixlen (atributo de ipaddress.IPv6Network), 1342
 MAX_Py_ssize_t (en el módulo test.support), 1615
 maxarray (atributo de reprlib.Repr), 265
 maxdeque (atributo de reprlib.Repr), 265
 maxdict (atributo de reprlib.Repr), 265
 maxDiff (atributo de unittest.TestCase), 1535
 maxfrozenset (atributo de reprlib.Repr), 265
 MAXIMUM_SUPPORTED (atributo de ssl.TLSVersion), 985
 maximum_version (atributo de ssl.SSLContext), 995
 maxlen (atributo de collections.deque), 223
 maxlevel (atributo de reprlib.Repr), 265
 maxlist (atributo de reprlib.Repr), 265
 maxlong (atributo de reprlib.Repr), 265
 maxother (atributo de reprlib.Repr), 265
 maxpp() (en el módulo audioop), 1349
 maxset (atributo de reprlib.Repr), 265
 maxsize (atributo de asyncio.Queue), 893
 maxsize (en el módulo sys), 1702
 maxstring (atributo de reprlib.Repr), 265
 maxtuple (atributo de reprlib.Repr), 265
 maxunicode (en el módulo sys), 1702
 MAXYEAR (en el módulo datetime), 178
 MB_ICONASTERISK (en el módulo winsound), 1878
 MB_ICONEXCLAMATION (en el módulo winsound), 1879
 MB_ICONHAND (en el módulo winsound), 1879
 MB_ICONQUESTION (en el módulo winsound), 1879
 MB_OK (en el módulo winsound), 1879
 mbox (clase en mailbox), 1111
 mboxMessage (clase en mailbox), 1117
 mean (atributo de statistics.NormalDist), 339
 mean() (en el módulo statistics), 333
 median (atributo de statistics.NormalDist), 339
 median() (en el módulo statistics), 334
 median_grouped() (en el módulo statistics), 335
 median_high() (en el módulo statistics), 335
 median_low() (en el módulo statistics), 335
 MemberDescriptorType (en el módulo types), 256
 memfd_create() (en el módulo os), 594
 memmove() (en el módulo ctypes), 761
 --memo
 pickletools command line option, 1855
 MemoryBIO (clase en ssl), 1004
 MemoryError, 91
 MemoryHandler (clase en logging.handlers), 701
 memoryview
 objeto, 54
 memoryview (clase incorporada), 68
 memset() (en el módulo ctypes), 761
 merge() (en el módulo heapq), 238
 Message (clase en email.message), 1076
 Message (clase en mailbox), 1115
 message digest, MD5, 543
 message_factory (atributo de email.policy.Policy), 1055
 message_from_binary_file() (en el módulo email), 1049
 message_from_bytes() (en el módulo email), 1049
 message_from_file() (en el módulo email), 1049
 message_from_string() (en el módulo email), 1049
 MessageBeep() (en el módulo winsound), 1877
 MessageClass (atributo de http.server.BaseHTTPRequestHandler), 1304
 MessageError, 1060
 MessageParseError, 1060
 messages (en el módulo xml.parsers.expat.errors), 1196
 meta buscadores de ruta, 1942
 meta() (en el módulo curses), 708
 meta_path (en el módulo sys), 1703
 metaclass, 1942
 metaclass (2to3 fixer), 1610
 MetaPathFinder (clase en importlib.abc), 1792
 metavar (atributo de optparse.Option), 1915
 MetavarTypeHelpFormatter (clase en argparse), 637
 Meter (clase en tkinter.tix), 1460

method
 magic, 1942
 objeto, 84
 special, 1946
method (atributo de `urllib.request.Request`), 1225
METHOD_BLOWFISH (en el módulo `crypt`), 1885
method_calls (atributo de `unittest.mock.Mock`), 1556
METHOD_CRYPT (en el módulo `crypt`), 1885
METHOD_MD5 (en el módulo `crypt`), 1885
METHOD_SHA256 (en el módulo `crypt`), 1885
METHOD_SHA512 (en el módulo `crypt`), 1885
methodattrs (2to3 fixer), 1610
methodcaller() (en el módulo `operator`), 369
MethodDescriptorType (en el módulo `types`), 255
methodHelp() (método de `xmlrpc.client.ServerProxy.system`), 1322
methods
 bytearray, 57
 bytes, 57
 string, 45
methods (atributo de `pycbr.Class`), 1836
methods (en el módulo `crypt`), 1885
methodSignature() (método de `xmlrpc.client.ServerProxy.system`), 1322
MethodType (en el módulo `types`), 255
MethodWrapperType (en el módulo `types`), 255
método, 1942
método especial, 1946
método mágico, 1942
MFD_ALLOW_SEALING (en el módulo `os`), 594
MFD_CLOEXEC (en el módulo `os`), 594
MFD_HUGE_1GB (en el módulo `os`), 594
MFD_HUGE_1MB (en el módulo `os`), 594
MFD_HUGE_2GB (en el módulo `os`), 594
MFD_HUGE_2MB (en el módulo `os`), 594
MFD_HUGE_8MB (en el módulo `os`), 594
MFD_HUGE_16GB (en el módulo `os`), 594
MFD_HUGE_16MB (en el módulo `os`), 594
MFD_HUGE_32MB (en el módulo `os`), 594
MFD_HUGE_64KB (en el módulo `os`), 594
MFD_HUGE_256MB (en el módulo `os`), 594
MFD_HUGE_512KB (en el módulo `os`), 594
MFD_HUGE_512MB (en el módulo `os`), 594
MFD_HUGE_MASK (en el módulo `os`), 594
MFD_HUGE_SHIFT (en el módulo `os`), 594
MFD_HUGETLB (en el módulo `os`), 594
MH (clase en `mailbox`), 1112
MHMessage (clase en `mailbox`), 1118
microsecond (atributo de `datetime.datetime`), 190
microsecond (atributo de `datetime.time`), 198
MIME
 base64 encoding, 1127
 content type, 1124
 headers, 1124, 1202
 quoted-printable encoding, 1133
MIMEApplication (clase en `email.mime.application`), 1085
MIMEAudio (clase en `email.mime.audio`), 1085
MIMEBase (clase en `email.mime.base`), 1084
MIMEImage (clase en `email.mime.image`), 1085
MIMEMessage (clase en `email.mime.message`), 1086
MIMEMultipart (clase en `email.mime.multipart`), 1084
MIMENonMultipart (clase en `email.mime.nonmultipart`), 1084
MIMEPart (clase en `email.message`), 1046
MIMEText (clase en `email.mime.text`), 1086
MimeTypes (clase en `mimetypes`), 1126
mimetypes (módulo), 1124
MIMEVersionHeader (clase en `email.headerregistry`), 1064
min
 función incorporada, 38
min (atributo de `datetime.date`), 183
min (atributo de `datetime.datetime`), 190
min (atributo de `datetime.time`), 198
min (atributo de `datetime.timedelta`), 180
min() (función incorporada), 16
min() (método de `decimal.Context`), 313
min() (método de `decimal.Decimal`), 306
de MIN_EMIN (en el módulo `decimal`), 314
MIN_ETINY (en el módulo `decimal`), 314
min_mag() (método de `decimal.Context`), 313
min_mag() (método de `decimal.Decimal`), 306
MINEQUAL (en el módulo `token`), 1828
MINIMUM_SUPPORTED (atributo de `ssl.TLSVersion`), 985
minimum_version (atributo de `ssl.SSLContext`), 996
minmax() (en el módulo `audioop`), 1349
minor (atributo de `email.headerregistry.MIMEVersionHeader`), 1064
minor() (en el módulo `os`), 581
MINUS (en el módulo `token`), 1827
minus() (método de `decimal.Context`), 313
minute (atributo de `datetime.datetime`), 190
minute (atributo de `datetime.time`), 198
MINYEAR (en el módulo `datetime`), 178
mirrored() (en el módulo `unicodedata`), 144
misc_header (atributo de `cmd.Cmd`), 1420
--missing
 trace command line option, 1660
MISSING (atributo de `contextvars.Token`), 860
MISSING_C_DOCSTRINGS (en el módulo `test.support`), 1616
missing_compiler_executable() (en el módulo `test.support`), 1625
MissingSectionHeaderError, 535
MIXERDEV, 1362
mkd() (método de `ftplib.FTP`), 1261
mkdir() (en el módulo `os`), 580
mkdir() (método de `pathlib.Path`), 385
mkdtemp() (en el módulo `tempfile`), 405
mkfifo() (en el módulo `os`), 581
mknod() (en el módulo `os`), 581
mksalt() (en el módulo `crypt`), 1885
mkstemp() (en el módulo `tempfile`), 405

- `mktemp()` (en el módulo `tempfile`), 407
- `mtime()` (en el módulo `time`), 625
- `mktime_tz()` (en el módulo `email.utils`), 1093
- `mlsd()` (método de `ftplib.FTP`), 1261
- `mmap` (clase en `mmap`), 1031
- `mmap` (módulo), 1031
- `MMDF` (clase en `mailbox`), 1114
- `MMDFMessage` (clase en `mailbox`), 1121
- `Mock` (clase en `unittest.mock`), 1550
- `mock_add_spec()` (método de `unittest.mock.Mock`), 1553
- `mock_calls` (atributo de `unittest.mock.Mock`), 1557
- `mock_open()` (en el módulo `unittest.mock`), 1582
- `mod()` (en el módulo `operator`), 366
- `mode` (atributo de `io.FileIO`), 617
- `mode` (atributo de `ossaudiodev.oss_audio_device`), 1364
- `mode` (atributo de `statistics.NormalDist`), 339
- `mode` (atributo de `tarfile.TarInfo`), 502
- `mode()` (en el módulo `statistics`), 336
- `mode()` (en el módulo `turtle`), 1410
- `modes`
 - `file`, 17
- `modf()` (en el módulo `math`), 290
- `modified()` (método de `urllib.robotparser.RobotFileParser`), 1247
- `Modify()` (método de `msilib.View`), 1863
- `modify()` (método de `select.devpoll`), 1009
- `modify()` (método de `select.epoll`), 1010
- `modify()` (método de `selectors.BaseSelector`), 1015
- `modify()` (método de `select.poll`), 1011
- `module`
 - `search_path`, 410, 1703, 1773
- `module` (atributo de `pyclbr.Class`), 1836
- `module` (atributo de `pyclbr.Function`), 1835
- `module_for_loader()` (en el módulo `importlib.util`), 1805
- `module_from_spec()` (en el módulo `importlib.util`), 1805
- `module_repr()` (método de `importlib.abc.Loader`), 1794
- `ModuleFinder` (clase en `modulefinder`), 1786
- `modulefinder` (módulo), 1786
- `ModuleInfo` (clase en `pkgutil`), 1783
- `ModuleNotFoundError`, 91
- `modules` (atributo de `modulefinder.ModuleFinder`), 1786
- `modules` (en el módulo `sys`), 1703
- `modules_cleanup()` (en el módulo `test.support`), 1623
- `modules_setup()` (en el módulo `test.support`), 1623
- `ModuleSpec` (clase en `importlib.machinery`), 1802
- `ModuleType` (clase en `types`), 255
- módulo, 1942
 - `__main__`, 1787, 1788
 - `_locale`, 1376
 - `array`, 54
 - `base64`, 1130
 - `bdb`, 1640
 - `binhex`, 1130
 - `cmd`, 1640
 - `copy`, 437
 - `crypt`, 1882
 - `dbm.gnu`, 439
 - `dbm.ndbm`, 439
 - `errno`, 92
 - `glob`, 409
 - `imp`, 25
 - `math`, 32, 298
 - `os`, 1881
 - `pickle`, 259, 437, 438, 441
 - `pty`, 570
 - `pwd`, 391
 - `pyexpat`, 1189
 - `re`, 45, 409
 - `shelve`, 441
 - `signal`, 864
 - `sitecustomize`, 1774
 - `socket`, 1199
 - `stat`, 586
 - `string`, 1381
 - `struct`, 968
 - `sys`, 19
 - `types`, 84
 - `urllib.request`, 1250
 - `usercustomize`, 1774
 - `uu`, 1130
- módulo de extensión, 1939
- `monotonic()` (en el módulo `time`), 625
- `monotonic_ns()` (en el módulo `time`), 625
- `month` (atributo de `datetime.date`), 184
- `month` (atributo de `datetime.datetime`), 190
- `month()` (en el módulo `calendar`), 216
- `month_abbr` (en el módulo `calendar`), 216
- `month_name` (en el módulo `calendar`), 216
- `monthcalendar()` (en el módulo `calendar`), 216
- `monthdatescalendar()` (método de `calendar.Calendar`), 213
- `monthdays2calendar()` (método de `calendar.Calendar`), 213
- `monthdayscalendar()` (método de `calendar.Calendar`), 213
- `montrange()` (en el módulo `calendar`), 216
- `Morsel` (clase en `http.cookies`), 1309
- `most_common()` (método de `collections.Counter`), 220
- `mouseinterval()` (en el módulo `curses`), 708
- `mousemask()` (en el módulo `curses`), 708
- `move()` (en el módulo `shutil`), 414
- `move()` (método de `curses.panel.Panel`), 726
- `move()` (método de `curses.window`), 716
- `move()` (método de `mmap.mmap`), 1033
- `move()` (método de `tkinter.ttk.Treeview`), 1454
- `move_to_end()` (método de `collections.OrderedDict`), 231
- `MozillaCookieJar` (clase en `http.cookiejar`), 1315
- `MRO`, 1943
- `mro()` (método de `class`), 86

- msg (atributo de *http.client.HTTPResponse*), 1255
 msg (atributo de *json.JSONDecodeError*), 1102
 msg (atributo de *re.error*), 120
 msg (atributo de *traceback.TracebackException*), 1749
 msg() (método de *telnetlib.Telnet*), 1289
 msi, 1861
 msilib (módulo), 1861
 msvcrt (módulo), 1867
 mt_interact() (método de *telnetlib.Telnet*), 1290
 mtime (atributo de *gzip.GzipFile*), 474
 mtime (atributo de *tarfile.TarInfo*), 501
 mtime() (método de *urllib.robotparser.RobotFileParser*), 1247
 mul() (en el módulo *audioop*), 1349
 mul() (en el módulo *operator*), 366
 MultiCall (clase en *xmlrpc.client*), 1325
 MULTILINE (en el módulo *re*), 116
 MultiLoopChildWatcher (clase en *asyncio*), 937
 multimode() (en el módulo *statistics*), 336
 MultipartConversionError, 1060
 multiply() (método de *decimal.Context*), 313
 multiprocessing (módulo), 782
 multiprocessing.connection (módulo), 812
 multiprocessing.dummy (módulo), 816
 multiprocessing.Manager() (en el módulo *multiprocessing.sharedctypes*), 802
 multiprocessing.managers (módulo), 803
 multiprocessing.pool (módulo), 809
 multiprocessing.shared_memory (módulo), 825
 multiprocessing.sharedctypes (módulo), 800
 mutable, 1943
 sequence types, 40
 MutableMapping (clase en *collections.abc*), 235
 MutableMapping (clase en *typing*), 1486
 MutableSequence (clase en *collections.abc*), 235
 MutableSequence (clase en *typing*), 1486
 MutableSet (clase en *collections.abc*), 235
 MutableSet (clase en *typing*), 1486
 mvderwin() (método de *curses.window*), 716
 mvwin() (método de *curses.window*), 716
 myrights() (método de *imaplib.IMAP4*), 1268
- ## N
- n N
 timeit command line option, 1657
 N_TOKENS (en el módulo *token*), 1829
 n_waiting (atributo de *threading.Barrier*), 781
 name (atributo de *codecs.CodecInfo*), 159
 name (atributo de *contextvars.ContextVar*), 859
 name (atributo de *doctest.DocTest*), 1512
 name (atributo de *email.headerregistry.BaseHeader*), 1062
 name (atributo de *hashlib.hash*), 545
 name (atributo de *hmac.HMAC*), 554
 name (atributo de *http.cookiejar.Cookie*), 1318
 name (atributo de *importlib.abc.FileLoader*), 1796
 name (atributo de *importlib.machinery.ExtensionFileLoader*), 1802
 name (atributo de *importlib.machinery.ModuleSpec*), 1803
 name (atributo de *importlib.machinery.SourceFileLoader*), 1801
 name (atributo de *importlib.machinery.SourcelessFileLoader*), 1802
 name (atributo de *inspect.Parameter*), 1764
 name (atributo de *io.FileIO*), 617
 name (atributo de *multiprocessing.Process*), 789
 name (atributo de *multiprocessing.shared_memory.SharedMemory*), 826
 name (atributo de *os.DirEntry*), 585
 name (atributo de *ossaudiodev.oss_audio_device*), 1364
 name (atributo de *pyclbr.Class*), 1836
 name (atributo de *pyclbr.Function*), 1835
 name (atributo de *tarfile.TarInfo*), 501
 name (atributo de *threading.Thread*), 773
 name (atributo de *xml.dom.Attr*), 1168
 name (atributo de *xml.dom.DocumentType*), 1165
 name (atributo de *zipfile.Path*), 490
 name (en el módulo *os*), 559
 NAME (en el módulo *token*), 1827
 name() (en el módulo *unicodedata*), 144
 name2codepoint (en el módulo *html.entities*), 1140
 Named Shared Memory, 825
 NamedTemporaryFile() (en el módulo *tempfile*), 404
 NamedTuple (clase en *typing*), 1489
 namedtuple() (en el módulo *collections*), 227
 NameError, 91
 namelist() (método de *zipfile.ZipFile*), 487
 nameprep() (en el módulo *encodings.idna*), 175
 namer (atributo de *logging.handlers.BaseRotatingHandler*), 694
 namereplace_errors() (en el módulo *codecs*), 163
 Namespace (clase en *argparse*), 654
 Namespace (clase en *multiprocessing.managers*), 805
 namespace() (método de *imaplib.IMAP4*), 1268
 Namespace() (método de *multiprocessing.managers.SyncManager*), 804
 NAMESPACE_DNS (en el módulo *uuid*), 1293
 NAMESPACE_OID (en el módulo *uuid*), 1293
 NAMESPACE_URL (en el módulo *uuid*), 1293
 NAMESPACE_X500 (en el módulo *uuid*), 1293
 NamespaceErr, 1169
 namespaceURI (atributo de *xml.dom.Node*), 1164
 NaN, 11
 nan (en el módulo *cmath*), 297
 nan (en el módulo *math*), 294
 nanj (en el módulo *cmath*), 297
 NannyNag, 1834
 napms() (en el módulo *curses*), 709
 nargs (atributo de *optparse.Option*), 1914

- native_id (atributo de *threading.Thread*), 773
 nbytes (atributo de *memoryview*), 73
 ncurses_version (en el módulo *curses*), 718
 ndiff() (en el módulo *difflib*), 132
 ndim (atributo de *memoryview*), 74
 ne (2to3 fixer), 1610
 ne() (en el módulo *operator*), 365
 needs_input (atributo de *bz2.BZ2Decompressor*), 478
 needs_input (atributo de *lzma.LZMADecompressor*), 483
 neg() (en el módulo *operator*), 367
 netmask (atributo de *ipaddress.IPv4Network*), 1339
 netmask (atributo de *ipaddress.IPv6Network*), 1342
 NetmaskValueError, 1346
 netrc (clase en *netrc*), 535
 netrc (módulo), 535
 NetrcParseError, 535
 netscape (atributo de *http.cookiejar.CookiePolicy*), 1316
 network (atributo de *ipaddress.IPv4Interface*), 1344
 network (atributo de *ipaddress.IPv6Interface*), 1344
 Network News Transfer Protocol, 1271
 network_address (atributo de *ipaddress.IPv4Network*), 1339
 network_address (atributo de *ipaddress.IPv6Network*), 1342
 new() (en el módulo *hashlib*), 544
 new() (en el módulo *hmac*), 553
 new_alignment() (método de *formatter.writer*), 1859
 new_child() (método de *collections.ChainMap*), 218
 new_class() (en el módulo *types*), 253
 new_event_loop() (en el módulo *asyncio*), 897
 new_event_loop() (método de *asyncio.AbstractEventLoopPolicy*), 935
 new_font() (método de *formatter.writer*), 1859
 new_margin() (método de *formatter.writer*), 1859
 new_module() (en el módulo *imp*), 1929
 new_panel() (en el módulo *curses.panel*), 726
 new_spacing() (método de *formatter.writer*), 1860
 new_styles() (método de *formatter.writer*), 1860
 newgroups() (método de *nntplib.NNTP*), 1274
 NEWLINE (en el módulo *token*), 1827
 newlines (atributo de *io.TextIOBase*), 619
 newnews() (método de *nntplib.NNTP*), 1275
 newpad() (en el módulo *curses*), 709
 NewType() (en el módulo *typing*), 1491
 newwin() (en el módulo *curses*), 709
 next (2to3 fixer), 1610
 next (pdb command), 1644
 next() (función incorporada), 16
 next() (método de *nntplib.NNTP*), 1276
 next() (método de *tarfile.TarFile*), 499
 next() (método de *tkinter.tk.Treeview*), 1454
 next_minus() (método de *decimal.Context*), 313
 next_minus() (método de *decimal.Decimal*), 306
 next_plus() (método de *decimal.Context*), 313
 next_plus() (método de *decimal.Decimal*), 306
 next_toward() (método de *decimal.Context*), 313
 next_toward() (método de *decimal.Decimal*), 306
 nextfile() (en el módulo *fileinput*), 395
 nextkey() (método de *dbm.gnu.gdbm*), 444
 nextSibling (atributo de *xml.dom.Node*), 1164
 ngettext() (en el módulo *gettext*), 1368
 ngettext() (método de *gettext.GNUTranslations*), 1372
 ngettext() (método de *gettext.NullTranslations*), 1370
 nice() (en el módulo *os*), 599
 nis (módulo), 1897
 NL (en el módulo *token*), 1829
 nl() (en el módulo *curses*), 709
 nl_langinfo() (en el módulo *locale*), 1378
 nlargest() (en el módulo *heapq*), 238
 nlst() (método de *ftplib.FTP*), 1261
 NNTP
 protocol, 1271
 NNTP (clase en *nntplib*), 1272
 nntp_implementation (atributo de *nntplib.NNTP*), 1273
 NNTP_SSL (clase en *nntplib*), 1272
 nntp_version (atributo de *nntplib.NNTP*), 1273
 NNTPDataError, 1273
 NNTPError, 1273
 nntplib (módulo), 1271
 NNTPPermanentError, 1273
 NNTPProtocolError, 1273
 NNTPReplyError, 1273
 NNTPTemporaryError, 1273
 no_proxy, 1223
 no_tracing() (en el módulo *test.support*), 1621
 no_type_check() (en el módulo *typing*), 1492
 no_type_check_decorator() (en el módulo *typing*), 1492
 nocbreak() (en el módulo *curses*), 709
 NoDataAllowedErr, 1169
 node() (en el módulo *platform*), 727
 nodelay() (método de *curses.window*), 716
 nodeName (atributo de *xml.dom.Node*), 1164
 NodeTransformer (clase en *ast*), 1823
 nodeType (atributo de *xml.dom.Node*), 1163
 nodeValue (atributo de *xml.dom.Node*), 1164
 NodeVisitor (clase en *ast*), 1822
 noecho() (en el módulo *curses*), 709
 NOEXPR (en el módulo *locale*), 1379
 nombre calificado, 1945
 NoModificationAllowedErr, 1170
 nonblock() (método de *ossaudio-dev.oss_audio_device*), 1363
 NonCallableMagicMock (clase en *unittest.mock*), 1577
 NonCallableMock (clase en *unittest.mock*), 1557
 None (Built-in object), 29
 None (variable incorporada), 27
 nonl() (en el módulo *curses*), 709

- ul style="list-style-type: none; padding-left: 0;">
- nonzero (*2to3 fixer*), 1610
- noop() (*método de imaplib.IMAP4*), 1268
- noop() (*método de poplib.POP3*), 1264
- NoOptionError, 535
- NOP (*opcode*), 1845
- noqiflush() (*en el módulo curses*), 709
- noraw() (*en el módulo curses*), 709
- no-report
 - trace command line option, 1660
- NoReturn (*en el módulo typing*), 1493
- NORMAL_PRIORITY_CLASS (*en el módulo subprocess*), 847
- NormalDist (*clase en statistics*), 339
- normalize() (*en el módulo locale*), 1380
- normalize() (*en el módulo unicodedata*), 144
- normalize() (*método de decimal.Context*), 313
- normalize() (*método de decimal.Decimal*), 307
- normalize() (*método de xml.dom.Node*), 1165
- NORMALIZE_WHITESPACE (*en el módulo doctest*), 1504
- normalvariate() (*en el módulo random*), 329
- normcase() (*en el módulo os.path*), 392
- normpath() (*en el módulo os.path*), 392
- NoSectionError, 534
- NoSuchMailboxError, 1122
- not
 - operador, 30
- not in
 - operador, 30, 38
- not_() (*en el módulo operator*), 366
- NotADirectoryError, 96
- notationDecl() (*método de xml.sax.handler.DTDHandler*), 1183
- NotationDeclHandler() (*método de xml.parsers.expat.xmlparser*), 1193
- notations (*atributo de xml.dom.DocumentType*), 1166
- NotConnected, 1252
- NoteBook (*clase en tkinter.tix*), 1461
- Notebook (*clase en tkinter.ttk*), 1447
- NotEmptyError, 1122
- NOTEQUAL (*en el módulo token*), 1828
- NotFoundErr, 1169
- notify() (*método de asyncio.Condition*), 887
- notify() (*método de threading.Condition*), 777
- notify_all() (*método de asyncio.Condition*), 887
- notify_all() (*método de threading.Condition*), 777
- notimeout() (*método de curses.window*), 716
- NotImplemented (*variable incorporada*), 27
- NotImplementedError, 91
- NotStandaloneHandler() (*método de xml.parsers.expat.xmlparser*), 1194
- NotSupportedErr, 1169
- NotSupportedError, 460
- noutrefresh() (*método de curses.window*), 716
- now() (*método de clase de datetime.datetime*), 187
- npgettext() (*en el módulo gettext*), 1368
- npgettext() (*método de gettext.GNUTranslations*), 1372
- npgettext() (*método de gettext.NullTranslations*), 1370
- NSIG (*en el módulo signal*), 1026
- nsmallest() (*en el módulo heapq*), 238
- NT_OFFSET (*en el módulo token*), 1829
- NTEventLogHandler (*clase en logging.handlers*), 700
- ntohl() (*en el módulo socket*), 960
- ntohs() (*en el módulo socket*), 960
- ntransfercmd() (*método de ftplib.FTP*), 1261
- nullcontext() (*en el módulo contextlib*), 1731
- NullFormatter (*clase en formatter*), 1859
- NullHandler (*clase en logging*), 693
- NullImporter (*clase en imp*), 1932
- NullTranslations (*clase en gettext*), 1370
- NullWriter (*clase en formatter*), 1860
- num_addresses (*atributo de ipaddress.IPv4Network*), 1339
- num_addresses (*atributo de ipaddress.IPv6Network*), 1342
- num_tickets (*atributo de ssl.SSLContext*), 996
- Number (*clase en numbers*), 285
- NUMBER (*en el módulo token*), 1827
- number=N
 - timeit command line option, 1657
- number_class() (*método de decimal.Context*), 313
- number_class() (*método de decimal.Decimal*), 307
- numbers (*módulo*), 285
- numerator (*atributo de fractions.Fraction*), 325
- numerator (*atributo de numbers.Rational*), 286
- numeric
 - conversions, 32
 - literals, 31
 - object, 30
 - objeto, 31
 - types, operations on, 31
- numeric() (*en el módulo unicodedata*), 144
- Numerical Python, 22
- número complejo, 1937
- numinput() (*en el módulo turtle*), 1410
- numliterals (*2to3 fixer*), 1610
- ## O
- o
 - pickletools command line option, 1855
 - o <output>
 - zipapp command line option, 1684
 - O_APPEND (*en el módulo os*), 569
 - O_ASYNC (*en el módulo os*), 570
 - O_BINARY (*en el módulo os*), 570
 - O_CLOEXEC (*en el módulo os*), 569
 - O_CREAT (*en el módulo os*), 569
 - O_DIRECT (*en el módulo os*), 570
 - O_DIRECTORY (*en el módulo os*), 570
 - O_DSYNC (*en el módulo os*), 569

- `O_EXCL` (en el módulo `os`), 569
- `O_EXLOCK` (en el módulo `os`), 570
- `O_NDELAY` (en el módulo `os`), 569
- `O_NOATIME` (en el módulo `os`), 570
- `O_NOCTTY` (en el módulo `os`), 569
- `O_NOFOLLOW` (en el módulo `os`), 570
- `O_NOINHERIT` (en el módulo `os`), 570
- `O_NONBLOCK` (en el módulo `os`), 569
- `O_PATH` (en el módulo `os`), 570
- `O_RANDOM` (en el módulo `os`), 570
- `O_RDONLY` (en el módulo `os`), 569
- `O_RDWR` (en el módulo `os`), 569
- `O_RSYNC` (en el módulo `os`), 569
- `O_SEQUENTIAL` (en el módulo `os`), 570
- `O_SHLOCK` (en el módulo `os`), 570
- `O_SHORT_LIVED` (en el módulo `os`), 570
- `O_SYNC` (en el módulo `os`), 569
- `O_TEMPORARY` (en el módulo `os`), 570
- `O_TEXT` (en el módulo `os`), 570
- `O_TMPFILE` (en el módulo `os`), 570
- `O_TRUNC` (en el módulo `os`), 569
- `O_WRONLY` (en el módulo `os`), 569
- `obj` (atributo de `memoryview`), 73
- `object`
 - `code`, 84, 441
 - `numeric`, 30
- `object` (atributo de `UnicodeError`), 94
- `object` (clase incorporada), 16
- `objects`
 - `comparing`, 30
 - `flattening`, 421
 - `marshalling`, 421
 - `persistent`, 421
 - `pickling`, 421
 - `serializing`, 421
- objeto, 1943
 - `Boolean`, 31
 - `bytearray`, 40, 54, 56
 - `bytes`, 54
 - `complex number`, 31
 - `dictionary`, 77
 - `floating point`, 31
 - `integer`, 31
 - `io.StringIO`, 44
 - `list`, 40, 41
 - `mapping`, 77
 - `memoryview`, 54
 - `method`, 84
 - `numeric`, 31
 - `range`, 42
 - `sequence`, 38
 - `set`, 75
 - `socket`, 950
 - `string`, 44
 - `traceback`, 1695, 1747
 - `tuple`, 39, 42
 - `type`, 24
- objeto archivo, 1939
- objeto tipo ruta, 1944
- objetos tipo archivo, 1939
- objetos tipo binarios, 1937
- `obufcount()` (método de `ossaudio-dev.oss_audio_device`), 1364
- `obuffree()` (método de `ossaudio-dev.oss_audio_device`), 1364
- `oct()` (función incorporada), 16
- `octal`
 - `literals`, 31
- `octdigits` (en el módulo `string`), 99
- `offset` (atributo de `SyntaxError`), 93
- `offset` (atributo de `traceback.TracebackException`), 1749
- `offset` (atributo de `xml.parsers.expat.ExpatError`), 1194
- `OK` (en el módulo `curses`), 718
- `old_value` (atributo de `contextvars.Token`), 859
- `OleDLL` (clase en `ctypes`), 754
- `onclick()` (en el módulo `turtle`), 1408
- `ondrag()` (en el módulo `turtle`), 1404
- `onecmd()` (método de `cmd.Cmd`), 1419
- `onkey()` (en el módulo `turtle`), 1408
- `onkeypress()` (en el módulo `turtle`), 1408
- `onkeyrelease()` (en el módulo `turtle`), 1408
- `onrelease()` (en el módulo `turtle`), 1403
- `onscreenclick()` (en el módulo `turtle`), 1408
- `ontimer()` (en el módulo `turtle`), 1409
- `OP` (en el módulo `token`), 1829
- `OP_ALL` (en el módulo `ssl`), 981
- `OP_CIPHER_SERVER_PREFERENCE` (en el módulo `ssl`), 982
- `OP_ENABLE_MIDDLEBOX_COMPAT` (en el módulo `ssl`), 982
- `OP_IGNORE_UNEXPECTED_EOF` (en el módulo `ssl`), 983
- `OP_NO_COMPRESSION` (en el módulo `ssl`), 983
- `OP_NO_RENEGOTIATION` (en el módulo `ssl`), 982
- `OP_NO_SSLv2` (en el módulo `ssl`), 981
- `OP_NO_SSLv3` (en el módulo `ssl`), 982
- `OP_NO_TICKET` (en el módulo `ssl`), 983
- `OP_NO_TLSv1` (en el módulo `ssl`), 982
- `OP_NO_TLSv1_1` (en el módulo `ssl`), 982
- `OP_NO_TLSv1_2` (en el módulo `ssl`), 982
- `OP_NO_TLSv1_3` (en el módulo `ssl`), 982
- `OP_SINGLE_DH_USE` (en el módulo `ssl`), 982
- `OP_SINGLE_ECDH_USE` (en el módulo `ssl`), 982
- `open()` (en el módulo `aifc`), 1350
- `open()` (en el módulo `bz2`), 476
- `open()` (en el módulo `codecs`), 160
- `open()` (en el módulo `dbm`), 442
- `open()` (en el módulo `dbm.dumb`), 445
- `open()` (en el módulo `dbm.gnu`), 444
- `open()` (en el módulo `dbm.ndbm`), 445
- `open()` (en el módulo `gzip`), 473
- `open()` (en el módulo `io`), 611
- `open()` (en el módulo `lzma`), 480
- `open()` (en el módulo `os`), 569

- `open()` (en el módulo `ossaudiodev`), 1361
- `open()` (en el módulo `shelve`), 438
- `open()` (en el módulo `sunau`), 1353
- `open()` (en el módulo `tarfile`), 495
- `open()` (en el módulo `tokenize`), 1831
- `open()` (en el módulo `wave`), 1355
- `open()` (en el módulo `webbrowser`), 1200
- `open()` (función incorporada), 16
- `open()` (método de clase de `tarfile.TarFile`), 498
- `open()` (método de `imaplib.IMAP4`), 1268
- `open()` (método de `pathlib.Path`), 386
- `open()` (método de `pipes.Template`), 1892
- `open()` (método de `telnetlib.Telnet`), 1289
- `open()` (método de `urllib.request.OpenerDirector`), 1226
- `open()` (método de `urllib.request.URLOpener`), 1236
- `open()` (método de `webbrowser.controller`), 1202
- `open()` (método de `zipfile.Path`), 490
- `open()` (método de `zipfile.ZipFile`), 487
- `open_binary()` (en el módulo `importlib.resources`), 1798
- `open_code()` (en el módulo `io`), 611
- `open_connection()` (en el módulo `asyncio`), 879
- `open_new()` (en el módulo `webbrowser`), 1200
- `open_new()` (método de `webbrowser.controller`), 1202
- `open_new_tab()` (en el módulo `webbrowser`), 1200
- `open_new_tab()` (método de `webbrowser.controller`), 1202
- `open_osfhandle()` (en el módulo `msvcrt`), 1867
- `open_resource()` (método de `importlib.abc.ResourceReader`), 1795
- `open_text()` (en el módulo `importlib.resources`), 1798
- `open_unix_connection()` (en el módulo `asyncio`), 880
- `open_unknown()` (método de `urllib.request.URLOpener`), 1236
- `open_urlresource()` (en el módulo `test.support`), 1622
- `OpenDatabase()` (en el módulo `msilib`), 1861
- `OpenerDirector` (clase en `urllib.request`), 1222
- `openfp()` (en el módulo `sunau`), 1353
- `openfp()` (en el módulo `wave`), 1356
- `OpenKey()` (en el módulo `winreg`), 1872
- `OpenKeyEx()` (en el módulo `winreg`), 1872
- `openlog()` (en el módulo `syslog`), 1898
- `openmixer()` (en el módulo `ossaudiodev`), 1362
- `openpty()` (en el módulo `os`), 570
- `openpty()` (en el módulo `pty`), 1888
- `OpenSSL`
 - (use in module `hashlib`), 544
 - (use in module `ssl`), 973
- `OPENSSL_VERSION` (en el módulo `ssl`), 984
- `OPENSSL_VERSION_INFO` (en el módulo `ssl`), 984
- `OPENSSL_VERSION_NUMBER` (en el módulo `ssl`), 984
- `OpenView()` (método de `msilib.Database`), 1862
- operador
 - `%` (percent), 31
 - `&` (ampersand), 32
 - `*` (asterisk), 31
 - `**`, 31
 - `/` (slash), 31
 - `//`, 31
 - `<` (less), 30
 - `<<`, 32
 - `<=`, 30
 - `!=`, 30
 - `==`, 30
 - `>` (greater), 30
 - `>=`, 30
 - `>>`, 32
 - `^` (caret), 32
 - `|` (vertical bar), 32
 - `~` (tilde), 32
- and, 29, 30
- in, 30, 38
- is, 30
- is not, 30
- not, 30
- not in, 30, 38
- or, 29, 30
- operation
 - concatenation, 38
 - repetition, 38
 - slice, 38
 - subscript, 38
- `OperationalError`, 460
- operations
 - bitwise, 32
 - Boolean, 29, 30
 - masking, 32
 - shifting, 32
- operations on
 - dictionary type, 77
 - integer types, 32
 - list type, 40
 - mapping types, 77
 - numeric types, 31
 - sequence types, 38, 40
- operator
 - `-` (minus), 31
 - `+` (plus), 31
 - comparison, 30
- operator (2to3 fixer), 1610
- operator (módulo), 365
- `opmap` (en el módulo `dis`), 1854
- `opname` (en el módulo `dis`), 1854
- `optim_args_from_interpreter_flags()` (en el módulo `test.support`), 1619
- `optimize()` (en el módulo `pickletools`), 1856
- `OPTIMIZED_BYTECODE_SUFFIXES` (en el módulo `importlib.machinery`), 1799
- Optional (en el módulo `typing`), 1494
- `OptionGroup` (clase en `optparse`), 1908
- `OptionMenu` (clase en `tkinter.tix`), 1460
- `OptionParser` (clase en `optparse`), 1911
- options (atributo de `doctest.Example`), 1513

- options (atributo de `ssl.SSLContext`), 996
 - Options (clase en `ssl`), 983
 - options() (método de `configparser.ConfigParser`), 531
 - optionxform() (método de `configparser.ConfigParser`), 533
 - optparse (módulo), 1901
 - or
 - operador, 29, 30
 - or_() (en el módulo `operator`), 367
 - ord() (función incorporada), 19
 - orden de resolución de métodos, 1942
 - ordered_attributes (atributo de `xml.parsers.expat.xmlparser`), 1191
 - OrderedDict (clase en `collections`), 231
 - OrderedDict (clase en `typing`), 1488
 - origin (atributo de `importlib.machinery.ModuleSpec`), 1803
 - origin_req_host (atributo de `urllib.request.Request`), 1225
 - origin_server (atributo de `wsgiref.handlers.BaseHandler`), 1218
 - os
 - módulo, 1881
 - os (módulo), 559
 - os_environ (atributo de `wsgiref.handlers.BaseHandler`), 1216
 - OSError, 91
 - os.path (módulo), 389
 - ossaudiodev (módulo), 1361
 - OSSAudioError, 1361
 - outfile
 - json.tool command line option, 1105
 - output (atributo de `subprocess.CalledProcessError`), 838
 - output (atributo de `subprocess.TimeoutExpired`), 838
 - output (atributo de `unittest.TestCase`), 1532
 - output() (método de `http.cookies.BaseCookie`), 1309
 - output() (método de `http.cookies.Morsel`), 1310
 - output=<file>
 - pickletools command line option, 1855
 - output=<output>
 - zipapp command line option, 1684
 - output_charset (atributo de `email.charset.Charset`), 1090
 - output_charset() (método de `gettext.NullTranslations`), 1371
 - output_codec (atributo de `email.charset.Charset`), 1090
 - output_difference() (método de `doctest.OutputChecker`), 1515
 - OutputChecker (clase en `doctest`), 1515
 - OutputString() (método de `http.cookies.Morsel`), 1310
 - OutsideDestinationError, 497
 - over() (método de `nnplib.NNTP`), 1275
 - Overflow (clase en `decimal`), 316
 - OverflowError, 92
 - overlap() (método de `statistics.NormalDist`), 340
 - overlaps() (método de `ipaddress.IPv4Network`), 1340
 - overlaps() (método de `ipaddress.IPv6Network`), 1342
 - overlay() (método de `curses.window`), 716
 - overload() (en el módulo `typing`), 1492
 - overwrite() (método de `curses.window`), 716
 - owner() (método de `pathlib.Path`), 386
- ## P
- p
 - pickletools command line option, 1855
 - timeit command line option, 1657
 - unittest-discover command line option, 1522
 - p (pdb command), 1645
 - p <interpreter>
 - zipapp command line option, 1684
 - P_ALL (en el módulo `os`), 604
 - P_DETACH (en el módulo `os`), 602
 - P_NOWAIT (en el módulo `os`), 602
 - P_NOWAITO (en el módulo `os`), 602
 - P_OVERLAY (en el módulo `os`), 602
 - P_PGID (en el módulo `os`), 604
 - P_PID (en el módulo `os`), 604
 - P_WAIT (en el módulo `os`), 602
 - pack() (en el módulo `struct`), 154
 - pack() (método de `mailbox.MH`), 1112
 - pack() (método de `struct.Struct`), 158
 - pack_array() (método de `xdrlib.Packer`), 537
 - pack_bytes() (método de `xdrlib.Packer`), 537
 - pack_double() (método de `xdrlib.Packer`), 537
 - pack_farray() (método de `xdrlib.Packer`), 537
 - pack_float() (método de `xdrlib.Packer`), 537
 - pack_fopaque() (método de `xdrlib.Packer`), 537
 - pack_fstring() (método de `xdrlib.Packer`), 537
 - pack_into() (en el módulo `struct`), 154
 - pack_into() (método de `struct.Struct`), 158
 - pack_list() (método de `xdrlib.Packer`), 537
 - pack_opaque() (método de `xdrlib.Packer`), 537
 - pack_string() (método de `xdrlib.Packer`), 537
 - package, 1774
 - Package (en el módulo `importlib.resources`), 1798
 - packed (atributo de `ipaddress.IPv4Address`), 1335
 - packed (atributo de `ipaddress.IPv6Address`), 1336
 - Packer (clase en `xdrlib`), 536
 - packing
 - binary data, 153
 - packing (widgets), 1435
 - PAGER, 1496
 - pair_content() (en el módulo `curses`), 709
 - pair_number() (en el módulo `curses`), 709
 - PanedWindow (clase en `tkinter.tix`), 1461
 - paquete, 1943
 - paquete de espacios de nombres, 1943

- paquete provisorio, 1945
- paquete regular, 1946
- Parameter (clase en inspect), 1764
- ParameterizedMIMEHeader (clase en email.headerregistry), 1064
- parameters (atributo de inspect.Signature), 1763
- parámetro, 1943
- params (atributo de email.headerregistry.ParameterizedMIMEHeader), 1064
- pardir (en el módulo os), 608
- paren (2to3 fixer), 1610
- parent (atributo de importlib.machinery.ModuleSpec), 1803
- parent (atributo de pycbr.Class), 1836
- parent (atributo de pycbr.Function), 1835
- parent (atributo de urllib.request.BaseHandler), 1227
- parent () (método de tkinter.ttk.Treeview), 1454
- parent_process () (en el módulo multiprocessing), 794
- parentNode (atributo de xml.dom.Node), 1163
- parents (atributo de collections.ChainMap), 218
- paretovariate () (en el módulo random), 329
- parse () (en el módulo ast), 1821
- parse () (en el módulo cgi), 1205
- parse () (en el módulo xml.dom.minidom), 1171
- parse () (en el módulo xml.dom.pulldom), 1176
- parse () (en el módulo xml.etree.ElementTree), 1151
- parse () (en el módulo xml.sax), 1178
- parse () (método de doctest.DocTestParser), 1514
- parse () (método de email.parser.BytesParser), 1048
- parse () (método de email.parser.Parser), 1048
- parse () (método de string.Formatter), 100
- parse () (método de urllib.robotparser.RobotFileParser), 1247
- parse () (método de xml.etree.ElementTree.ElementTree), 1156
- Parse () (método de xml.parsers.expat.xmlparser), 1190
- parse () (método de xml.sax.xmlreader.XMLReader), 1186
- parse_and_bind () (en el módulo readline), 147
- parse_args () (método de argparse.ArgumentParser), 651
- PARSE_COLNAMES (en el módulo sqlite3), 448
- parse_config_h () (en el módulo sysconfig), 1713
- PARSE_DECLTYPES (en el módulo sqlite3), 448
- parse_header () (en el módulo cgi), 1206
- parse_headers () (en el módulo http.client), 1251
- parse_intermixed_args () (método de argparse.ArgumentParser), 661
- parse_known_args () (método de argparse.ArgumentParser), 660
- parse_known_intermixed_args () (método de argparse.ArgumentParser), 661
- parse_multipart () (en el módulo cgi), 1205
- parse_qs () (en el módulo urllib.parse), 1240
- parse_qs1 () (en el módulo urllib.parse), 1240
- parseaddr () (en el módulo email.utils), 1092
- parsebytes () (método de email.parser.BytesParser), 1048
- parsedate () (en el módulo email.utils), 1093
- parsedate_to_datetime () (en el módulo email.utils), 1093
- parsedate_tz () (en el módulo email.utils), 1093
- ParseError (clase en xml.etree.ElementTree), 1161
- ParseFile () (método de xml.parsers.expat.xmlparser), 1190
- ParseFlags () (en el módulo imaplib), 1266
- Parser (clase en email.parser), 1048
- parser (módulo), 1813
- ParserCreate () (en el módulo xml.parsers.expat), 1189
- ParserError, 1816
- ParseResult (clase en urllib.parse), 1244
- ParseResultBytes (clase en urllib.parse), 1244
- parsestr () (método de email.parser.Parser), 1048
- parseString () (en el módulo xml.dom.minidom), 1172
- parseString () (en el módulo xml.dom.pulldom), 1177
- parseString () (en el módulo xml.sax), 1178
- parsing
 - Python source code, 1813
 - URL, 1238
- ParsingError, 535
- partial (atributo de asyncio.IncompleteReadError), 896
- partial () (en el módulo functools), 360
- partial () (método de imaplib.IMAP4), 1268
- partialmethod (clase en functools), 360
- parties (atributo de threading.Barrier), 781
- partition () (método de bytearray), 58
- partition () (método de bytes), 58
- partition () (método de str), 49
- pass_ () (método de poplib.POP3), 1263
- Paste, 1467
- patch () (en el módulo test.support), 1625
- patch () (en el módulo unittest.mock), 1566
- patch.dict () (en el módulo unittest.mock), 1570
- patch.multiple () (en el módulo unittest.mock), 1571
- patch.object () (en el módulo unittest.mock), 1569
- patch.stopall () (en el módulo unittest.mock), 1573
- path
 - configuration file, 1774
 - module search, 410, 1703, 1773
 - operations, 373, 389
- PATH, 596, 597, 601, 602, 608, 1199, 1206, 1208, 1773
- path (atributo de http.cookiejar.Cookie), 1318
- path (atributo de http.server.BaseHTTPRequestHandler), 1303
- path (atributo de importlib.abc.FileLoader), 1796
- path (atributo de importlib.machinery.ExtensionFileLoader),

- 1802
- `path` (atributo de `importlib.machinery.FileFinder`), 1801
- `path` (atributo de `importlib.machinery.SourceFileLoader`), 1801
- `path` (atributo de `importlib.machinery.SourcelessFileLoader`), 1802
- `path` (atributo de `os.DirEntry`), 585
- `Path` (clase en `pathlib`), 382
- `Path` (clase en `zipfile`), 490
- `path` (en el módulo `sys`), 1703
- `Path browser`, 1464
- `path()` (en el módulo `importlib.resources`), 1799
- `path_hook()` (método de clase de `importlib.machinery.FileFinder`), 1801
- `path_hooks` (en el módulo `sys`), 1703
- `path_importer_cache` (en el módulo `sys`), 1703
- `path_mtime()` (método de `importlib.abc.SourceLoader`), 1797
- `path_return_ok()` (método de `http.cookiejar.CookiePolicy`), 1315
- `path_stats()` (método de `importlib.abc.SourceLoader`), 1797
- `path_stats()` (método de `importlib.machinery.SourceFileLoader`), 1801
- `pathconf()` (en el módulo `os`), 581
- `pathconf_names` (en el módulo `os`), 582
- `PathEntryFinder` (clase en `importlib.abc`), 1792
- `PathFinder` (clase en `importlib.machinery`), 1800
- `pathlib` (módulo), 373
- `PathLike` (clase en `os`), 561
- `pathname2url()` (en el módulo `urllib.request`), 1221
- `pathsep` (en el módulo `os`), 608
- `pattern` (atributo de `re.error`), 120
- `pattern` (atributo de `re.Pattern`), 121
- `Pattern` (clase en `typing`), 1489
- `--pattern pattern`
- `unittest-discover` command line option, 1522
- `pause()` (en el módulo `signal`), 1027
- `pause_reading()` (método de `asyncio.ReadTransport`), 924
- `pause_writing()` (método de `asyncio.BaseProtocol`), 927
- `PAX_FORMAT` (en el módulo `tarfile`), 497
- `pax_headers` (atributo de `tarfile.TarFile`), 501
- `pax_headers` (atributo de `tarfile.TarInfo`), 502
- `pbkdf2_hmac()` (en el módulo `hashlib`), 546
- `pd()` (en el módulo `turtle`), 1396
- `Pdb` (clase en `pdb`), 1641
- `Pdb` (class in `pdb`), 1640
- `pdb` (módulo), 1640
- `.pdbrc`
- file, 1642
- `pdf()` (método de `statistics.NormalDist`), 340
- `peek()` (método de `bz2.BZ2File`), 476
- `peek()` (método de `gzip.GzipFile`), 474
- `peek()` (método de `io.BufferedReader`), 618
- `peek()` (método de `lzma.LZMAFile`), 480
- `peek()` (método de `weakref.finalize`), 249
- `peer` (atributo de `smtpd.SMTPChannel`), 1287
- `PEM_cert_to_DER_cert()` (en el módulo `ssl`), 978
- `pen()` (en el módulo `turtle`), 1396
- `pencolor()` (en el módulo `turtle`), 1397
- `pending` (atributo de `ssl.MemoryBIO`), 1004
- `pending()` (método de `ssl.SSLSocket`), 989
- `PendingDeprecationWarning`, 96
- `pendown()` (en el módulo `turtle`), 1396
- `pensize()` (en el módulo `turtle`), 1396
- `penup()` (en el módulo `turtle`), 1396
- `PEP`, 1944
- `PERCENT` (en el módulo `token`), 1828
- `PERCENTEQUAL` (en el módulo `token`), 1828
- `perf_counter()` (en el módulo `time`), 625
- `perf_counter_ns()` (en el módulo `time`), 625
- `Performance`, 1654
- `perm()` (en el módulo `math`), 290
- `PermissionError`, 96
- `permutations()` (en el módulo `itertools`), 350
- `Persist()` (método de `msilib.SummaryInformation`), 1863
- `persistence`, 421
- `persistent`
- objects, 421
- `persistent_id` (*pickle protocol*), 430
- `persistent_id()` (método de `pickle.Pickler`), 425
- `persistent_load` (*pickle protocol*), 430
- `persistent_load()` (método de `pickle.Unpickler`), 426
- `PF_CAN` (en el módulo `socket`), 954
- `PF_PACKET` (en el módulo `socket`), 955
- `PF_RDS` (en el módulo `socket`), 955
- `pformat()` (en el módulo `pprint`), 260
- `pformat()` (método de `pprint.PrettyPrinter`), 261
- `pgettext()` (en el módulo `gettext`), 1368
- `pgettext()` (método de `gettext.GNUTranslations`), 1372
- `pgettext()` (método de `gettext.NullTranslations`), 1370
- `PGO` (en el módulo `test.support`), 1615
- `phase()` (en el módulo `cmath`), 295
- `pi` (en el módulo `cmath`), 297
- `pi` (en el módulo `math`), 294
- `pi()` (método de `xml.etree.ElementTree.TreeBuilder`), 1158
- `pickle`
- módulo, 259, 437, 438, 441
- `pickle` (módulo), 421
- `pickle()` (en el módulo `copyreg`), 438
- `PickleBuffer` (clase en `pickle`), 426
- `PickleError`, 424
- `Pickler` (clase en `pickle`), 424
- `pickletools` (módulo), 1855
- `pickletools` command line option
- a, 1855
- annotate, 1855

--indentlevel=<num>, 1855
-l, 1855
-m, 1855
--memo, 1855
-o, 1855
--output=<file>, 1855
-p, 1855
--preamble=<preamble>, 1855
pickling
 objects, 421
PicklingError, 424
pid (atributo de *asyncio.asyncio.subprocess.Process*), 892
pid (atributo de *multiprocessing.Process*), 789
pid (atributo de *subprocess.Popen*), 845
PIPE (en el módulo *subprocess*), 838
Pipe() (en el módulo *multiprocessing*), 792
pipe() (en el módulo *os*), 570
pipe2() (en el módulo *os*), 570
PIPE_BUF (en el módulo *select*), 1008
pipe_connection_lost() (método de *asyncio.SubprocessProtocol*), 929
pipe_data_received() (método de *asyncio.SubprocessProtocol*), 929
PIPE_MAX_SIZE (en el módulo *test.support*), 1615
pipes (módulo), 1892
PKG_DIRECTORY (en el módulo *imp*), 1931
pkgutil (módulo), 1783
placeholder (atributo de *textwrap.TextWrapper*), 143
platform (en el módulo *sys*), 1703
platform (módulo), 727
platform() (en el módulo *platform*), 728
PlaySound() (en el módulo *winsound*), 1877
plist
 file, 539
plistlib (módulo), 539
plock() (en el módulo *os*), 599
PLUS (en el módulo *token*), 1827
plus() (método de *decimal.Context*), 313
PLUSEQUAL (en el módulo *token*), 1828
pm() (en el módulo *pdb*), 1641
pointer() (en el módulo *ctypes*), 762
POINTER() (en el módulo *ctypes*), 761
polar() (en el módulo *cmath*), 295
Policy (clase en *email.policy*), 1054
poll() (en el módulo *select*), 1007
poll() (método de *multiprocessing.connection.Connection*), 796
poll() (método de *select.devpoll*), 1009
poll() (método de *select.epoll*), 1010
poll() (método de *select.poll*), 1011
poll() (método de *subprocess.Popen*), 844
PollSelector (clase en *selectors*), 1016
Pool (clase en *multiprocessing.pool*), 809
pop() (método de *array.array*), 245
pop() (método de *collections.deque*), 223
pop() (método de *dict*), 79
pop() (método de *frozenset*), 77
pop() (método de *mailbox.Mailbox*), 1109
pop() (sequence method), 40
POP3
 protocol, 1262
POP3 (clase en *poplib*), 1262
POP3_SSL (clase en *poplib*), 1262
pop_alignment() (método de *formatter.formatter*), 1858
pop_all() (método de *contextlib.ExitStack*), 1735
POP_BLOCK (opcode), 1848
POP_EXCEPT (opcode), 1848
POP_FINALLY (opcode), 1848
pop_font() (método de *formatter.formatter*), 1858
POP_JUMP_IF_FALSE (opcode), 1851
POP_JUMP_IF_TRUE (opcode), 1851
pop_margin() (método de *formatter.formatter*), 1859
pop_source() (método de *shlex.shlex*), 1425
pop_style() (método de *formatter.formatter*), 1859
POP_TOP (opcode), 1845
Popen (clase en *subprocess*), 840
popen() (en el módulo *os*), 599
popen() (in module *os*), 1008
popitem() (método de *collections.OrderedDict*), 231
popitem() (método de *dict*), 79
popitem() (método de *mailbox.Mailbox*), 1109
popleft() (método de *collections.deque*), 223
poplib (módulo), 1262
PopupMenu (clase en *tkinter.tix*), 1460
porción, 1945
port (atributo de *http.cookiejar.Cookie*), 1318
port_specified (atributo de *http.cookiejar.Cookie*), 1318
pos (atributo de *json.JSONDecodeError*), 1102
pos (atributo de *re.error*), 120
pos (atributo de *re.Match*), 124
pos() (en el módulo *operator*), 367
pos() (en el módulo *turtle*), 1394
position (atributo de *xml.etree.ElementTree.ParseError*), 1161
position() (en el módulo *turtle*), 1394
POSIX
 I/O control, 1886
 threads, 862
posix (módulo), 1881
POSIX Shared Memory, 825
POSIX_FADV_DONTNEED (en el módulo *os*), 571
POSIX_FADV_NOREUSE (en el módulo *os*), 571
POSIX_FADV_NORMAL (en el módulo *os*), 571
POSIX_FADV_RANDOM (en el módulo *os*), 571
POSIX_FADV_SEQUENTIAL (en el módulo *os*), 571
POSIX_FADV_WILLNEED (en el módulo *os*), 571
posix_fadvise() (en el módulo *os*), 570
posix_fallocate() (en el módulo *os*), 570
posix_spawn() (en el módulo *os*), 599
POSIX_SPAWN_CLOSE (en el módulo *os*), 600
POSIX_SPAWN_DUP2 (en el módulo *os*), 600
POSIX_SPAWN_OPEN (en el módulo *os*), 600

- `posix_spawn()` (en el módulo `os`), 600
`POSIXLY_CORRECT`, 663
`PosixPath` (clase en `pathlib`), 382
`post()` (método de `nnplib.NNTP`), 1277
`post()` (método de `ossaudiodev.oss_audio_device`), 1364
`post_handshake_auth` (atributo de `ssl.SSLContext`), 996
`post_mortem()` (en el módulo `pdb`), 1641
`post_setup()` (método de `venv.EnvBuilder`), 1679
`postcmd()` (método de `cmd.Cmd`), 1419
`postloop()` (método de `cmd.Cmd`), 1419
`pow()` (en el módulo `math`), 292
`pow()` (en el módulo `operator`), 367
`pow()` (función incorporada), 19
`power()` (método de `decimal.Context`), 313
`pp` (`pdb` command), 1645
`pp()` (en el módulo `pprint`), 260
`pprint` (módulo), 259
`pprint()` (en el módulo `pprint`), 260
`pprint()` (método de `pprint.PrettyPrinter`), 261
`prcal()` (en el módulo `calendar`), 216
`pread()` (en el módulo `os`), 571
`preadv()` (en el módulo `os`), 571
`preamble` (atributo de `email.message.EmailMessage`), 1045
`preamble` (atributo de `email.message.Message`), 1083
`--preamble=<preamble>`
`pickletools` command line option, 1855
`precmd()` (método de `cmd.Cmd`), 1419
`prefix` (atributo de `xml.dom.Attr`), 1168
`prefix` (atributo de `xml.dom.Node`), 1164
`prefix` (atributo de `zipimport.zipimporter`), 1782
`prefix` (en el módulo `sys`), 1704
`PREFIXES` (en el módulo `site`), 1775
`prefixlen` (atributo de `ipaddress.IPv4Network`), 1339
`prefixlen` (atributo de `ipaddress.IPv6Network`), 1342
`preloop()` (método de `cmd.Cmd`), 1419
`prepare()` (método de `logging.handlers.QueueHandler`), 703
`prepare()` (método de `logging.handlers.QueueListener`), 704
`prepare_class()` (en el módulo `types`), 253
`prepare_input_source()` (en el módulo `xml.sax.saxutils`), 1184
`prepend()` (método de `pipes.Template`), 1892
`PrettyPrinter` (clase en `pprint`), 259
`prev()` (método de `tkinter.ttk.Treeview`), 1454
`previousSibling` (atributo de `xml.dom.Node`), 1163
`print` (2to3 fixer), 1611
`print()` (función incorporada), 20
`print_callees()` (método de `pstats.Stats`), 1652
`print_callers()` (método de `pstats.Stats`), 1652
`print_directory()` (en el módulo `cgi`), 1206
`print_envron()` (en el módulo `cgi`), 1206
`print_envron_usage()` (en el módulo `cgi`), 1206
`print_exc()` (en el módulo `traceback`), 1748
`print_exc()` (método de `timeit.Timer`), 1656
`print_exception()` (en el módulo `traceback`), 1747
`PRINT_EXPR` (opcode), 1848
`print_form()` (en el módulo `cgi`), 1206
`print_help()` (método de `argparse.ArgumentParser`), 660
`print_last()` (en el módulo `traceback`), 1748
`print_stack()` (en el módulo `traceback`), 1748
`print_stack()` (método de `asyncio.Task`), 877
`print_stats()` (método de `profile.Profile`), 1650
`print_stats()` (método de `pstats.Stats`), 1652
`print_tb()` (en el módulo `traceback`), 1747
`print_usage()` (método de `argparse.ArgumentParser`), 660
`print_usage()` (método de `optparse.OptionParser`), 1920
`print_version()` (método de `optparse.OptionParser`), 1910
`printable` (en el módulo `string`), 100
`printdir()` (método de `zipfile.ZipFile`), 488
`printf-style formatting`, 52, 66
`PRIOPGRP` (en el módulo `os`), 563
`PRIOPROCESS` (en el módulo `os`), 563
`PRIOUSER` (en el módulo `os`), 563
`PriorityQueue` (clase en `asyncio`), 894
`PriorityQueue` (clase en `queue`), 856
`prlimit()` (en el módulo `resource`), 1893
`prmonth()` (en el módulo `calendar`), 216
`prmonth()` (método de `calendar.TextCalendar`), 214
`ProactorEventLoop` (clase en `asyncio`), 915
`process`
`group`, 562, 563
`id`, 563
`id of parent`, 563
`killring`, 599
`scheduling priority`, 563, 564
`signalling`, 599
`--process`
`timeit` command line option, 1657
`Process` (clase en `multiprocessing`), 788
`process()` (método de `logging.LoggerAdapter`), 676
`process_exited()` (método de `asyncio.SubprocessProtocol`), 929
`process_message()` (método de `smtpd.SMTPServer`), 1285
`process_request()` (método de `socketserver.BaseServer`), 1298
`process_time()` (en el módulo `time`), 625
`process_time_ns()` (en el módulo `time`), 625
`process_tokens()` (en el módulo `tabnanny`), 1834
`ProcessError`, 790
`processes, light-weight`, 862
`ProcessingInstruction` (en el módulo `xml.etree.ElementTree`), 1151
`processingInstruction()` (método de `xml.sax.handler.ContentHandler`), 1183

`ProcessingInstructionHandler()` (método de `xml.parsers.expat.xmlparser`), 1193

`ProcessLookupError`, 96

`processor time`, 625, 628

`processor()` (en el módulo `platform`), 728

`ProcessPoolExecutor` (clase en `concurrent.futures`), 832

`prod()` (en el módulo `math`), 291

`product()` (en el módulo `itertools`), 351

`Profile` (clase en `profile`), 1649

`profile` (módulo), 1649

`profile function`, 770, 1699, 1705

`profiler`, 1699, 1705

`profiling, deterministic`, 1646

`ProgrammingError`, 460

`Progressbar` (clase en `tkinter.ttk`), 1449

`prompt` (atributo de `cmd.Cmd`), 1420

`prompt_user_passwd()` (método de `urllib.request.FancyURLopener`), 1237

`prompts, interpreter`, 1704

`propagate` (atributo de `logging.Logger`), 666

`property` (clase incorporada), 20

`property list`, 539

`property_declaration_handler` (en el módulo `xml.sax.handler`), 1180

`property_dom_node` (en el módulo `xml.sax.handler`), 1181

`property_lexical_handler` (en el módulo `xml.sax.handler`), 1180

`property_xml_string` (en el módulo `xml.sax.handler`), 1181

`PropertyMock` (clase en `unittest.mock`), 1558

`prot_c()` (método de `ftplib.FTP_TLS`), 1262

`prot_p()` (método de `ftplib.FTP_TLS`), 1262

`proto` (atributo de `socket.socket`), 969

`protocol`

- CGI, 1202
- context management, 82
- copy, 429
- FTP, 1237, 1257
- HTTP, 1202, 1237, 1248, 1250, 1302
- IMAP4, 1265
- IMAP4_SSL, 1265
- IMAP4_stream, 1265
- iterator, 37
- NNTP, 1271
- POP3, 1262
- SMTP, 1278
- Telnet, 1288

`protocol` (atributo de `ssl.SSLContext`), 996

`Protocol` (clase en `asyncio`), 926

`Protocol` (clase en `typing`), 1484

`PROTOCOL_SSLv2` (en el módulo `ssl`), 981

`PROTOCOL_SSLv3` (en el módulo `ssl`), 981

`PROTOCOL_SSLv23` (en el módulo `ssl`), 981

`PROTOCOL_TLS` (en el módulo `ssl`), 980

`PROTOCOL_TLS_CLIENT` (en el módulo `ssl`), 980

`PROTOCOL_TLS_SERVER` (en el módulo `ssl`), 980

`PROTOCOL_TLSv1` (en el módulo `ssl`), 981

`PROTOCOL_TLSv1_1` (en el módulo `ssl`), 981

`PROTOCOL_TLSv1_2` (en el módulo `ssl`), 981

`protocol_version` (atributo de `http.server.BaseHTTPRequestHandler`), 1304

`PROTOCOL_VERSION` (atributo de `imaplib.IMAP4`), 1271

`ProtocolError` (clase en `xmlrpc.client`), 1325

`proxy()` (en el módulo `weakref`), 248

`proxyauth()` (método de `imaplib.IMAP4`), 1268

`ProxyBasicAuthHandler` (clase en `urllib.request`), 1224

`ProxyDigestAuthHandler` (clase en `urllib.request`), 1224

`ProxyHandler` (clase en `urllib.request`), 1223

`ProxyType` (en el módulo `weakref`), 249

`ProxyTypes` (en el módulo `weakref`), 249

`pryear()` (método de `calendar.TextCalendar`), 214

`ps1` (en el módulo `sys`), 1704

`ps2` (en el módulo `sys`), 1704

`pstats` (módulo), 1650

`pstdev()` (en el módulo `statistics`), 336

`pthread_getcpuclockid()` (en el módulo `time`), 623

`pthread_kill()` (en el módulo `signal`), 1027

`pthread_sigmask()` (en el módulo `signal`), 1027

`pthreads`, 862

`pty`

- módulo, 570
- (módulo), 1888

`pu()` (en el módulo `turtle`), 1396

`publicId` (atributo de `xml.dom.DocumentType`), 1165

`PullDom` (clase en `xml.dom.pulldom`), 1176

`punctuation` (en el módulo `string`), 100

`punctuation_chars` (atributo de `shlex.shlex`), 1426

`PurePath` (clase en `pathlib`), 375

- `PurePath.anchor` (en el módulo `pathlib`), 378
- `PurePath.drive` (en el módulo `pathlib`), 378
- `PurePath.name` (en el módulo `pathlib`), 379
- `PurePath.parent` (en el módulo `pathlib`), 378
- `PurePath.parents` (en el módulo `pathlib`), 378
- `PurePath.parts` (en el módulo `pathlib`), 377
- `PurePath.root` (en el módulo `pathlib`), 378
- `PurePath.stem` (en el módulo `pathlib`), 379
- `PurePath.suffix` (en el módulo `pathlib`), 379
- `PurePath.suffixes` (en el módulo `pathlib`), 379

`PurePosixPath` (clase en `pathlib`), 376

`PureProxy` (clase en `smtpd`), 1286

`PureWindowsPath` (clase en `pathlib`), 376

`purge()` (en el módulo `re`), 120

`Purpose.CLIENT_AUTH` (en el módulo `ssl`), 985

`Purpose.SERVER_AUTH` (en el módulo `ssl`), 984

`push()` (método de `asynchat.async_chat`), 1022

`push()` (método de `code.InteractiveConsole`), 1779

`push()` (método de `contextlib.ExitStack`), 1734

`push_alignment()` (método de `formatter.formatter`), 1858

- `push_async_callback()` (método de `contextlib.AsyncExitStack`), 1735
- `push_async_exit()` (método de `contextlib.AsyncExitStack`), 1735
- `push_font()` (método de `formatter.formatter`), 1858
- `push_margin()` (método de `formatter.formatter`), 1858
- `push_source()` (método de `shlex.shlex`), 1425
- `push_style()` (método de `formatter.formatter`), 1859
- `push_token()` (método de `shlex.shlex`), 1424
- `push_with_producer()` (método de `async_chat.async_chat`), 1022
- `pushbutton()` (método de `msilib.Dialog`), 1866
- `put()` (método de `asyncio.Queue`), 894
- `put()` (método de `multiprocessing.Queue`), 792
- `put()` (método de `multiprocessing.SimpleQueue`), 793
- `put()` (método de `queue.Queue`), 856
- `put()` (método de `queue.SimpleQueue`), 858
- `put_nowait()` (método de `asyncio.Queue`), 894
- `put_nowait()` (método de `multiprocessing.Queue`), 792
- `put_nowait()` (método de `queue.Queue`), 857
- `put_nowait()` (método de `queue.SimpleQueue`), 858
- `putch()` (en el módulo `msvcrt`), 1868
- `putenv()` (en el módulo `os`), 563
- `putheader()` (método de `http.client.HTTPConnection`), 1254
- `putp()` (en el módulo `curses`), 709
- `putrequest()` (método de `http.client.HTTPConnection`), 1254
- `putwch()` (en el módulo `msvcrt`), 1868
- `putwin()` (método de `curses.window`), 716
- `pvariance()` (en el módulo `statistics`), 336
- `pwd`
módulo, 391
- `pwd` (módulo), 1882
- `pwd()` (método de `ftplib.FTP`), 1261
- `pwrite()` (en el módulo `os`), 572
- `pwritev()` (en el módulo `os`), 572
- `py_compile` (módulo), 1836
- `PY_COMPILED` (en el módulo `imp`), 1931
- `PY_FROZEN` (en el módulo `imp`), 1931
- `py_object` (clase en `ctypes`), 765
- `PY_SOURCE` (en el módulo `imp`), 1931
- `pycache_prefix` (en el módulo `sys`), 1694
- `PycInvalidationMode` (clase en `py_compile`), 1837
- `pyclbr` (módulo), 1835
- `PyCompileError`, 1836
- `PyDLL` (clase en `ctypes`), 755
- `pydoc` (módulo), 1496
- `pyexpat`
módulo, 1189
- `PYFUNCTYPE()` (en el módulo `ctypes`), 758
- `Python 3000`, 1945
- `Python Editor`, 1464
- `Python Enhancement Proposals`
PEP 1, 1944
- PEP 205, 250
- PEP 227, 1754
- PEP 235, 1790
- PEP 237, 54, 68
- PEP 238, 1754, 1939
- PEP 249, 446, 448
- PEP 255, 1754
- PEP 263, 1790, 1830, 1831
- PEP 273, 1781
- PEP 278, 1947
- PEP 282, 417, 681
- PEP 292, 108
- PEP 302, 25, 410, 1703, 1781, 1783, 1785, 1790, 1792, 1793, 1795, 1796, 1932, 1939, 1942
- PEP 305, 511
- PEP 307, 423
- PEP 324, 836
- PEP 328, 25, 1754, 1790
- PEP 338, 1789
- PEP 342, 235
- PEP 343, 1739, 1754, 1937
- PEP 362, 1766, 1936, 1944
- PEP 366, 1789, 1790
- PEP 370, 1776
- PEP 378, 103
- PEP 383, 162, 951
- PEP 393, 168, 1703
- PEP 397, 1678
- PEP 405, 1675
- PEP 411, 1700, 1707, 1945
- PEP 420, 1790, 1939, 1943, 1945
- PEP 421, 1701
- PEP 428, 374
- PEP 442, 1757
- PEP 443, 1940
- PEP 451, 1703, 1784, 1789, 1790, 1939
- PEP 453, 1674
- PEP 461, 68
- PEP 468, 231
- PEP 475, 19, 96, 569, 573, 574, 604, 626, 963, 965, 967, 1008, 1012, 1015, 1029, 1030
- PEP 479, 93, 1754
- PEP 483, 1477
- PEP 484, 1477, 1479, 1483, 1485, 1492, 1821, 1935, 1939, 1947
- PEP 485, 290, 297
- PEP 488, 1616, 1790, 1804, 1836
- PEP 489, 1790, 1800, 1802
- PEP 492, 236, 1772, 1936, 1937
- PEP 498, 1939
- PEP 506, 555
- PEP 515, 103
- PEP 519, 1944
- PEP 524, 609
- PEP 525, 236, 1700, 1707, 1772, 1936
- PEP 526, 1477, 1490, 1495, 1721, 1727, 1821, 1935, 1947
- PEP 529, 579, 1698, 1707

- PEP 544, 1477, 1483, 1485
PEP 552, 1790, 1837
PEP 557, 1721
PEP 560, 254
PEP 563, 1754
PEP 566, 1808, 1811
PEP 567, 858, 900, 901, 920
PEP 574, 423, 435
PEP 578, 1629, 1691
PEP 585, 1495
PEP 586, 1477
PEP 589, 1477, 1491
PEP 591, 1477, 1492, 1495
PEP 706, 503
PEP 3101, 100
PEP 3105, 1754
PEP 3112, 1754
PEP 3115, 254
PEP 3116, 1947
PEP 3118, 70
PEP 3119, 237, 1741
PEP 3120, 1790
PEP 3141, 285, 1741
PEP 3147, 1616, 1788, 1790, 1804, 18361840, 1930
PEP 3148, 835
PEP 3149, 1691
PEP 3151, 96, 953, 1007, 1893
PEP 3154, 423
PEP 3155, 1945
PEP 3333, 12091214, 1217, 1218
--python=<interpreter>
zipapp command line option, 1684
python_branch() (en el módulo platform), 728
python_build() (en el módulo platform), 728
python_compiler() (en el módulo platform), 728
PYTHON_DOM, 1162
python_implementation() (en el módulo platform), 728
python_is_optimized() (en el módulo test.support), 1616
python_revision() (en el módulo platform), 728
python_version() (en el módulo platform), 728
python_version_tuple() (en el módulo platform), 728
PYTHONASYNCIODEBUG, 912, 947
PYTHONBREAKPOINT, 1693
PYTHONDEVMODE, 1628
PYTHONDOCS, 1497
PYTHONDONTWRITEBYTECODE, 1694
PYTHONFAULTHANDLER, 1638
PYTHONHOME, 1627
Pythónico, 1945
PYTHONINTMAXSTRDIGITS, 87, 1702
PYTHONIOENCODING, 1708
PYTHONLEGACYWINDOWSFSENCODING, 1707
PYTHONLEGACYWINDOWSSTDIO, 1708
PYTHONNOUSERSITE, 1775
PYTHONPATH, 1206, 1627, 1703
PYTHONPYCACHEPREFIX, 1694
PYTHONSTARTUP, 150, 1471, 1702, 1774
PYTHONTRACEMALLOC, 1662, 1666
PYTHONTRACEMALLOC`comienzo, configura la variable del entorno a ``25`, 1662
PYTHONUSERBASE, 1775
PYTHONUSERSITE, 1627
PYTHONUTF8, 1708
PYTHONWARNINGS, 1717
PyZipFile (clase en zipfile), 491
- ## Q
- q
compileall command line option, 1838
qiflush() (en el módulo curses), 709
QName (clase en xml.etree.ElementTree), 1157
qsize() (método de asyncio.Queue), 894
qsize() (método de multiprocessing.Queue), 792
qsize() (método de queue.Queue), 856
qsize() (método de queue.SimpleQueue), 858
quantiles() (en el módulo statistics), 338
quantiles() (método de statistics.NormalDist), 340
quantize() (método de decimal.Context), 313
quantize() (método de decimal.Decimal), 307
QueryInfoKey() (en el módulo winreg), 1872
QueryReflectionKey() (en el módulo winreg), 1874
QueryValue() (en el módulo winreg), 1872
QueryValueEx() (en el módulo winreg), 1872
queue (atributo de sched.scheduler), 855
Queue (clase en asyncio), 893
Queue (clase en multiprocessing), 792
Queue (clase en queue), 855
queue (módulo), 855
Queue() (método de multiprocessing.managers.SyncManager), 804
QueueEmpty, 895
QueueFull, 895
QueueHandler (clase en logging.handlers), 703
QueueListener (clase en logging.handlers), 703
quick_ratio() (método de difflib.SequenceMatcher), 136
quit (pdb command), 1646
quit (variable incorporada), 28
quit() (método de ftplib.FTP), 1261
quit() (método de nntplib.NNTP), 1274
quit() (método de poplib.POP3), 1264
quit() (método de smtplib.SMTP), 1284
quopri (módulo), 1133
quote() (en el módulo email.utils), 1092
quote() (en el módulo shlex), 1423
quote() (en el módulo urllib.parse), 1244
QUOTE_ALL (en el módulo csv), 514
quote_from_bytes() (en el módulo urllib.parse), 1245

QUOTE_MINIMAL (en el módulo *csv*), 514
 QUOTE_NONE (en el módulo *csv*), 514
 QUOTE_NONNUMERIC (en el módulo *csv*), 514
 quote_plus() (en el módulo *urllib.parse*), 1245
 quoteattr() (en el módulo *xml.sax.saxutils*), 1184
 quotechar (atributo de *csv.Dialect*), 515
 quoted-printable
 encoding, 1133
 quotes (atributo de *shlex.shlex*), 1425
 quoting (atributo de *csv.Dialect*), 515

R

-r
 compileall command line option, 1838
 trace command line option, 1660
 -R
 trace command line option, 1660
 -r N
 timeit command line option, 1657
 R_OK (en el módulo *os*), 576
 radians() (en el módulo *math*), 293
 radians() (en el módulo *turtle*), 1396
 RadioButtonGroup (clase en *msilib*), 1866
 radiogroup() (método de *msilib.Dialog*), 1866
 radix() (método de *decimal.Context*), 313
 radix() (método de *decimal.Decimal*), 307
 RADIXCHAR (en el módulo *locale*), 1378
 raise
 sentencia, 89
 raise (2to3 fixer), 1611
 raise_on_defect (atributo de *email.policy.Policy*), 1055
 raise_signal() (en el módulo *signal*), 1027
 RAISE_VARARGS (opcode), 1852
 RAND_add() (en el módulo *ssl*), 977
 RAND_bytes() (en el módulo *ssl*), 976
 RAND_egd() (en el módulo *ssl*), 977
 RAND_pseudo_bytes() (en el módulo *ssl*), 976
 RAND_status() (en el módulo *ssl*), 976
 randbelow() (en el módulo *secrets*), 555
 randbits() (en el módulo *secrets*), 555
 randint() (en el módulo *random*), 327
 Random (clase en *random*), 330
 random (módulo), 326
 random() (en el módulo *random*), 329
 randrange() (en el módulo *random*), 327
 range
 objeto, 42
 range (clase incorporada), 42
 RARROW (en el módulo *token*), 1829
 ratecv() (en el módulo *audioop*), 1349
 ratio() (método de *diffib.SequenceMatcher*), 136
 Rational (clase en *numbers*), 286
 raw (atributo de *io.BufferedIOBase*), 615
 raw() (en el módulo *curses*), 710
 raw() (método de *pickle.PickleBuffer*), 427

raw_data_manager (en el módulo *email.contentmanager*), 1068
 raw_decode() (método de *json.JSONDecoder*), 1100
 raw_input (2to3 fixer), 1611
 raw_input() (método de *code.InteractiveConsole*), 1779
 RawArray() (en el módulo *multiprocessing.sharedctypes*), 800
 RawConfigParser (clase en *configparser*), 534
 RawDescriptionHelpFormatter (clase en *argparse*), 637
 RawIOBase (clase en *io*), 614
 RawPen (clase en *turtle*), 1412
 RawTextHelpFormatter (clase en *argparse*), 637
 RawTurtle (clase en *turtle*), 1412
 RawValue() (en el módulo *multiprocessing.sharedctypes*), 801
 RBRACE (en el módulo *token*), 1828
 rcpttos (atributo de *smtpd.SMTPChannel*), 1287
 re
 módulo, 45, 409
 re (atributo de *re.Match*), 124
 re (módulo), 110
 read() (en el módulo *os*), 572
 read() (método de *asyncio.StreamReader*), 880
 read() (método de *chunk.Chunk*), 1359
 read() (método de *codecs.StreamReader*), 166
 read() (método de *configparser.ConfigParser*), 531
 read() (método de *http.client.HTTPResponse*), 1255
 read() (método de *imaplib.IMAP4*), 1269
 read() (método de *io.BufferedIOBase*), 615
 read() (método de *io.BufferedReader*), 618
 read() (método de *io.RawIOBase*), 614
 read() (método de *io.TextIOBase*), 619
 read() (método de *mimetypes.MimeTypes*), 1126
 read() (método de *mmap.mmap*), 1033
 read() (método de *ossaudiodev.oss_audio_device*), 1362
 read() (método de *ssl.MemoryBIO*), 1004
 read() (método de *ssl.SSLSocket*), 986
 read() (método de *urllib.robotparser.RobotFileParser*), 1247
 read() (método de *zipfile.ZipFile*), 489
 read1() (método de *io.BufferedIOBase*), 615
 read1() (método de *io.BufferedReader*), 618
 read1() (método de *io.BytesIO*), 617
 read_all() (método de *telnetlib.Telnet*), 1289
 read_binary() (en el módulo *importlib.resources*), 1798
 read_byte() (método de *mmap.mmap*), 1033
 read_bytes() (método de *pathlib.Path*), 386
 read_bytes() (método de *zipfile.Path*), 490
 read_dict() (método de *configparser.ConfigParser*), 532
 read_eager() (método de *telnetlib.Telnet*), 1289
 read_enviro() (en el módulo *wsgiref.handlers*), 1218
 read_events() (método de

- `xml.etree.ElementTree.XMLPullParser()`, 1160
- `read_file()` (método de `configparser.ConfigParser`), 532
- `read_history_file()` (en el módulo `readline`), 148
- `read_init_file()` (en el módulo `readline`), 147
- `read_lazy()` (método de `telnetlib.Telnet`), 1289
- `read_mime_types()` (en el módulo `mimetypes`), 1125
- `read_sb_data()` (método de `telnetlib.Telnet`), 1289
- `read_some()` (método de `telnetlib.Telnet`), 1289
- `read_string()` (método de `configparser.ConfigParser`), 532
- `read_text()` (en el módulo `importlib.resources`), 1798
- `read_text()` (método de `pathlib.Path`), 386
- `read_text()` (método de `zipfile.Path`), 490
- `read_token()` (método de `shlex.shlex`), 1424
- `read_until()` (método de `telnetlib.Telnet`), 1289
- `read_very_eager()` (método de `telnetlib.Telnet`), 1289
- `read_very_lazy()` (método de `telnetlib.Telnet`), 1289
- `read_windows_registry()` (método de `mimetypes.MimeTypes`), 1126
- `READABLE` (en el módulo `tkinter`), 1440
- `readable()` (método de `asyncore.dispatcher`), 1019
- `readable()` (método de `io.IOBase`), 613
- `readall()` (método de `io.RawIOBase`), 615
- `reader()` (en el módulo `csv`), 512
- `ReadError`, 496
- `readexactly()` (método de `asyncio.StreamReader`), 881
- `readfp()` (método de `configparser.ConfigParser`), 533
- `readfp()` (método de `mimetypes.MimeTypes`), 1126
- `readframes()` (método de `aifc.aifc`), 1351
- `readframes()` (método de `sunau.AU_read`), 1354
- `readframes()` (método de `wave.Wave_read`), 1356
- `readinto()` (método de `http.client.HTTPResponse`), 1255
- `readinto()` (método de `io.BufferedIOBase`), 616
- `readinto()` (método de `io.RawIOBase`), 615
- `readinto1()` (método de `io.BufferedIOBase`), 616
- `readinto1()` (método de `io.BytesIO`), 617
- `readline` (módulo), 147
- `readline()` (método de `asyncio.StreamReader`), 880
- `readline()` (método de `codecs.StreamReader`), 166
- `readline()` (método de `imaplib.IMAP4`), 1269
- `readline()` (método de `io.IOBase`), 613
- `readline()` (método de `io.TextIOBase`), 619
- `readline()` (método de `mmap.mmap`), 1034
- `readlines()` (método de `codecs.StreamReader`), 166
- `readlines()` (método de `io.IOBase`), 613
- `readlink()` (en el módulo `os`), 582
- `readmodule()` (en el módulo `pyclbr`), 1835
- `readmodule_ex()` (en el módulo `pyclbr`), 1835
- `readonly` (atributo de `memoryview`), 74
- `readPlist()` (en el módulo `plistlib`), 540
- `readPlistFromBytes()` (en el módulo `plistlib`), 540
- `ReadTransport` (clase en `asyncio`), 922
- `readuntil()` (método de `asyncio.StreamReader`), 881
- `readv()` (en el módulo `os`), 573
- `ready()` (método de `multiprocessing.pool.AsyncResult`), 811
- `real` (atributo de `numbers.Complex`), 285
- `Real` (clase en `numbers`), 285
- `Real Media File Format`, 1358
- `real_max_memuse` (en el módulo `test.support`), 1616
- `real_quick_ratio()` (método de `difflib.SequenceMatcher`), 136
- `realpath()` (en el módulo `os.path`), 393
- `REALTIME_PRIORITY_CLASS` (en el módulo `subprocess`), 847
- `reap_children()` (en el módulo `test.support`), 1623
- `reap_threads()` (en el módulo `test.support`), 1621
- `reason` (atributo de `http.client.HTTPResponse`), 1255
- `reason` (atributo de `ssl.SSLError`), 975
- `reason` (atributo de `UnicodeError`), 94
- `reason` (atributo de `urllib.error.HTTPError`), 1247
- `reason` (atributo de `urllib.error.URLError`), 1246
- `reattach()` (método de `tkinter.ttk.Treeview`), 1454
- `rebanada`, 1946
- `recontrols()` (método de `ossaudiodev.oss_mixer_device`), 1365
- `received_data` (atributo de `smtpd.SMTPChannel`), 1287
- `received_lines` (atributo de `smtpd.SMTPChannel`), 1287
- `recent()` (método de `imaplib.IMAP4`), 1269
- `recolección de basura`, 1939
- `reconfigure()` (método de `io.TextIOWrapper`), 620
- `record_original_stdout()` (en el módulo `test.support`), 1618
- `records` (atributo de `unittest.TestCase`), 1532
- `rect()` (en el módulo `cmath`), 295
- `rectangle()` (en el módulo `curses.textpad`), 722
- `RecursionError`, 92
- `recursive_repr()` (en el módulo `reprlib`), 265
- `recv()` (método de `asyncore.dispatcher`), 1019
- `recv()` (método de `multiprocessing.connection.Connection`), 796
- `recv()` (método de `socket.socket`), 965
- `recv_bytes()` (método de `multiprocessing.connection.Connection`), 796
- `recv_bytes_into()` (método de `multiprocessing.connection.Connection`), 796
- `recv_into()` (método de `socket.socket`), 966
- `recvfrom()` (método de `socket.socket`), 965
- `recvfrom_into()` (método de `socket.socket`), 966
- `recvmsg()` (método de `socket.socket`), 965
- `recvmsg_into()` (método de `socket.socket`), 966
- `redirect_request()` (método de `urllib.request.HTTPRedirectHandler`), 1228
- `redirect_stderr()` (en el módulo `contextlib`), 1733
- `redirect_stdout()` (en el módulo `contextlib`), 1732

- `redisplay()` (en el módulo `readline`), 148
`redrawln()` (método de `curses.window`), 716
`redrawwin()` (método de `curses.window`), 716
`reduce (2to3 fixer)`, 1611
`reduce()` (en el módulo `functools`), 361
`reducer_override()` (método de `pickle.Pickler`), 425
`ref` (clase en `weakref`), 247
`refcount_test()` (en el módulo `test.support`), 1621
`ReferenceError`, 92
`ReferenceType` (en el módulo `weakref`), 249
`refold_source` (atributo de `email.policy.EmailPolicy`), 1057
`refresh()` (método de `curses.window`), 717
`REG_BINARY` (en el módulo `winreg`), 1876
`REG_DWORD` (en el módulo `winreg`), 1876
`REG_DWORD_BIG_ENDIAN` (en el módulo `winreg`), 1876
`REG_DWORD_LITTLE_ENDIAN` (en el módulo `winreg`), 1876
`REG_EXPAND_SZ` (en el módulo `winreg`), 1876
`REG_FULL_RESOURCE_DESCRIPTOR` (en el módulo `winreg`), 1876
`REG_LINK` (en el módulo `winreg`), 1876
`REG_MULTI_SZ` (en el módulo `winreg`), 1876
`REG_NONE` (en el módulo `winreg`), 1876
`REG_QWORD` (en el módulo `winreg`), 1876
`REG_QWORD_LITTLE_ENDIAN` (en el módulo `winreg`), 1876
`REG_RESOURCE_LIST` (en el módulo `winreg`), 1876
`REG_RESOURCE_REQUIREMENTS_LIST` (en el módulo `winreg`), 1876
`REG_SZ` (en el módulo `winreg`), 1876
`register()` (en el módulo `atexit`), 1746
`register()` (en el módulo `codecs`), 160
`register()` (en el módulo `faulthandler`), 1639
`register()` (en el módulo `webbrowser`), 1200
`register()` (método de `abc.ABCMeta`), 1742
`register()` (método de `multiprocessing.managers.BaseManager`), 803
`register()` (método de `select.devpoll`), 1009
`register()` (método de `select.epoll`), 1010
`register()` (método de `selectors.BaseSelector`), 1015
`register()` (método de `select.poll`), 1011
`register_adapter()` (en el módulo `sqlite3`), 449
`register_archive_format()` (en el módulo `shutil`), 418
`register_at_fork()` (en el módulo `os`), 601
`register_converter()` (en el módulo `sqlite3`), 449
`register_defect()` (método de `email.policy.Policy`), 1056
`register_dialect()` (en el módulo `csv`), 512
`register_error()` (en el módulo `codecs`), 162
`register_function()` (método de `xmlrpc.server.CGIXMLRPCRequestHandler`), 1331
`register_function()` (método de `xmlrpc.server.SimpleXMLRPCServer`), 1328
`register_instance()` (método de `xmlrpc.server.CGIXMLRPCRequestHandler`), 1331
`register_instance()` (método de `xmlrpc.server.SimpleXMLRPCServer`), 1328
`register_introspection_functions()` (método de `xmlrpc.server.CGIXMLRPCRequestHandler`), 1331
`register_introspection_functions()` (método de `xmlrpc.server.SimpleXMLRPCServer`), 1329
`register_multicall_functions()` (método de `xmlrpc.server.CGIXMLRPCRequestHandler`), 1332
`register_multicall_functions()` (método de `xmlrpc.server.SimpleXMLRPCServer`), 1329
`register_namespace()` (en el módulo `xml.etree.ElementTree`), 1151
`register_optionflag()` (en el módulo `doctest`), 1505
`register_shape()` (en el módulo `turtle`), 1411
`register_unpack_format()` (en el módulo `shutil`), 418
`registerDOMImplementation()` (en el módulo `xml.dom`), 1162
`registerResult()` (en el módulo `unittest`), 1547
`relative`
 URL, 1238
`relative_to()` (método de `pathlib.PurePath`), 381
`release()` (en el módulo `platform`), 728
`release()` (método de `_thread.lock`), 863
`release()` (método de `asyncio.Condition`), 887
`release()` (método de `asyncio.Lock`), 886
`release()` (método de `asyncio.Semaphore`), 888
`release()` (método de `logging.Handler`), 669
`release()` (método de `memoryview`), 71
`release()` (método de `multiprocessing.Lock`), 798
`release()` (método de `multiprocessing.RLock`), 799
`release()` (método de `pickle.PickleBuffer`), 427
`release()` (método de `threading.Condition`), 777
`release()` (método de `threading.Lock`), 774
`release()` (método de `threading.RLock`), 775
`release()` (método de `threading.Semaphore`), 778
`release_lock()` (en el módulo `imp`), 1931
`reload (2to3 fixer)`, 1611
`reload()` (en el módulo `imp`), 1929
`reload()` (en el módulo `importlib`), 1791
`relpath()` (en el módulo `os.path`), 393
`remainder()` (en el módulo `math`), 291
`remainder()` (método de `decimal.Context`), 314
`remainder_near()` (método de `decimal.Context`), 314
`remainder_near()` (método de `decimal.Decimal`), 307
`RemoteDisconnected`, 1252

- `remove()` (en el módulo `os`), 582
- `remove()` (método de `array.array`), 245
- `remove()` (método de `collections.deque`), 223
- `remove()` (método de `frozenset`), 77
- `remove()` (método de `mailbox.Mailbox`), 1107
- `remove()` (método de `mailbox.MH`), 1113
- `remove()` (método de `xml.etree.ElementTree.Element`), 1155
- `remove()` (sequence method), 40
- `remove_child_handler()` (método de `asyncio.AbstractChildWatcher`), 936
- `remove_done_callback()` (método de `asyncio.Future`), 920
- `remove_done_callback()` (método de `asyncio.Task`), 877
- `remove_flag()` (método de `mailbox.MaildirMessage`), 1116
- `remove_flag()` (método de `mailbox.mboxMessage`), 1117
- `remove_flag()` (método de `mailbox.MMDfMessage`), 1121
- `remove_folder()` (método de `mailbox.Maildir`), 1110
- `remove_folder()` (método de `mailbox.MH`), 1112
- `remove_header()` (método de `urllib.request.Request`), 1225
- `remove_history_item()` (en el módulo `readline`), 148
- `remove_label()` (método de `mailbox.BabylMessage`), 1120
- `remove_option()` (método de `configparser.ConfigParser`), 533
- `remove_option()` (método de `optparse.OptionParser`), 1919
- `remove_pyc()` (método de `msilib.Directory`), 1865
- `remove_reader()` (método de `asyncio.loop`), 907
- `remove_section()` (método de `configparser.ConfigParser`), 533
- `remove_sequence()` (método de `mailbox.MHMessage`), 1119
- `remove_signal_handler()` (método de `asyncio.loop`), 909
- `remove_writer()` (método de `asyncio.loop`), 907
- `removeAttribute()` (método de `xml.dom.Element`), 1167
- `removeAttributeNode()` (método de `xml.dom.Element`), 1167
- `removeAttributeNS()` (método de `xml.dom.Element`), 1167
- `removeChild()` (método de `xml.dom.Node`), 1164
- `removedirs()` (en el módulo `os`), 583
- `removeFilter()` (método de `logging.Handler`), 670
- `removeFilter()` (método de `logging.Logger`), 668
- `removeHandler()` (en el módulo `unittest`), 1547
- `removeHandler()` (método de `logging.Logger`), 668
- `removeResult()` (en el módulo `unittest`), 1547
- `removexattr()` (en el módulo `os`), 595
- `rename()` (en el módulo `os`), 583
- `rename()` (método de `ftplib.FTP`), 1261
- `rename()` (método de `imaplib.IMAP4`), 1269
- `rename()` (método de `pathlib.Path`), 386
- `renames (2to3 fixer)`, 1611
- `renames()` (en el módulo `os`), 583
- `reopenIfNeeded()` (método de `logging.handlers.WatchedFileHandler`), 694
- `reorganize()` (método de `dbm.gnu.gdbm`), 444
- `repeat()` (en el módulo `itertools`), 351
- `repeat()` (en el módulo `timeit`), 1655
- `repeat()` (método de `timeit.Timer`), 1656
- `--repeat=N`
 - `timeit` command line option, 1657
- `repetition`
 - operation, 38
- `replace()` (en el módulo `dataclasses`), 1726
- `replace()` (en el módulo `os`), 583
- `replace()` (método de `bytearray`), 58
- `replace()` (método de `bytes`), 58
- `replace()` (método de `curses.panel.Panel`), 726
- `replace()` (método de `datetime.date`), 184
- `replace()` (método de `datetime.datetime`), 191
- `replace()` (método de `datetime.time`), 199
- `replace()` (método de `inspect.Parameter`), 1765
- `replace()` (método de `inspect.Signature`), 1764
- `replace()` (método de `pathlib.Path`), 386
- `replace()` (método de `str`), 49
- `replace()` (método de `types.CodeType`), 255
- `replace_errors()` (en el módulo `codecs`), 163
- `replace_header()` (método de `email.message.EmailMessage`), 1041
- `replace_header()` (método de `email.message.Message`), 1080
- `replace_history_item()` (en el módulo `readline`), 148
- `replace_whitespace` (atributo de `textwrap.TextWrapper`), 142
- `replaceChild()` (método de `xml.dom.Node`), 1165
- `ReplacePackage()` (en el módulo `modulefinder`), 1786
- `--report`
 - `trace` command line option, 1660
- `report()` (método de `filecmp.dircmp`), 402
- `report()` (método de `modulefinder.ModuleFinder`), 1786
- `REPORT_CDIF` (en el módulo `doctest`), 1505
- `report_failure()` (método de `doctest.DocTestRunner`), 1514
- `report_full_closure()` (método de `filecmp.dircmp`), 402
- `REPORT_NDIFF` (en el módulo `doctest`), 1505
- `REPORT_ONLY_FIRST_FAILURE` (en el módulo `doctest`), 1505
- `report_partial_closure()` (método de `filecmp.dircmp`), 402
- `report_start()` (método de `doctest.DocTestRunner`), 1514

- `report_success()` (método de `doc-test.DocTestRunner`), 1514
- `REPORT_UDIFF` (en el módulo `doctest`), 1505
- `report_unexpected_exception()` (método de `doctest.DocTestRunner`), 1515
- `REPORTING_FLAGS` (en el módulo `doctest`), 1505
- `repr` (2to3 fixer), 1611
- `Repr` (clase en `reprlib`), 264
- `repr()` (en el módulo `reprlib`), 265
- `repr()` (función incorporada), 21
- `repr()` (método de `reprlib.Repr`), 265
- `repr1()` (método de `reprlib.Repr`), 266
- `reprlib` (módulo), 264
- `Request` (clase en `urllib.request`), 1222
- `request()` (método de `http.client.HTTPConnection`), 1253
- `request_queue_size` (atributo de `socketserver.BaseServer`), 1297
- `request_rate()` (método de `urllib.robotparser.RobotFileParser`), 1247
- `request_uri()` (en el módulo `wsgiref.util`), 1210
- `request_version` (atributo de `http.server.BaseHTTPRequestHandler`), 1303
- `RequestHandlerClass` (atributo de `socketserver.BaseServer`), 1297
- `requestline` (atributo de `http.server.BaseHTTPRequestHandler`), 1303
- `requires()` (en el módulo `test.support`), 1616
- `requires_bz2()` (en el módulo `test.support`), 1621
- `requires_docstrings()` (en el módulo `test.support`), 1621
- `requires_freebsd_version()` (en el módulo `test.support`), 1621
- `requires_gzip()` (en el módulo `test.support`), 1621
- `requires_IEEE_754()` (en el módulo `test.support`), 1621
- `requires_linux_version()` (en el módulo `test.support`), 1621
- `requires_lzma()` (en el módulo `test.support`), 1621
- `requires_mac_version()` (en el módulo `test.support`), 1621
- `requires_resource()` (en el módulo `test.support`), 1621
- `requires_zlib()` (en el módulo `test.support`), 1621
- `reserved` (atributo de `zipfile.ZipInfo`), 493
- `RESERVED_FUTURE` (en el módulo `uuid`), 1293
- `RESERVED_MICROSOFT` (en el módulo `uuid`), 1293
- `RESERVED_NCS` (en el módulo `uuid`), 1293
- `reset()` (en el módulo `turtle`), 1406
- `reset()` (método de `bdb.Bdb`), 1634
- `reset()` (método de `codecs.IncrementalDecoder`), 165
- `reset()` (método de `codecs.IncrementalEncoder`), 164
- `reset()` (método de `codecs.StreamReader`), 167
- `reset()` (método de `codecs.StreamWriter`), 166
- `reset()` (método de `contextvars.ContextVar`), 859
- `reset()` (método de `html.parser.HTMLParser`), 1137
- `reset()` (método de `ossaudiodev.oss_audio_device`), 1364
- `reset()` (método de `pipes.Template`), 1892
- `reset()` (método de `threading.Barrier`), 781
- `reset()` (método de `xdrlib.Packer`), 537
- `reset()` (método de `xdrlib.Unpacker`), 538
- `reset()` (método de `xml.dom.pulldom.DOMEEventStream`), 1177
- `reset()` (método de `xml.sax.xmlreader.IncrementalParser`), 1187
- `reset_mock()` (método de `unittest.mock.AsyncMock`), 1562
- `reset_mock()` (método de `unittest.mock.Mock`), 1553
- `reset_prog_mode()` (en el módulo `curses`), 710
- `reset_shell_mode()` (en el módulo `curses`), 710
- `resetbuffer()` (método de `code.InteractiveConsole`), 1779
- `resetlocale()` (en el módulo `locale`), 1380
- `resetscreen()` (en el módulo `turtle`), 1406
- `resetty()` (en el módulo `curses`), 710
- `resetwarnings()` (en el módulo `warnings`), 1720
- `resize()` (en el módulo `ctypes`), 762
- `resize()` (método de `curses.window`), 717
- `resize()` (método de `mmap.mmap`), 1034
- `resize_term()` (en el módulo `curses`), 710
- `resizemode()` (en el módulo `turtle`), 1400
- `resizeterm()` (en el módulo `curses`), 710
- `resolution` (atributo de `datetime.date`), 184
- `resolution` (atributo de `datetime.datetime`), 190
- `resolution` (atributo de `datetime.time`), 198
- `resolution` (atributo de `datetime.timedelta`), 180
- `resolve()` (método de `pathlib.Path`), 387
- `resolve_bases()` (en el módulo `types`), 254
- `resolve_name()` (en el módulo `importlib.util`), 1804
- `resolveEntity()` (método de `xml.sax.handler.EntityResolver`), 1183
- `Resource` (en el módulo `importlib.resources`), 1798
- `resource` (módulo), 1893
- `resource_path()` (método de `importlib.abc.ResourceReader`), 1795
- `ResourceDenied`, 1614
- `ResourceLoader` (clase en `importlib.abc`), 1795
- `ResourceReader` (clase en `importlib.abc`), 1794
- `ResourceWarning`, 97
- `response` (atributo de `nnplib.NNTPError`), 1273
- `response()` (método de `imaplib.IMAP4`), 1269
- `ResponseNotReady`, 1252
- `responses` (atributo de `http.server.BaseHTTPRequestHandler`), 1304
- `responses` (en el módulo `http.client`), 1252
- `restart` (`pdb` command), 1645
- `restore()` (en el módulo `difflib`), 132
- `restype` (atributo de `ctypes._FuncPtr`), 757
- `result()` (método de `asyncio.Future`), 919
- `result()` (método de `asyncio.Task`), 876
- `result()` (método de `concurrent.futures.Future`), 834
- `results()` (método de `trace.Trace`), 1661

`resume_reading()` (método de `asyncio.ReadTransport`), 924
`resume_writing()` (método de `asyncio.BaseProtocol`), 927
`retr()` (método de `poplib.POP3`), 1264
`retrbinary()` (método de `ftplib.FTP`), 1260
`retrieve()` (método de `urllib.request.URLopener`), 1236
`retrlines()` (método de `ftplib.FTP`), 1260
`return` (*pdb* command), 1644
`return_annotation` (atributo de `inspect.Signature`), 1763
`return_ok()` (método de `http.cookiejar.CookiePolicy`), 1315
`return_value` (atributo de `unittest.mock.Mock`), 1554
`RETURN_VALUE` (opcode), 1848
`returncode` (atributo de `asyncio.subprocess.Process`), 892
`returncode` (atributo de `subprocess.CallProcessError`), 838
`returncode` (atributo de `subprocess.CompletedProcess`), 837
`returncode` (atributo de `subprocess.Popen`), 845
`retval` (*pdb* command), 1646
`reverse()` (en el módulo `audioop`), 1349
`reverse()` (método de `array.array`), 245
`reverse()` (método de `collections.deque`), 223
`reverse()` (*sequence method*), 40
`reverse_order()` (método de `pstats.Stats`), 1651
`reverse_pointer` (atributo de `ipaddress.IPv4Address`), 1335
`reverse_pointer` (atributo de `ipaddress.IPv6Address`), 1336
`reversed()` (función incorporada), 21
`Reversible` (clase en `collections.abc`), 235
`Reversible` (clase en `typing`), 1485
`revert()` (método de `http.cookiejar.FileCookieJar`), 1314
`rewind()` (método de `aifc.aifc`), 1351
`rewind()` (método de `sunau.AU_read`), 1354
`rewind()` (método de `wave.Wave_read`), 1356
RFC
RFC 821, 1278, 1280
RFC 822, 627, 1069, 1086, 1254, 1281, 1282, 1284, 1371
RFC 854, 1288, 1289
RFC 959, 1257
RFC 977, 1271
RFC 1014, 536
RFC 1123, 627
RFC 1321, 543
RFC 1422, 997, 1006
RFC 1521, 1130, 1133
RFC 1522, 1133
RFC 1524, 1105, 1106
RFC 1730, 1265
RFC 1738, 1246
RFC 1750, 977
RFC 1766, 1379, 1380
RFC 1808, 1238, 1246
RFC 1832, 536
RFC 1869, 1278, 1280
RFC 1870, 1285, 1288
RFC 1939, 1262
RFC 2045, 1037, 1041, 1042, 1064, 1065, 1080, 1081, 1087, 1127, 1129
RFC 2045#section-6.8, 1324
RFC 2046, 1037, 1068, 1087
RFC 2047, 1037, 1057, 1062, 1063, 1087, 1088, 1093
RFC 2060, 1270
RFC 2068, 1308
RFC 2104, 553
RFC 2109, 13081313, 1317, 1318
RFC 2183, 1037, 1043, 1082
RFC 2231, 1037, 1041, 1042, 1080, 1081, 1087, 1094
RFC 2295, 1250
RFC 2342, 1268
RFC 2368, 1246
RFC 2373, 1335, 1336
RFC 2396, 1241, 1244, 1246
RFC 2397, 1231
RFC 2449, 1263
RFC 2518, 1249
RFC 2595, 1262, 1264
RFC 2616, 1211, 1214, 1228, 1229, 1236, 1247
RFC 2732, 1246
RFC 2774, 1250
RFC 2818, 977
RFC 2821, 1037
RFC 2822, 627, 1078, 10861088, 10921094, 1115, 1251, 1303
RFC 2964, 1313
RFC 2965, 1222, 1225, 13111313, 13151319
RFC 2980, 1271, 1277
RFC 3056, 1337
RFC 3171, 1335
RFC 3229, 1249
RFC 3280, 987
RFC 3330, 1336
RFC 3454, 146
RFC 3490, 173175
RFC 3490#section-3.1, 175
RFC 3492, 173, 174
RFC 3493, 973
RFC 3501, 1270
RFC 3542, 961
RFC 3548, 1127, 1128, 1131
RFC 3659, 1261
RFC 3879, 1337
RFC 3927, 1336
RFC 3977, 1271, 12731275, 1277
RFC 3986, 1240, 1242, 1244, 1246, 1303
RFC 4086, 1006
RFC 4122, 12911293

- RFC 4180, 511
 RFC 4193, 1337
 RFC 4217, 1258
 RFC 4291, 1336
 RFC 4380, 1337
 RFC 4627, 1096, 1104
 RFC 4642, 1273
 RFC 4918, 1249, 1250
 RFC 4954, 1281
 RFC 5161, 1267
 RFC 5246, 984, 1006
 RFC 5280, 977, 978, 1006
 RFC 5321, 1066, 1285, 1286
 RFC 5322, 1037, 1038, 1048, 1051, 1052, 1055, 1057, 1058, 1060, 1061, 1063, 1066, 1075, 1283
 RFC 5424, 698
 RFC 5735, 1336
 RFC 5842, 1249, 1250
 RFC 5891, 174
 RFC 5895, 174
 RFC 5929, 988
 RFC 6066, 983, 993, 1006
 RFC 6125, 977
 RFC 6152, 1285
 RFC 6531, 1039, 1057, 1279, 1285, 1286
 RFC 6532, 1037, 1038, 1048, 1057
 RFC 6585, 1250
 RFC 6855, 1267
 RFC 6856, 1264
 RFC 7159, 1102, 1104
 RFC 7230, 1222, 1254
 RFC 7231, 1249, 1250
 RFC 7232, 1249
 RFC 7233, 1249
 RFC 7235, 1249
 RFC 7238, 1249
 RFC 7301, 983, 992
 RFC 7525, 1006
 RFC 7540, 1250
 RFC 7693, 546
 RFC 7725, 1250
 RFC 7914, 546
 RFC 8305, 902
 rfc2109 (atributo de `http.cookiejar.Cookie`), 1318
 rfc2109_as_netscape (atributo de `http.cookiejar.DefaultCookiePolicy`), 1317
 rfc2965 (atributo de `http.cookiejar.CookiePolicy`), 1316
 RFC_4122 (en el módulo `uuid`), 1293
 rfile (atributo de `http.server.BaseHTTPRequestHandler`), 1303
 rfind() (método de `bytearray`), 58
 rfind() (método de `bytes`), 58
 rfind() (método de `mmap.mmap`), 1034
 rfind() (método de `str`), 49
 rgb_to_hls() (en el módulo `colorsys`), 1359
 rgb_to_hsv() (en el módulo `colorsys`), 1359
 rgb_to_yiq() (en el módulo `colorsys`), 1359
 rglob() (método de `pathlib.Path`), 387
 right (atributo de `filecmp.dircmp`), 402
 right() (en el módulo `turtle`), 1390
 right_list (atributo de `filecmp.dircmp`), 402
 right_only (atributo de `filecmp.dircmp`), 402
 RIGHTSHIFT (en el módulo `token`), 1828
 RIGHTSHIFTEQUAL (en el módulo `token`), 1829
 rindex() (método de `bytearray`), 59
 rindex() (método de `bytes`), 59
 rindex() (método de `str`), 49
 rjust() (método de `bytearray`), 60
 rjust() (método de `bytes`), 60
 rjust() (método de `str`), 49
 rlcompleter (módulo), 151
 rlecode_hqx() (en el módulo `binascii`), 1131
 rledecode_hqx() (en el módulo `binascii`), 1131
 RLIM_INFINITY (en el módulo `resource`), 1893
 RLIMIT_AS (en el módulo `resource`), 1894
 RLIMIT_CORE (en el módulo `resource`), 1894
 RLIMIT_CPU (en el módulo `resource`), 1894
 RLIMIT_DATA (en el módulo `resource`), 1894
 RLIMIT_FSIZE (en el módulo `resource`), 1894
 RLIMIT_MEMLOCK (en el módulo `resource`), 1894
 RLIMIT_MSGQUEUE (en el módulo `resource`), 1894
 RLIMIT_NICE (en el módulo `resource`), 1894
 RLIMIT_NOFILE (en el módulo `resource`), 1894
 RLIMIT_NPROC (en el módulo `resource`), 1894
 RLIMIT_NPTS (en el módulo `resource`), 1895
 RLIMIT_OFILE (en el módulo `resource`), 1894
 RLIMIT_RSS (en el módulo `resource`), 1894
 RLIMIT_RTPRIO (en el módulo `resource`), 1895
 RLIMIT_RTTIME (en el módulo `resource`), 1895
 RLIMIT_SBSIZE (en el módulo `resource`), 1895
 RLIMIT_SIGPENDING (en el módulo `resource`), 1895
 RLIMIT_STACK (en el módulo `resource`), 1894
 RLIMIT_SWAP (en el módulo `resource`), 1895
 RLIMIT_VMEM (en el módulo `resource`), 1894
 RLock (clase en `multiprocessing`), 798
 RLock (clase en `threading`), 775
 RLock() (método de `multiprocessing.managers.SyncManager`), 804
 rmd() (método de `ftplib.FTP`), 1261
 rmdir() (en el módulo `os`), 583
 rmdir() (en el módulo `test.support`), 1616
 rmdir() (método de `pathlib.Path`), 387
 RMFF, 1358
 rms() (en el módulo `audioop`), 1349
 rmtree() (en el módulo `shutil`), 413
 rmtree() (en el módulo `test.support`), 1616
 RobotFileParser (clase en `urllib.robotparser`), 1247
 robots.txt, 1247
 rollback() (método de `sqlite3.Connection`), 450
 ROT_FOUR (opcode), 1845
 ROT_THREE (opcode), 1845
 ROT_TWO (opcode), 1845
 rotate() (método de `collections.deque`), 223

- `rotate()` (método de `decimal.Context`), 314
- `rotate()` (método de `decimal.Decimal`), 308
- `rotate()` (método de `logging.handlers.BaseRotatingHandler`), 694
- `RotatingFileHandler` (clase en `logging.handlers`), 695
- `rotation_filename()` (método de `logging.handlers.BaseRotatingHandler`), 694
- `rotator` (atributo de `logging.handlers.BaseRotatingHandler`), 694
- `round()` (función incorporada), 21
- `ROUND_05UP` (en el módulo `decimal`), 315
- `ROUND_CEILING` (en el módulo `decimal`), 315
- `ROUND_DOWN` (en el módulo `decimal`), 315
- `ROUND_FLOOR` (en el módulo `decimal`), 315
- `ROUND_HALF_DOWN` (en el módulo `decimal`), 315
- `ROUND_HALF_EVEN` (en el módulo `decimal`), 315
- `ROUND_HALF_UP` (en el módulo `decimal`), 315
- `ROUND_UP` (en el módulo `decimal`), 315
- `Rounded` (clase en `decimal`), 316
- `Row` (clase en `sqlite3`), 459
- `row_factory` (atributo de `sqlite3.Connection`), 454
- `rowcount` (atributo de `sqlite3.Cursor`), 458
- `RPAR` (en el módulo `token`), 1827
- `rpartition()` (método de `bytearray`), 59
- `rpartition()` (método de `bytes`), 59
- `rpartition()` (método de `str`), 49
- `rpc_paths` (atributo de `xmlrpc.server.SimpleXMLRPCRequestHandler`), 1329
- `rpop()` (método de `poplib.POP3`), 1264
- `rset()` (método de `poplib.POP3`), 1264
- `rshift()` (en el módulo `operator`), 367
- `rsplit()` (método de `bytearray`), 60
- `rsplit()` (método de `bytes`), 60
- `rsplit()` (método de `str`), 49
- `RSQB` (en el módulo `token`), 1827
- `rstrip()` (método de `bytearray`), 60
- `rstrip()` (método de `bytes`), 60
- `rstrip()` (método de `str`), 49
- `rt()` (en el módulo `turtle`), 1390
- `RTLD_DEEPBIND` (en el módulo `os`), 609
- `RTLD_GLOBAL` (en el módulo `os`), 609
- `RTLD_LAZY` (en el módulo `os`), 609
- `RTLD_LOCAL` (en el módulo `os`), 609
- `RTLD_NODELETE` (en el módulo `os`), 609
- `RTLD_NOLOAD` (en el módulo `os`), 609
- `RTLD_NOW` (en el módulo `os`), 609
- `ruler` (atributo de `cmd.Cmd`), 1420
- `run` (`pdb` command), 1645
- `Run script`, 1466
- `run()` (en el módulo `asyncio`), 869
- `run()` (en el módulo `pdb`), 1641
- `run()` (en el módulo `profile`), 1649
- `run()` (en el módulo `subprocess`), 836
- `run()` (método de `bdb.Bdb`), 1637
- `run()` (método de `contextvars.Context`), 860
- `run()` (método de `doctest.DocTestRunner`), 1515
- `run()` (método de `multiprocessing.Process`), 789
- `run()` (método de `pdb.Pdb`), 1642
- `run()` (método de `profile.Profile`), 1650
- `run()` (método de `sched.scheduler`), 855
- `run()` (método de `test.support.BasicTestRunner`), 1627
- `run()` (método de `threading.Thread`), 772
- `run()` (método de `trace.Trace`), 1661
- `run()` (método de `unittest.IsolatedAsyncioTestCase`), 1536
- `run()` (método de `unittest.TestCase`), 1529
- `run()` (método de `unittest.TestSuite`), 1538
- `run()` (método de `unittest.TextTestRunner`), 1544
- `run()` (método de `wsgiref.handlers.BaseHandler`), 1215
- `run_coroutine_threadsafe()` (en el módulo `asyncio`), 874
- `run_docstring_examples()` (en el módulo `doctest`), 1509
- `run_doctest()` (en el módulo `test.support`), 1617
- `run_forever()` (método de `asyncio.loop`), 899
- `run_in_executor()` (método de `asyncio.loop`), 910
- `run_in_subinterp()` (en el módulo `test.support`), 1625
- `run_module()` (en el módulo `runpy`), 1787
- `run_path()` (en el módulo `runpy`), 1788
- `run_python_until_end()` (en el módulo `test.support.script_helper`), 1627
- `run_script()` (método de `modulefinder.ModuleFinder`), 1786
- `run_unittest()` (en el módulo `test.support`), 1617
- `run_until_complete()` (método de `asyncio.loop`), 899
- `run_with_locale()` (en el módulo `test.support`), 1621
- `run_with_tz()` (en el módulo `test.support`), 1621
- `runcall()` (en el módulo `pdb`), 1641
- `runcall()` (método de `bdb.Bdb`), 1637
- `runcall()` (método de `pdb.Pdb`), 1642
- `runcall()` (método de `profile.Profile`), 1650
- `runcode()` (método de `code.InteractiveInterpreter`), 1778
- `runtctx()` (en el módulo `profile`), 1649
- `runtctx()` (método de `bdb.Bdb`), 1637
- `runtctx()` (método de `profile.Profile`), 1650
- `runtctx()` (método de `trace.Trace`), 1661
- `runeval()` (en el módulo `pdb`), 1641
- `runeval()` (método de `bdb.Bdb`), 1637
- `runeval()` (método de `pdb.Pdb`), 1642
- `runfunc()` (método de `trace.Trace`), 1661
- `running()` (método de `concurrent.futures.Future`), 834
- `runpy` (módulo), 1787
- `runsource()` (método de `code.InteractiveInterpreter`), 1778
- `runtime_checkable()` (en el módulo `typing`), 1493
- `RuntimeError`, 92
- `RuntimeWarning`, 96
- `RUSAGE_BOTH` (en el módulo `resource`), 1896
- `RUSAGE_CHILDREN` (en el módulo `resource`), 1896
- `RUSAGE_SELF` (en el módulo `resource`), 1896

RUSAGE_THREAD (en el módulo *resource*), 1896
 ruta de importación, 1941
 RWF_DSYNC (en el módulo *os*), 572
 RWF_HIPRI (en el módulo *os*), 571
 RWF_NOWAIT (en el módulo *os*), 571
 RWF_SYNC (en el módulo *os*), 572

S

-s
 trace command line option, 1660
 unittest-discover command line option, 1522
 S (en el módulo *re*), 117
 -s S
 timeit command line option, 1657
 S_ENFMT (en el módulo *stat*), 400
 S_IEXEC (en el módulo *stat*), 400
 S_IFBLK (en el módulo *stat*), 399
 S_IFCHR (en el módulo *stat*), 399
 S_IFDIR (en el módulo *stat*), 399
 S_IFDOOR (en el módulo *stat*), 399
 S_IFIFO (en el módulo *stat*), 399
 S_IFLNK (en el módulo *stat*), 398
 S_IFMT () (en el módulo *stat*), 397
 S_IFPORT (en el módulo *stat*), 399
 S_IFREG (en el módulo *stat*), 398
 S_IFSOCK (en el módulo *stat*), 398
 S_IFWHT (en el módulo *stat*), 399
 S_IMODE () (en el módulo *stat*), 397
 S_IREAD (en el módulo *stat*), 400
 S_IRGRP (en el módulo *stat*), 399
 S_IROTH (en el módulo *stat*), 400
 S_IRUSR (en el módulo *stat*), 399
 S_IRWXG (en el módulo *stat*), 399
 S_IRWXO (en el módulo *stat*), 400
 S_IRWXU (en el módulo *stat*), 399
 S_ISBLK () (en el módulo *stat*), 396
 S_ISCHR () (en el módulo *stat*), 396
 S_ISDIR () (en el módulo *stat*), 396
 S_ISDOOR () (en el módulo *stat*), 397
 S_ISFIFO () (en el módulo *stat*), 397
 S_ISGID (en el módulo *stat*), 399
 S_ISLNK () (en el módulo *stat*), 397
 S_ISPORT () (en el módulo *stat*), 397
 S_ISREG () (en el módulo *stat*), 397
 S_ISSOCK () (en el módulo *stat*), 397
 S_ISUID (en el módulo *stat*), 399
 S_ISVTX (en el módulo *stat*), 399
 S_ISWHT () (en el módulo *stat*), 397
 S_IWGRP (en el módulo *stat*), 399
 S_IWOTH (en el módulo *stat*), 400
 S_IWRITE (en el módulo *stat*), 400
 S_IWUSR (en el módulo *stat*), 399
 S_IXGRP (en el módulo *stat*), 400
 S_IXOTH (en el módulo *stat*), 400
 S_IXUSR (en el módulo *stat*), 399
 safe (atributo de *uuid.SafeUUID*), 1291

safe_substitute () (método de *string.Template*), 108
 SafeChildWatcher (clase en *asyncio*), 937
 saferepr () (en el módulo *pprint*), 261
 SafeUUID (clase en *uuid*), 1291
 saltos de líneas universales, 1947
 same_files (atributo de *filecmp.dircmp*), 403
 same_quantum () (método de *decimal.Context*), 314
 same_quantum () (método de *decimal.Decimal*), 308
 samefile () (en el módulo *os.path*), 393
 samefile () (método de *pathlib.Path*), 387
 SameFileError, 411
 sameopenfile () (en el módulo *os.path*), 393
 samestat () (en el módulo *os.path*), 393
 sample () (en el módulo *random*), 328
 samples () (método de *statistics.NormalDist*), 339
 save () (método de *http.cookiejar.FileCookieJar*), 1314
 SAVEDCWD (en el módulo *test.support*), 1615
 SaveKey () (en el módulo *winreg*), 1873
 SaveSignals (clase en *test.support*), 1626
 savetty () (en el módulo *curses*), 710
 SAX2DOM (clase en *xml.dom.pulldom*), 1176
 SAXException, 1178
 SAXNotRecognizedException, 1178
 SAXNotSupportedException, 1179
 SAXParseException, 1178
 scaleb () (método de *decimal.Context*), 314
 scaleb () (método de *decimal.Decimal*), 308
 scandir () (en el módulo *os*), 584
 scanf (), 125
 sched (módulo), 854
 SCHED_BATCH (en el módulo *os*), 606
 SCHED_FIFO (en el módulo *os*), 606
 sched_get_priority_max () (en el módulo *os*), 606
 sched_get_priority_min () (en el módulo *os*), 606
 sched_getaffinity () (en el módulo *os*), 607
 sched_getparam () (en el módulo *os*), 607
 sched_getscheduler () (en el módulo *os*), 607
 SCHED_IDLE (en el módulo *os*), 606
 SCHED_OTHER (en el módulo *os*), 606
 sched_param (clase en *os*), 606
 sched_priority (atributo de *os.sched_param*), 606
 SCHED_RESET_ON_FORK (en el módulo *os*), 606
 SCHED_RR (en el módulo *os*), 606
 sched_rr_get_interval () (en el módulo *os*), 607
 sched_setaffinity () (en el módulo *os*), 607
 sched_setparam () (en el módulo *os*), 607
 sched_setscheduler () (en el módulo *os*), 607
 SCHED_SPORADIC (en el módulo *os*), 606
 sched_yield () (en el módulo *os*), 607
 scheduler (clase en *sched*), 854
 schema (en el módulo *msilib*), 1866
 Screen (clase en *turtle*), 1413
 screensize () (en el módulo *turtle*), 1406

- `script_from_examples()` (en el módulo `doctest`), 1516
- `scroll()` (método de `curses.window`), 717
- `ScrolledCanvas` (clase en `turtle`), 1413
- `scrollok()` (método de `curses.window`), 717
- `script()` (en el módulo `hashlib`), 546
- `seal()` (en el módulo `unittest.mock`), 1587
- `search`
 `path`, `module`, 410, 1703, 1773
- `search()` (en el módulo `re`), 117
- `search()` (método de `imaplib.IMAP4`), 1269
- `search()` (método de `re.Pattern`), 120
- `second` (atributo de `datetime.datetime`), 190
- `second` (atributo de `datetime.time`), 198
- `seconds since the epoch`, 622
- `secrets` (módulo), 555
- `SECTCRE` (atributo de `configparser.ConfigParser`), 528
- `sections()` (método de `configparser.ConfigParser`), 531
- `secuencia`, 1946
- `secure` (atributo de `http.cookiejar.Cookie`), 1318
- `secure hash algorithm`, SHA1, SHA224, SHA256, SHA384, SHA512, 543
- `Secure Sockets Layer`, 973
- `security`
 `CGI`, 1206
 `http.server`, 1308
- `see()` (método de `tkinter.ttk.Treeview`), 1454
- `seed()` (en el módulo `random`), 327
- `seek()` (método de `chunk.Chunk`), 1358
- `seek()` (método de `io.IOBase`), 614
- `seek()` (método de `io.TextIOBase`), 619
- `seek()` (método de `mmap.mmap`), 1034
- `SEEK_CUR` (en el módulo `os`), 569
- `SEEK_END` (en el módulo `os`), 569
- `SEEK_SET` (en el módulo `os`), 569
- `seekable()` (método de `io.IOBase`), 614
- `seen_greeting` (atributo de `smtpd.SMTPChannel`), 1287
- `Select` (clase en `tkinter.tix`), 1460
- `select` (módulo), 1007
- `select()` (en el módulo `select`), 1008
- `select()` (método de `imaplib.IMAP4`), 1269
- `select()` (método de `selectors.BaseSelector`), 1015
- `select()` (método de `tkinter.ttk.Notebook`), 1447
- `selected_alpn_protocol()` (método de `ssl.SSLSocket`), 988
- `selected_npn_protocol()` (método de `ssl.SSLSocket`), 988
- `selection()` (método de `tkinter.ttk.Treeview`), 1454
- `selection_add()` (método de `tkinter.ttk.Treeview`), 1454
- `selection_remove()` (método de `tkinter.ttk.Treeview`), 1454
- `selection_set()` (método de `tkinter.ttk.Treeview`), 1454
- `selection_toggle()` (método de `tkinter.ttk.Treeview`), 1455
- `selector` (atributo de `urllib.request.Request`), 1225
- `SelectorEventLoop` (clase en `asyncio`), 915
- `SelectorKey` (clase en `selectors`), 1014
- `selectors` (módulo), 1014
- `SelectSelector` (clase en `selectors`), 1016
- `Semaphore` (clase en `asyncio`), 888
- `Semaphore` (clase en `multiprocessing`), 799
- `Semaphore` (clase en `threading`), 778
- `Semaphore()` (método de `multiprocessing.managers.SyncManager`), 804
- `semaphores`, `binary`, 862
- `SEMI` (en el módulo `token`), 1827
- `send()` (método de `asyncore.dispatcher`), 1019
- `send()` (método de `http.client.HTTPConnection`), 1255
- `send()` (método de `imaplib.IMAP4`), 1269
- `send()` (método de `logging.handlers.DatagramHandler`), 698
- `send()` (método de `logging.handlers.SocketHandler`), 697
- `send()` (método de `multiprocessing.connection.Connection`), 796
- `send()` (método de `socket.socket`), 966
- `send_bytes()` (método de `multiprocessing.connection.Connection`), 796
- `send_error()` (método de `http.server.BaseHTTPRequestHandler`), 1304
- `send_flowling_data()` (método de `formatter.writer`), 1860
- `send_header()` (método de `http.server.BaseHTTPRequestHandler`), 1304
- `send_hor_rule()` (método de `formatter.writer`), 1860
- `send_label_data()` (método de `formatter.writer`), 1860
- `send_line_break()` (método de `formatter.writer`), 1860
- `send_literal_data()` (método de `formatter.writer`), 1860
- `send_message()` (método de `smtpplib.SMTP`), 1283
- `send_paragraph()` (método de `formatter.writer`), 1860
- `send_response()` (método de `http.server.BaseHTTPRequestHandler`), 1304
- `send_response_only()` (método de `http.server.BaseHTTPRequestHandler`), 1305
- `send_signal()` (método de `asyncio.subprocess.Process`), 891
- `send_signal()` (método de `asyncio.SubprocessTransport`), 926
- `send_signal()` (método de `subprocess.Popen`), 845
- `sendall()` (método de `socket.socket`), 967
- `sendcmd()` (método de `ftplib.FTP`), 1260
- `sendfile()` (en el módulo `os`), 573
- `sendfile()` (método de `asyncio.loop`), 906

- `sendfile()` (método de `socket.socket`), 968
- `sendfile()` (método de `wsgi-ref.handlers.BaseHandler`), 1217
- `SendfileNotAvailableError`, 896
- `sendmail()` (método de `smtplib.SMTP`), 1282
- `sendmsg()` (método de `socket.socket`), 967
- `sendmsg_afalg()` (método de `socket.socket`), 967
- `sendto()` (método de `asyncio.DatagramTransport`), 925
- `sendto()` (método de `socket.socket`), 967
- `sentencia`, 1946
- `assert`, 90
 - `del`, 40, 77
 - `except`, 89
 - `if`, 29
 - `import`, 25, 1773, 1928
 - `raise`, 89
 - `try`, 89
 - `while`, 29
- `sentinel` (atributo de `multiprocessing.Process`), 790
- `sentinel` (en el módulo `unittest.mock`), 1579
- `sep` (en el módulo `os`), 608
- `sequence`
- iteration, 37
 - objeto, 38
 - types, immutable, 39
 - types, mutable, 40
 - types, operations on, 38, 40
- `Sequence` (clase en `collections.abc`), 235
- `Sequence` (clase en `typing`), 1486
- `sequence` (en el módulo `msilib`), 1866
- `sequence2st()` (en el módulo `parser`), 1814
- `SequenceMatcher` (clase en `difflib`), 134
- `serializing`
- objects, 421
- `serve_forever()` (método de `asyncio.Server`), 914
- `serve_forever()` (método de `socketserver.BaseServer`), 1296
- `server`
- WWW, 1202, 1302
- `server` (atributo de `http.server.BaseHTTPRequestHandler`), 1303
- `Server` (clase en `asyncio`), 914
- `server_activate()` (método de `socketserver.BaseServer`), 1298
- `server_address` (atributo de `socketserver.BaseServer`), 1297
- `server_bind()` (método de `socketserver.BaseServer`), 1298
- `server_close()` (método de `socketserver.BaseServer`), 1297
- `server_hostname` (atributo de `ssl.SSLSocket`), 989
- `server_side` (atributo de `ssl.SSLSocket`), 989
- `server_software` (atributo de `wsgi-ref.handlers.BaseHandler`), 1216
- `server_version` (atributo de `http.server.BaseHTTPRequestHandler`), 1303
- `server_version` (atributo de `http.server.SimpleHTTPRequestHandler`), 1306
- `ServerProxy` (clase en `xmlrpc.client`), 1320
- `service_actions()` (método de `socketserver.BaseServer`), 1296
- `session` (atributo de `ssl.SSLSocket`), 989
- `session_reused` (atributo de `ssl.SSLSocket`), 989
- `session_stats()` (método de `ssl.SSLContext`), 995
- `set`
- objeto, 75
- `Set` (clase en `collections.abc`), 235
- `Set` (clase en `typing`), 1487
- `set` (clase incorporada), 75
- `Set Breakpoint`, 1467
- `set comprehension`, 1946
- `set()` (método de `asyncio.Event`), 886
- `set()` (método de `configparser.ConfigParser`), 533
- `set()` (método de `configparser.RawConfigParser`), 534
- `set()` (método de `contextvars.ContextVar`), 859
- `set()` (método de `http.cookies.Morsel`), 1310
- `set()` (método de `ossaudiodev.oss_mixer_device`), 1365
- `set()` (método de `test.support.EnvironmentVarGuard`), 1626
- `set()` (método de `threading.Event`), 779
- `set()` (método de `tkinter.ttk.Combobox`), 1444
- `set()` (método de `tkinter.ttk.Spinbox`), 1445
- `set()` (método de `tkinter.ttk.Treeview`), 1455
- `set()` (método de `xml.etree.ElementTree.Element`), 1154
- `SET_ADD` (opcode), 1848
- `set_allowed_domains()` (método de `http.cookiejar.DefaultCookiePolicy`), 1317
- `set_alpn_protocols()` (método de `ssl.SSLContext`), 992
- `set_app()` (método de `wsgi-ref.simple_server.WSGIServer`), 1213
- `set_asyncgen_hooks()` (en el módulo `sys`), 1707
- `set_authorizer()` (método de `sqlite3.Connection`), 452
- `set_auto_history()` (en el módulo `readline`), 149
- `set_blocked_domains()` (método de `http.cookiejar.DefaultCookiePolicy`), 1316
- `set_blocking()` (en el módulo `os`), 573
- `set_boundary()` (método de `email.message.EmailMessage`), 1042
- `set_boundary()` (método de `email.message.Message`), 1082
- `set_break()` (método de `bdb.Bdb`), 1636
- `set_charset()` (método de `email.message.Message`), 1078
- `set_child_watcher()` (en el módulo `asyncio`), 936
- `set_child_watcher()` (método de `asyncio.AbstractEventLoopPolicy`), 935
- `set_children()` (método de `tkinter.ttk.Treeview`), 1452
- `set_ciphers()` (método de `ssl.SSLContext`), 992

- `set_completer()` (en el módulo `readline`), 149
- `set_completer_delims()` (en el módulo `readline`), 150
- `set_completion_display_matches_hook()` (en el módulo `readline`), 150
- `set_content()` (en el módulo `email.contentmanager`), 1068
- `set_content()` (método de `email.contentmanager.ContentManager`), 1067
- `set_content()` (método de `email.message.EmailMessage`), 1044
- `set_continue()` (método de `bdb.Bdb`), 1636
- `set_cookie()` (método de `http.cookiejar.CookieJar`), 1313
- `set_cookie_if_ok()` (método de `http.cookiejar.CookieJar`), 1313
- `set_coroutine_origin_tracking_depth()` (en el módulo `sys`), 1707
- `set_current()` (método de `msilib.Feature`), 1865
- `set_data()` (método de `importlib.abc.SourceLoader`), 1797
- `set_data()` (método de `importlib.machinery.SourceFileLoader`), 1801
- `set_date()` (método de `mailbox.MaildirMessage`), 1116
- `set_debug()` (en el módulo `gc`), 1755
- `set_debug()` (método de `asyncio.loop`), 912
- `set_debuglevel()` (método de `ftplib.FTP`), 1259
- `set_debuglevel()` (método de `http.client.HTTPConnection`), 1253
- `set_debuglevel()` (método de `nnplib.NNTP`), 1277
- `set_debuglevel()` (método de `poplib.POP3`), 1263
- `set_debuglevel()` (método de `smtpplib.SMTP`), 1280
- `set_debuglevel()` (método de `telnetlib.Telnet`), 1289
- `set_default_executor()` (método de `asyncio.loop`), 911
- `set_default_type()` (método de `email.message.EmailMessage`), 1042
- `set_default_type()` (método de `email.message.Message`), 1080
- `set_default_verify_paths()` (método de `ssl.SSLContext`), 992
- `set_defaults()` (método de `argparse.ArgumentParser`), 659
- `set_defaults()` (método de `optparse.OptionParser`), 1920
- `set_ecdh_curve()` (método de `ssl.SSLContext`), 993
- `set_errno()` (en el módulo `ctypes`), 762
- `set_event_loop()` (en el módulo `asyncio`), 897
- `set_event_loop()` (método de `asyncio.AbstractEventLoopPolicy`), 935
- `set_event_loop_policy()` (en el módulo `asyncio`), 935
- `set_exception()` (método de `asyncio.Future`), 919
- `set_exception()` (método de `concurrent.futures.Future`), 835
- `set_exception_handler()` (método de `asyncio.loop`), 911
- `set_executable()` (en el módulo `multiprocessing`), 795
- `set_flags()` (método de `mailbox.MaildirMessage`), 1116
- `set_flags()` (método de `mailbox.mboxMessage`), 1117
- `set_flags()` (método de `mailbox.MMDfMessage`), 1121
- `set_from()` (método de `mailbox.mboxMessage`), 1117
- `set_from()` (método de `mailbox.MMDfMessage`), 1121
- `set_handle_inheritable()` (en el módulo `os`), 575
- `set_history_length()` (en el módulo `readline`), 148
- `set_info()` (método de `mailbox.MaildirMessage`), 1116
- `set_inheritable()` (en el módulo `os`), 575
- `set_inheritable()` (método de `socket.socket`), 968
- `set_int_max_str_digits()` (en el módulo `sys`), 1705
- `set_labels()` (método de `mailbox.BabylMessage`), 1120
- `set_last_error()` (en el módulo `ctypes`), 762
- `set_literal(2to3 fixer)`, 1611
- `set_loader()` (en el módulo `importlib.util`), 1805
- `set_match_tests()` (en el módulo `test.support`), 1617
- `set_memlimit()` (en el módulo `test.support`), 1618
- `set_name()` (método de `asyncio.Task`), 877
- `set_next()` (método de `bdb.Bdb`), 1635
- `set_nonstandard_attr()` (método de `http.cookiejar.Cookie`), 1318
- `set_npn_protocols()` (método de `ssl.SSLContext`), 992
- `set_ok()` (método de `http.cookiejar.CookiePolicy`), 1315
- `set_option_negotiation_callback()` (método de `telnetlib.Telnet`), 1290
- `set_output_charset()` (método de `gettext.NullTranslations`), 1371
- `set_package()` (en el módulo `importlib.util`), 1805
- `set_param()` (método de `email.message.EmailMessage`), 1042
- `set_param()` (método de `email.message.Message`), 1081
- `set_pasv()` (método de `ftplib.FTP`), 1260
- `set_payload()` (método de `email.message.Message`), 1078
- `set_policy()` (método de `http.cookiejar.CookieJar`), 1313
- `set_position()` (método de `xdr.lib.Unpacker`), 538
- `set_pre_input_hook()` (en el módulo `readline`), 149

- `set_progress_handler()` (método de `sqli-te3.Connection`), 452
- `set_protocol()` (método de `asyncio.BaseTransport`), 923
- `set_proxy()` (método de `urllib.request.Request`), 1226
- `set_quit()` (método de `bdb.Bdb`), 1636
- `set_recsrc()` (método de `ossaudio-dev.oss_mixer_device`), 1365
- `set_result()` (método de `asyncio.Future`), 919
- `set_result()` (método de `concurrent.futures.Future`), 835
- `set_return()` (método de `bdb.Bdb`), 1636
- `set_running_or_notify_cancel()` (método de `concurrent.futures.Future`), 834
- `set_seq1()` (método de `difflib.SequenceMatcher`), 134
- `set_seq2()` (método de `difflib.SequenceMatcher`), 134
- `set_seqs()` (método de `difflib.SequenceMatcher`), 134
- `set_sequences()` (método de `mailbox.MH`), 1112
- `set_sequences()` (método de `mailbox.MHMessage`), 1119
- `set_server_documentation()` (método de `xmlrpc.server.DocCGIXMLRPCRequestHandler`), 1333
- `set_server_documentation()` (método de `xmlrpc.server.DocXMLRPCServer`), 1333
- `set_server_name()` (método de `xmlrpc.server.DocCGIXMLRPCRequestHandler`), 1333
- `set_server_name()` (método de `xmlrpc.server.DocXMLRPCServer`), 1332
- `set_server_title()` (método de `xmlrpc.server.DocCGIXMLRPCRequestHandler`), 1333
- `set_server_title()` (método de `xmlrpc.server.DocXMLRPCServer`), 1332
- `set_servername_callback` (atributo de `ssl.SSLContext`), 993
- `set_spacing()` (método de `formatter.formatter`), 1859
- `set_start_method()` (en el módulo `multiprocessing`), 795
- `set_startup_hook()` (en el módulo `readline`), 149
- `set_step()` (método de `bdb.Bdb`), 1635
- `set_subdir()` (método de `mailbox.MaildirMessage`), 1116
- `set_task_factory()` (método de `asyncio.loop`), 901
- `set_terminator()` (método de `asyncio.chat.async_chat`), 1022
- `set_threshold()` (en el módulo `gc`), 1755
- `set_trace()` (en el módulo `bdb`), 1637
- `set_trace()` (en el módulo `pdb`), 1641
- `set_trace()` (método de `bdb.Bdb`), 1636
- `set_trace()` (método de `pdb.Pdb`), 1642
- `set_trace_callback()` (método de `sqli-te3.Connection`), 453
- `set_tunnel()` (método de `http.client.HTTPConnection`), 1253
- `set_type()` (método de `email.message.Message`), 1081
- `set_unittest_reportflags()` (en el módulo `doctest`), 1511
- `set_unixfrom()` (método de `email.message.EmailMessage`), 1040
- `set_unixfrom()` (método de `email.message.Message`), 1077
- `set_until()` (método de `bdb.Bdb`), 1636
- `set_url()` (método de `urllib.robotparser.RobotFileParser`), 1247
- `set_usage()` (método de `optparse.OptionParser`), 1920
- `set_userptr()` (método de `curses.panel.Panel`), 727
- `set_visible()` (método de `mailbox.BabylMessage`), 1120
- `set_wakeup_fd()` (en el módulo `signal`), 1028
- `set_write_buffer_limits()` (método de `asyncio.WriteTransport`), 924
- `setacl()` (método de `imaplib.IMAP4`), 1269
- `setannotation()` (método de `imaplib.IMAP4`), 1269
- `setattr()` (función incorporada), 22
- `setAttribute()` (método de `xml.dom.Element`), 1167
- `setAttributeNode()` (método de `xml.dom.Element`), 1167
- `setAttributeNodeNS()` (método de `xml.dom.Element`), 1167
- `setAttributeNS()` (método de `xml.dom.Element`), 1167
- `SetBase()` (método de `xml.parsers.expat.xmlparser`), 1190
- `setblocking()` (método de `socket.socket`), 968
- `setByteStream()` (método de `xml.sax.xmlreader.InputSource`), 1187
- `setcbreak()` (en el módulo `tty`), 1888
- `setCharacterStream()` (método de `xml.sax.xmlreader.InputSource`), 1188
- `setcheckinterval()` (en el módulo `sys`), 1704
- `setcomptype()` (método de `aifc.aifc`), 1352
- `setcomptype()` (método de `sunau.AU_write`), 1355
- `setcomptype()` (método de `wave.Wave_write`), 1357
- `setContentHandler()` (método de `xml.sax.xmlreader.XMLReader`), 1186
- `setcontext()` (en el módulo `decimal`), 309
- `setDaemon()` (método de `threading.Thread`), 773
- `setdefault()` (método de `dict`), 80
- `setdefault()` (método de `http.cookies.Morsel`), 1310
- `setdefaulttimeout()` (en el módulo `socket`), 961
- `setdlopenflags()` (en el módulo `sys`), 1705
- `setDocumentLocator()` (método de `xml.sax.handler.ContentHandler`), 1181
- `setDTDHandler()` (método de `xml.sax.xmlreader.XMLReader`), 1186
- `setegid()` (en el módulo `os`), 564
- `setEncoding()` (método de `http.client.HTTPConnection`), 1253

xml.sax.xmlreader.InputSource), 1187
setEntityResolver() (método de *xml.sax.xmlreader.XMLReader*), 1186
setErrorHandler() (método de *xml.sax.xmlreader.XMLReader*), 1186
seteuid() (en el módulo *os*), 564
setFeature() (método de *xml.sax.xmlreader.XMLReader*), 1186
setfirstweekday() (en el módulo *calendar*), 215
setfmt() (método de *ossaudiodev.oss_audio_device*), 1363
setFormatter() (método de *logging.Handler*), 670
setframerate() (método de *aifc.aifc*), 1352
setframerate() (método de *sunau.AU_write*), 1355
setframerate() (método de *wave.Wave_write*), 1357
setgid() (en el módulo *os*), 564
setgroups() (en el módulo *os*), 564
seth() (en el módulo *turtle*), 1391
setheading() (en el módulo *turtle*), 1391
sethostname() (en el módulo *socket*), 961
SetInteger() (método de *msilib.Record*), 1864
setitem() (en el módulo *operator*), 367
setitimer() (en el módulo *signal*), 1028
setLevel() (método de *logging.Handler*), 670
setLevel() (método de *logging.Logger*), 666
setlocale() (en el módulo *locale*), 1376
setLocale() (método de *xml.sax.xmlreader.XMLReader*), 1186
setLoggerClass() (en el módulo *logging*), 680
setlogmask() (en el módulo *syslog*), 1898
setLogRecordFactory() (en el módulo *logging*), 680
setmark() (método de *aifc.aifc*), 1352
setMaxConns() (método de *urllib.request.CacheFTPHandler*), 1232
setmode() (en el módulo *msvcrt*), 1867
setName() (método de *threading.Thread*), 773
setnchannels() (método de *aifc.aifc*), 1352
setnchannels() (método de *sunau.AU_write*), 1355
setnchannels() (método de *wave.Wave_write*), 1357
setnframes() (método de *aifc.aifc*), 1352
setnframes() (método de *sunau.AU_write*), 1355
setnframes() (método de *wave.Wave_write*), 1357
SetParamEntityParsing() (método de *xml.parsers.expat.xmlparser*), 1190
setparameters() (método de *ossaudiodev.oss_audio_device*), 1364
setparams() (método de *aifc.aifc*), 1352
setparams() (método de *sunau.AU_write*), 1355
setparams() (método de *wave.Wave_write*), 1357
setpassword() (método de *zipfile.ZipFile*), 489
setpgid() (en el módulo *os*), 564
setpgrp() (en el módulo *os*), 564
setpos() (en el módulo *turtle*), 1391
setpos() (método de *aifc.aifc*), 1351
setpos() (método de *sunau.AU_read*), 1354
setpos() (método de *wave.Wave_read*), 1357
setposition() (en el módulo *turtle*), 1391
setpriority() (en el módulo *os*), 564
setprofile() (en el módulo *sys*), 1705
setprofile() (en el módulo *threading*), 770
SetProperty() (método de *msilib.SummaryInformation*), 1863
setProperty() (método de *xml.sax.xmlreader.XMLReader*), 1186
setPublicId() (método de *xml.sax.xmlreader.InputSource*), 1187
setquota() (método de *imaplib.IMAP4*), 1269
setraw() (en el módulo *tty*), 1888
setrecursionlimit() (en el módulo *sys*), 1705
setregid() (en el módulo *os*), 564
SetReparseDeferralEnabled() (método de *xml.parsers.expat.xmlparser*), 1191
setresgid() (en el módulo *os*), 565
setresuid() (en el módulo *os*), 565
setreuid() (en el módulo *os*), 565
setrlimit() (en el módulo *resource*), 1893
setsampwidth() (método de *aifc.aifc*), 1352
setsampwidth() (método de *sunau.AU_write*), 1355
setsampwidth() (método de *wave.Wave_write*), 1357
setscreg() (método de *curses.window*), 717
setsid() (en el módulo *os*), 565
setsockopt() (método de *socket.socket*), 968
setstate() (en el módulo *random*), 327
setstate() (método de *codecs.IncrementalDecoder*), 165
setstate() (método de *codecs.IncrementalEncoder*), 164
setStream() (método de *logging.StreamHandler*), 692
SetStream() (método de *msilib.Record*), 1864
SetString() (método de *msilib.Record*), 1864
setswitchinterval() (en el módulo *sys*), 1706
setswitchinterval() (en el módulo *test.support*), 1617
setSystemId() (método de *xml.sax.xmlreader.InputSource*), 1187
setsyx() (en el módulo *curses*), 710
setTarget() (método de *logging.handlers.MemoryHandler*), 702
settiltangle() (en el módulo *turtle*), 1402
settimeout() (método de *socket.socket*), 968
setTimeout() (método de *urllib.request.CacheFTPHandler*), 1232
settrace() (en el módulo *sys*), 1706
settrace() (en el módulo *threading*), 770
setuid() (en el módulo *os*), 565
setundobuffer() (en el módulo *turtle*), 1405
setup() (en el módulo *turtle*), 1412
setup() (método de *socketserver.BaseRequestHandler*), 1298
setUp() (método de *unittest.TestCase*), 1528
--setup=S

- timeit command line option, 1657
- SETUP_ANNOTATIONS (opcode), 1848
- SETUP_ASYNC_WITH (opcode), 1848
- setup_environ() (método de `wsgiref.handlers.BaseHandler`), 1217
- SETUP_FINALLY (opcode), 1852
- setup_python() (método de `venv.EnvBuilder`), 1679
- setup_scripts() (método de `venv.EnvBuilder`), 1679
- setup_testing_defaults() (en el módulo `wsgiref.util`), 1210
- SETUP_WITH (opcode), 1849
- setUpClass() (método de `unittest.TestCase`), 1528
- setupterm() (en el módulo `curses`), 710
- SetValue() (en el módulo `winreg`), 1873
- SetValueEx() (en el módulo `winreg`), 1873
- setworldcoordinates() (en el módulo `turtle`), 1407
- setx() (en el módulo `turtle`), 1391
- setxattr() (en el módulo `os`), 595
- sety() (en el módulo `turtle`), 1391
- SF_APPEND (en el módulo `stat`), 400
- SF_ARCHIVED (en el módulo `stat`), 400
- SF_IMMUTABLE (en el módulo `stat`), 400
- SF_MNOWAIT (en el módulo `os`), 573
- SF_NODISKIO (en el módulo `os`), 573
- SF_NOUNLINK (en el módulo `stat`), 400
- SF_SNAPSHOT (en el módulo `stat`), 400
- SF_SYNC (en el módulo `os`), 573
- shape (atributo de `memoryview`), 74
- Shape (clase en `turtle`), 1413
- shape() (en el módulo `turtle`), 1400
- shapesize() (en el módulo `turtle`), 1401
- shapetransform() (en el módulo `turtle`), 1402
- share() (método de `socket.socket`), 968
- ShareableList (clase en `multiprocessing.shared_memory`), 828
- ShareableList() (método de `multiprocessing.managers.SharedMemoryManager`), 828
- Shared Memory, 825
- shared_ciphers() (método de `ssl.SSLSocket`), 987
- SharedMemory (clase en `multiprocessing.shared_memory`), 825
- SharedMemory() (método de `multiprocessing.managers.SharedMemoryManager`), 827
- SharedMemoryManager (clase en `multiprocessing.managers`), 827
- shearfactor() (en el módulo `turtle`), 1401
- Shelf (clase en `shelve`), 439
- shelve
 - módulo, 441
- shelve (módulo), 438
- shield() (en el módulo `asyncio`), 872
- shift() (método de `decimal.Context`), 314
- shift() (método de `decimal.Decimal`), 308
- shift_path_info() (en el módulo `wsgiref.util`), 1210
- shifting
 - operations, 32
- shlex (clase en `shlex`), 1424
- shlex (módulo), 1423
- shm (atributo de `multiprocessing.shared_memory.ShareableList`), 828
- shortDescription() (método de `unittest.TestCase`), 1535
- shorten() (en el módulo `textwrap`), 140
- shouldFlush() (método de `logging.handlers.BufferingHandler`), 701
- shouldFlush() (método de `logging.handlers.MemoryHandler`), 702
- shouldStop (atributo de `unittest.TestResult`), 1541
- show() (método de `curses.panel.Panel`), 727
- show_code() (en el módulo `dis`), 1843
- showsyntaxerror() (método de `code.InteractiveInterpreter`), 1778
- showtraceback() (método de `code.InteractiveInterpreter`), 1778
- showturtle() (en el módulo `turtle`), 1400
- showwarning() (en el módulo `warnings`), 1720
- shuffle() (en el módulo `random`), 328
- shutdown() (en el módulo `logging`), 679
- shutdown() (método de `concurrent.futures.Executor`), 830
- shutdown() (método de `imaplib.IMAP4`), 1269
- shutdown() (método de `multiprocessing.managers.BaseManager`), 803
- shutdown() (método de `socketserver.BaseServer`), 1297
- shutdown() (método de `socket.socket`), 968
- shutdown_asyncgens() (método de `asyncio.loop`), 899
- shutil (módulo), 410
- side_effect (atributo de `unittest.mock.Mock`), 1554
- SIG_BLOCK (en el módulo `signal`), 1026
- SIG_DFL (en el módulo `signal`), 1024
- SIG_IGN (en el módulo `signal`), 1024
- SIG_SETMASK (en el módulo `signal`), 1026
- SIG_UNBLOCK (en el módulo `signal`), 1026
- SIGABRT (en el módulo `signal`), 1024
- SIGALRM (en el módulo `signal`), 1024
- SIGBREAK (en el módulo `signal`), 1025
- SIGBUS (en el módulo `signal`), 1025
- SIGCHLD (en el módulo `signal`), 1025
- SIGCLD (en el módulo `signal`), 1025
- SIGCONT (en el módulo `signal`), 1025
- SIGFPE (en el módulo `signal`), 1025
- SIGHUP (en el módulo `signal`), 1025
- SIGILL (en el módulo `signal`), 1025
- SIGINT (en el módulo `signal`), 1025
- siginterrupt() (en el módulo `signal`), 1028
- SIGKILL (en el módulo `signal`), 1025
- signal
 - módulo, 864

- signal (módulo), 1024
- signal() (en el módulo *signal*), 1029
- signature (atributo de *inspect.BoundsArguments*), 1766
- Signature (clase en *inspect*), 1763
- signature() (en el módulo *inspect*), 1762
- sigpending() (en el módulo *signal*), 1029
- SIGPIPE (en el módulo *signal*), 1025
- SIGSEGV (en el módulo *signal*), 1025
- SIGTERM (en el módulo *signal*), 1025
- sigtimedwait() (en el módulo *signal*), 1029
- SIGUSR1 (en el módulo *signal*), 1025
- SIGUSR2 (en el módulo *signal*), 1025
- sigwait() (en el módulo *signal*), 1029
- sigwaitinfo() (en el módulo *signal*), 1029
- SIGWINCH (en el módulo *signal*), 1026
- Simple Mail Transfer Protocol, 1278
- SimpleCookie (clase en *http.cookies*), 1308
- simplefilter() (en el módulo *warnings*), 1720
- SimpleHandler (clase en *wsgiref.handlers*), 1215
- SimpleHTTPRequestHandler (clase en *http.server*), 1305
- SimpleNamespace (clase en *types*), 257
- SimpleQueue (clase en *multiprocessing*), 793
- SimpleQueue (clase en *queue*), 856
- SimpleXMLRPCRequestHandler (clase en *xmlrpc.server*), 1328
- SimpleXMLRPCServer (clase en *xmlrpc.server*), 1327
- sin() (en el módulo *cmath*), 296
- sin() (en el módulo *math*), 293
- SingleAddressHeader (clase en *email.headerregistry*), 1064
- singledispatch() (en el módulo *functools*), 361
- singledispatchmethod (clase en *functools*), 363
- sinh() (en el módulo *cmath*), 296
- sinh() (en el módulo *math*), 293
- SIO_KEEPAIVE_VALS (en el módulo *socket*), 955
- SIO_LOOPBACK_FAST_PATH (en el módulo *socket*), 955
- SIO_RCVALL (en el módulo *socket*), 955
- site (módulo), 1773
- site command line option
 - user-base, 1776
 - user-site, 1776
- site_maps() (método de *urllib.robotparser.RobotFileParser*), 1248
- sitecustomize
 - módulo, 1774
- site-packages
 - directory, 1773
- sixtofour (atributo de *ipaddress.IPv6Address*), 1337
- size (atributo de *multiprocessing.shared_memory.SharedMemory*), 826
- size (atributo de *struct.Struct*), 158
- size (atributo de *tarfile.TarInfo*), 501
- size (atributo de *tracemalloc.Statistic*), 1669
- size (atributo de *tracemalloc.StatisticDiff*), 1669
- size (atributo de *tracemalloc.Trace*), 1670
- size() (método de *ftplib.FTP*), 1261
- size() (método de *mmap.mmap*), 1034
- size_diff (atributo de *tracemalloc.StatisticDiff*), 1670
- Sized (clase en *collections.abc*), 234
- Sized (clase en *typing*), 1486
- sizeof() (en el módulo *ctypes*), 762
- SKIP (en el módulo *doctest*), 1505
- skip() (en el módulo *unittest*), 1526
- skip() (método de *chunk.Chunk*), 1359
- skip_unless_bind_unix_socket() (en el módulo *test.support*), 1620
- skip_unless_symlink() (en el módulo *test.support*), 1620
- skip_unless_xattr() (en el módulo *test.support*), 1620
- skipIf() (en el módulo *unittest*), 1526
- skipinitialspace (atributo de *csv.Dialect*), 515
- skipped (atributo de *unittest.TestResult*), 1541
- skippedEntity() (método de *xml.sax.handler.ContentHandler*), 1183
- SkipTest, 1526
- skipTest() (método de *unittest.TestCase*), 1529
- skipUnless() (en el módulo *unittest*), 1526
- SLASH (en el módulo *token*), 1827
- SLASHEQUAL (en el módulo *token*), 1828
- slave() (método de *nntplib.NNTP*), 1277
- sleep() (en el módulo *asyncio*), 870
- sleep() (en el módulo *time*), 625
- slice
 - assignment, 40
 - función incorporada, 1854
 - operation, 38
- slice (clase incorporada), 22
- SMALLEST (en el módulo *test.support*), 1616
- SMTP
 - protocol, 1278
- SMTP (clase en *smtplib*), 1278
- SMTP (en el módulo *email.policy*), 1058
- smtp_server (atributo de *smtpd.SMTPChannel*), 1287
- SMTP_SSL (clase en *smtplib*), 1279
- smtp_state (atributo de *smtpd.SMTPChannel*), 1287
- SMTPAuthenticationError, 1280
- SMTPChannel (clase en *smtpd*), 1286
- SMTPConnectError, 1280
- smtpd (módulo), 1285
- SMTPDataError, 1280
- SMTPException, 1279
- SMTPHandler (clase en *logging.handlers*), 701
- SMTPHeloError, 1280
- smtplib (módulo), 1278
- SMTPNotSupportedError, 1280
- SMTPRecipientsRefused, 1280
- SMTPResponseException, 1279
- SMTPSenderRefused, 1279
- SMTPServer (clase en *smtpd*), 1285

- SMTPServerDisconnected, 1279
- SMTPUTF8 (en el módulo *email.policy*), 1059
- Snapshot (clase en *tracemalloc*), 1668
- SND_ALIAS (en el módulo *winsound*), 1878
- SND_ASYNC (en el módulo *winsound*), 1878
- SND_FILENAME (en el módulo *winsound*), 1878
- SND_LOOP (en el módulo *winsound*), 1878
- SND_MEMORY (en el módulo *winsound*), 1878
- SND_NODEFAULT (en el módulo *winsound*), 1878
- SND_NOSTOP (en el módulo *winsound*), 1878
- SND_NOWAIT (en el módulo *winsound*), 1878
- SND_PURGE (en el módulo *winsound*), 1878
- sndhdr (módulo), 1361
- sni_callback (atributo de *ssl.SSLContext*), 993
- sniff() (método de *csv.Sniffer*), 514
- Sniffer (clase en *csv*), 514
- sock_accept() (método de *asyncio.loop*), 908
- SOCK_CLOEXEC (en el módulo *socket*), 954
- sock_connect() (método de *asyncio.loop*), 908
- SOCK_DGRAM (en el módulo *socket*), 953
- SOCK_MAX_SIZE (en el módulo *test.support*), 1615
- SOCK_NONBLOCK (en el módulo *socket*), 954
- SOCK_RAW (en el módulo *socket*), 953
- SOCK_RDM (en el módulo *socket*), 953
- sock_recv() (método de *asyncio.loop*), 907
- sock_recv_into() (método de *asyncio.loop*), 907
- sock_sendall() (método de *asyncio.loop*), 907
- sock_sendfile() (método de *asyncio.loop*), 908
- SOCK_SEQPACKET (en el módulo *socket*), 953
- SOCK_STREAM (en el módulo *socket*), 953
- socket
- módulo, 1199
 - objeto, 950
- socket (atributo de *socketserver.BaseServer*), 1297
- socket (módulo), 950
- socket() (en el módulo *socket*), 956
- socket() (in module *socket*), 1008
- socket() (método de *imaplib.IMAP4*), 1269
- socket_type (atributo de *socketserver.BaseServer*), 1297
- SocketHandler (clase en *logging.handlers*), 697
- socketpair() (en el módulo *socket*), 957
- sockets (atributo de *asyncio.Server*), 915
- socketserver (módulo), 1294
- SocketType (en el módulo *socket*), 958
- SOL_ALG (en el módulo *socket*), 955
- SOL_RDS (en el módulo *socket*), 955
- SOMAXCONN (en el módulo *socket*), 954
- sort() (método de *imaplib.IMAP4*), 1269
- sort() (método de *list*), 41
- sort_stats() (método de *pstats.Stats*), 1651
- sortdict() (en el módulo *test.support*), 1617
- sorted() (función incorporada), 22
- sort-keys
- json.tool command line option, 1105
- sortTestMethodsUsing (atributo de *unittest.TestLoader*), 1541
- source (atributo de *doctest.Example*), 1512
- source (atributo de *shlex.shlex*), 1426
- source (*pdb* command), 1645
- SOURCE_DATE_EPOCH, 1837, 1839
- source_from_cache() (en el módulo *imp*), 1930
- source_from_cache() (en el módulo *importlib.util*), 1804
- source_hash() (en el módulo *importlib.util*), 1805
- SOURCE_SUFFIXES (en el módulo *importlib.machinery*), 1799
- source_to_code() (método estático de *importlib.abc.InspectLoader*), 1796
- SourceFileLoader (clase en *importlib.machinery*), 1801
- sourcehook() (método de *shlex.shlex*), 1424
- SourcelessFileLoader (clase en *importlib.machinery*), 1801
- SourceLoader (clase en *importlib.abc*), 1796
- space
- in printf-style formatting, 53, 67
 - in string formatting, 103
- span() (método de *re.Match*), 124
- spawn() (en el módulo *pty*), 1888
- spawn_python() (en el módulo *test.support.script_helper*), 1628
- spawnl() (en el módulo *os*), 601
- spawnle() (en el módulo *os*), 601
- spawnlp() (en el módulo *os*), 601
- spawnlpe() (en el módulo *os*), 601
- spawnv() (en el módulo *os*), 601
- spawnve() (en el módulo *os*), 601
- spawnvp() (en el módulo *os*), 601
- spawnvpe() (en el módulo *os*), 601
- spec_from_file_location() (en el módulo *importlib.util*), 1805
- spec_from_loader() (en el módulo *importlib.util*), 1805
- special
- method, 1946
- SpecialFileError, 497
- specified_attributes (atributo de *xml.parsers.expat.xmlparser*), 1191
- speed() (en el módulo *turtle*), 1394
- speed() (método de *ossaudiodev.oss_audio_device*), 1363
- Spinbox (clase en *tkinter.ttk*), 1445
- split() (en el módulo *os.path*), 393
- split() (en el módulo *re*), 117
- split() (en el módulo *shlex*), 1423
- split() (método de *bytearray*), 61
- split() (método de *bytes*), 61
- split() (método de *re.Pattern*), 121
- split() (método de *str*), 49
- splitdrive() (en el módulo *os.path*), 394
- splittext() (en el módulo *os.path*), 394
- splitlines() (método de *bytearray*), 64
- splitlines() (método de *bytes*), 64
- splitlines() (método de *str*), 50
- SplitResult (clase en *urllib.parse*), 1244

- `SplitResultBytes` (clase en `urllib.parse`), 1244
- `SpooledTemporaryFile()` (en el módulo `tempfile`), 404
- `sprintf-style formatting`, 52, 66
- `spwd` (módulo), 1883
- `sqlite3` (módulo), 446
- `sqlite_version` (en el módulo `sqlite3`), 448
- `sqlite_version_info` (en el módulo `sqlite3`), 448
- `sqrt()` (en el módulo `cmath`), 296
- `sqrt()` (en el módulo `math`), 292
- `sqrt()` (método de `decimal.Context`), 314
- `sqrt()` (método de `decimal.Decimal`), 308
- `SSL`, 973
- `ssl` (módulo), 973
- `SSL_CERT_FILE`, 1006
- `SSL_CERT_PATH`, 1006
- `ssl_version` (atributo de `ftplib.FTP_TLS`), 1262
- `SSLCertVerificationError`, 976
- `SSLContext` (clase en `ssl`), 989
- `SSLEOFError`, 976
- `SSL_ERROR`, 975
- `SSL_ERROR_NUMBER` (clase en `ssl`), 985
- `SSLKEYLOGFILE`, 975
- `SSLObject` (clase en `ssl`), 1002
- `sslobject_class` (atributo de `ssl.SSLContext`), 994
- `SSLSession` (clase en `ssl`), 1004
- `SSLSocket` (clase en `ssl`), 985
- `sslsocket_class` (atributo de `ssl.SSLContext`), 994
- `SSLSyscallError`, 976
- `SSLv3` (atributo de `ssl.TLSVersion`), 985
- `SSLWantReadError`, 975
- `SSLWantWriteError`, 975
- `SSLZeroReturnError`, 975
- `st()` (en el módulo `turtle`), 1400
- `st2list()` (en el módulo `parser`), 1815
- `st2tuple()` (en el módulo `parser`), 1815
- `st_atime` (atributo de `os.stat_result`), 587
- `ST_ATIME` (en el módulo `stat`), 398
- `st_atime_ns` (atributo de `os.stat_result`), 588
- `st_birthtime` (atributo de `os.stat_result`), 588
- `st_blksize` (atributo de `os.stat_result`), 588
- `st_blocks` (atributo de `os.stat_result`), 588
- `st_creator` (atributo de `os.stat_result`), 588
- `st_ctime` (atributo de `os.stat_result`), 587
- `ST_CTIME` (en el módulo `stat`), 398
- `st_ctime_ns` (atributo de `os.stat_result`), 588
- `st_dev` (atributo de `os.stat_result`), 587
- `ST_DEV` (en el módulo `stat`), 398
- `st_file_attributes` (atributo de `os.stat_result`), 589
- `st_flags` (atributo de `os.stat_result`), 588
- `st_fstype` (atributo de `os.stat_result`), 588
- `st_gen` (atributo de `os.stat_result`), 588
- `st_gid` (atributo de `os.stat_result`), 587
- `ST_GID` (en el módulo `stat`), 398
- `st_ino` (atributo de `os.stat_result`), 587
- `ST_INO` (en el módulo `stat`), 398
- `st_mode` (atributo de `os.stat_result`), 587
- `ST_MODE` (en el módulo `stat`), 398
- `st_mtime` (atributo de `os.stat_result`), 587
- `ST_MTIME` (en el módulo `stat`), 398
- `st_mtime_ns` (atributo de `os.stat_result`), 588
- `st_nlink` (atributo de `os.stat_result`), 587
- `ST_NLINK` (en el módulo `stat`), 398
- `st_rdev` (atributo de `os.stat_result`), 588
- `st_reparse_tag` (atributo de `os.stat_result`), 589
- `st_rsize` (atributo de `os.stat_result`), 588
- `st_size` (atributo de `os.stat_result`), 587
- `ST_SIZE` (en el módulo `stat`), 398
- `st_type` (atributo de `os.stat_result`), 588
- `st_uid` (atributo de `os.stat_result`), 587
- `ST_UID` (en el módulo `stat`), 398
- `stack` (atributo de `traceback.TracebackException`), 1749
- `stack viewer`, 1466
- `stack()` (en el módulo `inspect`), 1770
- `stack_effect()` (en el módulo `dis`), 1844
- `stack_size()` (en el módulo `_thread`), 863
- `stack_size()` (en el módulo `threading`), 770
- `stackable`
 streams, 158
- `StackSummary` (clase en `traceback`), 1750
- `stamp()` (en el módulo `turtle`), 1393
- `standard_b64decode()` (en el módulo `base64`), 1127
- `standard_b64encode()` (en el módulo `base64`), 1127
- `standarderror (2to3 fixer)`, 1611
- `standend()` (método de `curses.window`), 717
- `standout()` (método de `curses.window`), 717
- `STAR` (en el módulo `token`), 1827
- `STAREQUAL` (en el módulo `token`), 1828
- `starmap()` (en el módulo `itertools`), 352
- `starmap()` (método de `multiprocessing.pool.Pool`), 810
- `starmap_async()` (método de `multiprocessing.pool.Pool`), 810
- `start` (atributo de `range`), 43
- `start` (atributo de `UnicodeError`), 94
- `start()` (en el módulo `tracemalloc`), 1666
- `start()` (método de `logging.handlers.QueueListener`), 704
- `start()` (método de `multiprocessing.managers.BaseManager`), 803
- `start()` (método de `multiprocessing.Process`), 789
- `start()` (método de `re.Match`), 123
- `start()` (método de `threading.Thread`), 772
- `start()` (método de `tkinter.ttk.Progressbar`), 1449
- `start()` (método de `xml.etree.ElementTree.TreeBuilder`), 1158
- `start_color()` (en el módulo `curses`), 710
- `start_component()` (método de `msilib.Directory`), 1865
- `start_new_thread()` (en el módulo `_thread`), 862
- `start_ns()` (método de `xml.etree.ElementTree.TreeBuilder`), 1158
- `start_server()` (en el módulo `asyncio`), 879

- `start_serving()` (método de `asyncio.Server`), 914
`start_threads()` (en el módulo `test.support`), 1620
`start_tls()` (método de `asyncio.loop`), 906
`start_unix_server()` (en el módulo `asyncio`), 880
`StartCdataSectionHandler()` (método de `xml.parsers.expat.xmlparser`), 1193
`--start-directory directory`
`unittest-discover` command line option, 1522
`StartDoctypeDeclHandler()` (método de `xml.parsers.expat.xmlparser`), 1192
`startDocument()` (método de `xml.sax.handler.ContentHandler`), 1181
`startElement()` (método de `xml.sax.handler.ContentHandler`), 1182
`StartElementHandler()` (método de `xml.parsers.expat.xmlparser`), 1193
`startElementNS()` (método de `xml.sax.handler.ContentHandler`), 1182
`STARTF_USESHOWWINDOW` (en el módulo `subprocess`), 847
`STARTF_USESTDHANDLES` (en el módulo `subprocess`), 847
`startfile()` (en el módulo `os`), 602
`StartNamespaceDeclHandler()` (método de `xml.parsers.expat.xmlparser`), 1193
`startPrefixMapping()` (método de `xml.sax.handler.ContentHandler`), 1181
`startswith()` (método de `bytearray`), 59
`startswith()` (método de `bytes`), 59
`startswith()` (método de `str`), 51
`startTest()` (método de `unittest.TestResult`), 1542
`startTestRun()` (método de `unittest.TestResult`), 1542
`starttls()` (método de `imaplib.IMAP4`), 1270
`starttls()` (método de `nnplib.NNTP`), 1274
`starttls()` (método de `smtpplib.SMTP`), 1282
`STARTUPINFO` (clase en `subprocess`), 846
`stat`
módulo, 586
`stat` (módulo), 396
`stat()` (en el módulo `os`), 586
`stat()` (método de `nnplib.NNTP`), 1276
`stat()` (método de `os.DirEntry`), 586
`stat()` (método de `pathlib.Path`), 383
`stat()` (método de `poplib.POP3`), 1264
`stat_result` (clase en `os`), 587
`state()` (método de `tkinter.ttk.Widget`), 1443
`staticmethod()` (función incorporada), 22
`Statistic` (clase en `tracemalloc`), 1669
`StatisticDiff` (clase en `tracemalloc`), 1669
`statistics` (módulo), 332
`statistics()` (método de `tracemalloc.Snapshot`), 1668
`StatisticsError`, 339
`Stats` (clase en `pstats`), 1650
`status` (atributo de `http.client.HTTPResponse`), 1255
`status()` (método de `imaplib.IMAP4`), 1270
`statvfs()` (en el módulo `os`), 589
`STD_ERROR_HANDLE` (en el módulo `subprocess`), 847
`STD_INPUT_HANDLE` (en el módulo `subprocess`), 847
`STD_OUTPUT_HANDLE` (en el módulo `subprocess`), 847
`StdMessageBox` (clase en `tkinter.tix`), 1460
`stderr` (atributo de `asyncio.asyncio.subprocess.Process`), 892
`stderr` (atributo de `subprocess.CalledProcessError`), 839
`stderr` (atributo de `subprocess.CompletedProcess`), 838
`stderr` (atributo de `subprocess.Popen`), 845
`stderr` (atributo de `subprocess.TimeoutExpired`), 838
`stderr` (en el módulo `sys`), 1707
`stdev` (atributo de `statistics.NormalDist`), 339
`stdev()` (en el módulo `statistics`), 337
`stdin` (atributo de `asyncio.asyncio.subprocess.Process`), 892
`stdin` (atributo de `subprocess.Popen`), 845
`stdin` (en el módulo `sys`), 1707
`stdout` (atributo de `asyncio.asyncio.subprocess.Process`), 892
`stdout` (atributo de `subprocess.CalledProcessError`), 839
`stdout` (atributo de `subprocess.CompletedProcess`), 837
`stdout` (atributo de `subprocess.Popen`), 845
`stdout` (atributo de `subprocess.TimeoutExpired`), 838
`STDOUT` (en el módulo `subprocess`), 838
`stdout` (en el módulo `sys`), 1707
`step` (atributo de `range`), 43
`step` (`pdb` command), 1644
`step()` (método de `tkinter.ttk.Progressbar`), 1449
`stereocontrols()` (método de `ossaudio-dev.oss_mixer_device`), 1365
`stls()` (método de `poplib.POP3`), 1264
`stop` (atributo de `range`), 43
`stop()` (en el módulo `tracemalloc`), 1666
`stop()` (método de `asyncio.loop`), 899
`stop()` (método de `logging.handlers.QueueListener`), 704
`stop()` (método de `tkinter.ttk.Progressbar`), 1449
`stop()` (método de `unittest.TestResult`), 1542
`stop_here()` (método de `bdb.Bdb`), 1635
`StopAsyncIteration`, 93
`StopIteration`, 92
`stopListening()` (en el módulo `logging.config`), 683
`stopTest()` (método de `unittest.TestResult`), 1542
`stopTestRun()` (método de `unittest.TestResult`), 1542
`storbinary()` (método de `ftplib.FTP`), 1260
`store()` (método de `imaplib.IMAP4`), 1270
`STORE_ACTIONS` (atributo de `optparse.Option`), 1927
`STORE_ATTR` (opcode), 1850
`STORE_DEREF` (opcode), 1852
`STORE_FAST` (opcode), 1852
`STORE_GLOBAL` (opcode), 1850
`STORE_NAME` (opcode), 1849
`STORE_SUBSCR` (opcode), 1847
`storlines()` (método de `ftplib.FTP`), 1260
`str` (built-in class)

- (see also `string`), 44
- `str` (clase incorporada), 44
- `str()` (en el módulo `locale`), 1381
- `strcoll()` (en el módulo `locale`), 1380
- `StreamError`, 496
- `StreamHandler` (clase en `logging`), 692
- `streamreader` (atributo de `codecs.CodecInfo`), 159
- `StreamReader` (clase en `asyncio`), 880
- `StreamReader` (clase en `codecs`), 166
- `StreamReaderWriter` (clase en `codecs`), 167
- `StreamRecoder` (clase en `codecs`), 167
- `StreamRequestHandler` (clase en `socketserver`), 1298
- `streams`, 158
 - `stackable`, 158
- `streamwriter` (atributo de `codecs.CodecInfo`), 159
- `StreamWriter` (clase en `asyncio`), 881
- `StreamWriter` (clase en `codecs`), 165
- `strerror` (atributo de `OSError`), 92
- `strerror()` (en el módulo `os`), 565
- `strftime()` (en el módulo `time`), 626
- `strftime()` (método de `datetime.date`), 186
- `strftime()` (método de `datetime.datetime`), 195
- `strftime()` (método de `datetime.time`), 200
- `strict` (atributo de `csv.Dialect`), 515
- `strict` (en el módulo `email.policy`), 1059
- `strict_domain` (atributo de `http.cookiejar.DefaultCookiePolicy`), 1317
- `strict_errors()` (en el módulo `codecs`), 162
- `strict_ns_domain` (atributo de `http.cookiejar.DefaultCookiePolicy`), 1317
- `strict_ns_set_initial_dollar` (atributo de `http.cookiejar.DefaultCookiePolicy`), 1317
- `strict_ns_set_path` (atributo de `http.cookiejar.DefaultCookiePolicy`), 1317
- `strict_ns_unverifiable` (atributo de `http.cookiejar.DefaultCookiePolicy`), 1317
- `strict_rfc2965_unverifiable` (atributo de `http.cookiejar.DefaultCookiePolicy`), 1317
- `strides` (atributo de `memoryview`), 74
- `string`
 - `format()` (built-in function), 12
 - `formatting`, `printf`, 52
 - `interpolation`, `printf`, 52
 - `methods`, 45
 - módulo, 1381
 - objeto, 44
 - `str` (built-in class), 44
 - `str()` (built-in function), 23
 - text sequence type, 44
- `string` (atributo de `re.Match`), 124
- `STRING` (en el módulo `token`), 1827
- `string` (módulo), 99
- `string_at()` (en el módulo `ctypes`), 762
- `StringIO` (clase en `io`), 620
- `stringprep` (módulo), 145
- `strip()` (método de `bytearray`), 61
- `strip()` (método de `bytes`), 61
- `strip()` (método de `str`), 51
- `strip_dirs()` (método de `pstats.Stats`), 1650
- `strip_python_strerr()` (en el módulo `test.support`), 1619
- `stripspaces` (atributo de `curses.textpad.Textbox`), 723
- `strptime()` (en el módulo `time`), 627
- `strptime()` (método de clase de `datetime.datetime`), 190
- `strsignal()` (en el módulo `signal`), 1027
- `struct`
 - módulo, 968
- `Struct` (clase en `struct`), 158
- `struct` (módulo), 153
- `struct_time` (clase en `time`), 627
- `Structure` (clase en `ctypes`), 766
- `structures`
 - C, 153
- `strxfrm()` (en el módulo `locale`), 1380
- `STType` (en el módulo `parser`), 1816
- `Style` (clase en `tkinter.ttk`), 1455
- `sub()` (en el módulo `operator`), 367
- `sub()` (en el módulo `re`), 118
- `sub()` (método de `re.Pattern`), 121
- `subdirs` (atributo de `filecmp.dircmp`), 403
- `SubElement()` (en el módulo `xml.etree.ElementTree`), 1151
- `submit()` (método de `concurrent.futures.Executor`), 830
- `submodule_search_locations` (atributo de `importlib.machinery.ModuleSpec`), 1803
- `subn()` (en el módulo `re`), 119
- `subn()` (método de `re.Pattern`), 121
- `subnet_of()` (método de `ipaddress.IPv4Network`), 1341
- `subnet_of()` (método de `ipaddress.IPv6Network`), 1342
- `subnets()` (método de `ipaddress.IPv4Network`), 1340
- `subnets()` (método de `ipaddress.IPv6Network`), 1342
- `Subnormal` (clase en `decimal`), 316
- `suboffsets` (atributo de `memoryview`), 75
- `subpad()` (método de `curses.window`), 717
- `subprocess` (módulo), 836
- `subprocess_exec()` (método de `asyncio.loop`), 912
- `subprocess_shell()` (método de `asyncio.loop`), 913
- `SubprocessError`, 838
- `SubprocessProtocol` (clase en `asyncio`), 926
- `SubprocessTransport` (clase en `asyncio`), 922
- `subscribe()` (método de `imaplib.IMAP4`), 1270
- `subscript`
 - assignment, 40
 - operation, 38
- `subsequent_indent` (atributo de `textwrap.TextWrapper`), 142
- `substitute()` (método de `string.Template`), 108
- `subTest()` (método de `unittest.TestCase`), 1529
- `subtract()` (método de `collections.Counter`), 221

- `subtract()` (método de `decimal.Context`), 314
- `subtype` (atributo de `email.headerregistry.ContentTypeHeader`), 1064
- `subwin()` (método de `curses.window`), 717
- `successful()` (método de `multiprocessing.pool.AsyncResult`), 811
- `suffix_map` (atributo de `mimetypes.MimeTypes`), 1126
- `suffix_map` (en el módulo `mimetypes`), 1125
- `suite()` (en el módulo `parser`), 1814
- `suiteClass` (atributo de `unittest.TestLoader`), 1541
- `sum()` (función incorporada), 23
- `summarize()` (método de `doctest.DocTestRunner`), 1515
- `summarize_address_range()` (en el módulo `ipaddress`), 1345
- `--summary`
trace command line option, 1660
- `sunau` (módulo), 1353
- `super` (atributo de `pyclbr.Class`), 1836
- `super()` (función incorporada), 23
- `supernet()` (método de `ipaddress.IPv4Network`), 1340
- `supernet()` (método de `ipaddress.IPv6Network`), 1342
- `supernet_of()` (método de `ipaddress.IPv4Network`), 1341
- `supernet_of()` (método de `ipaddress.IPv6Network`), 1342
- `supports_bytes_environ` (en el módulo `os`), 565
- `supports_dir_fd` (en el módulo `os`), 590
- `supports_effective_ids` (en el módulo `os`), 590
- `supports_fd` (en el módulo `os`), 590
- `supports_follow_symlinks` (en el módulo `os`), 590
- `supports_unicode_filenames` (en el módulo `os.path`), 394
- `SupportsAbs` (clase en `typing`), 1486
- `SupportsBytes` (clase en `typing`), 1486
- `SupportsComplex` (clase en `typing`), 1486
- `SupportsFloat` (clase en `typing`), 1485
- `SupportsIndex` (clase en `typing`), 1486
- `SupportsInt` (clase en `typing`), 1485
- `SupportsRound` (clase en `typing`), 1486
- `suppress()` (en el módulo `contextlib`), 1731
- `SuppressCrashReport` (clase en `test.support`), 1626
- `SW_HIDE` (en el módulo `subprocess`), 847
- `swap_attr()` (en el módulo `test.support`), 1620
- `swap_item()` (en el módulo `test.support`), 1620
- `swapcase()` (método de `bytearray`), 64
- `swapcase()` (método de `bytes`), 64
- `swapcase()` (método de `str`), 51
- `sym_name` (en el módulo `symbol`), 1826
- `Symbol` (clase en `symtable`), 1825
- `symbol` (módulo), 1826
- `SymbolTable` (clase en `symtable`), 1824
- `symlink()` (en el módulo `os`), 590
- `symlink_to()` (método de `pathlib.Path`), 387
- `symmetric_difference()` (método de `frozenset`), 76
- `symmetric_difference_update()` (método de `frozenset`), 77
- `symtable` (módulo), 1824
- `symtable()` (en el módulo `symtable`), 1824
- `sync()` (en el módulo `os`), 591
- `sync()` (método de `dbm.dumb.dumbdbm`), 446
- `sync()` (método de `dbm.gnu.gdbm`), 444
- `sync()` (método de `ossaudiodev.oss_audio_device`), 1363
- `sync()` (método de `shelve.Shelf`), 439
- `syncdown()` (método de `curses.window`), 717
- `synchronized()` (en el módulo `multiprocessing.sharedctypes`), 801
- `SyncManager` (clase en `multiprocessing.managers`), 804
- `syncok()` (método de `curses.window`), 717
- `syncup()` (método de `curses.window`), 717
- `SyntaxErr`, 1170
- `SyntaxError`, 93
- `SyntaxWarning`, 96
- `sys`
módulo, 19
- `sys` (módulo), 1691
- `sys_exc (2to3 fixer)`, 1611
- `sys_version` (atributo de `http.server.BaseHTTPRequestHandler`), 1303
- `sysconf()` (en el módulo `os`), 608
- `sysconf_names` (en el módulo `os`), 608
- `sysconfig` (módulo), 1710
- `syslog` (módulo), 1897
- `syslog()` (en el módulo `syslog`), 1897
- `SysLogHandler` (clase en `logging.handlers`), 698
- `system()` (en el módulo `os`), 603
- `system()` (en el módulo `platform`), 728
- `system_alias()` (en el módulo `platform`), 728
- `system_must_validate_cert()` (en el módulo `test.support`), 1617
- `SystemError`, 93
- `SystemExit`, 93
- `systemId` (atributo de `xml.dom.DocumentType`), 1165
- `SystemRandom` (clase en `random`), 330
- `SystemRandom` (clase en `secrets`), 555
- `SystemRoot`, 842
- ## T
- `-t`
trace command line option, 1660
`unittest-discover` command line option, 1523
- `-T`
trace command line option, 1660
- `-t <tarfile>`
tarfile command line option, 507

`-t <zipfile>`
 `zipfile` command line option, 494
`T_FMT` (en el módulo *locale*), 1378
`T_FMT_AMPM` (en el módulo *locale*), 1378
`tab()` (método de *tkinter.ttk.Notebook*), 1447
`TabError`, 93
`tabnanny` (módulo), 1834
`tabs()` (método de *tkinter.ttk.Notebook*), 1447
`tabsize` (atributo de *textwrap.TextWrapper*), 142
`tabular`
 data, 511
`tag` (atributo de *xml.etree.ElementTree.Element*), 1153
`tag_bind()` (método de *tkinter.ttk.Treeview*), 1455
`tag_configure()` (método de *tkinter.ttk.Treeview*), 1455
`tag_has()` (método de *tkinter.ttk.Treeview*), 1455
`tagName` (atributo de *xml.dom.Element*), 1167
`tail` (atributo de *xml.etree.ElementTree.Element*), 1153
`take_snapshot()` (en el módulo *tracemalloc*), 1666
`takewhile()` (en el módulo *itertools*), 352
`tan()` (en el módulo *cmath*), 296
`tan()` (en el módulo *math*), 293
`tanh()` (en el módulo *cmath*), 296
`tanh()` (en el módulo *math*), 293
`tar_filter()` (en el módulo *tarfile*), 504
`TarError`, 496
`TarFile` (clase en *tarfile*), 497
`tarfile` (módulo), 495
`tarfile` command line option
 `-c <tarfile> <source1> ...`
 `<sourceN>`, 507
 `--create <tarfile> <source1> ...`
 `<sourceN>`, 507
 `-e <tarfile> [<output_dir>]`, 507
 `--extract <tarfile>`
 `[<output_dir>]`, 507
 `--filter <filtername>`, 507
 `-l <tarfile>`, 507
 `--list <tarfile>`, 507
 `-t <tarfile>`, 507
 `--test <tarfile>`, 507
 `-v`, 507
 `--verbose`, 507
`target` (atributo de *xml.dom.ProcessingInstruction*), 1169
`tarinfo` (atributo de *tarfile.FilterError*), 496
`TarInfo` (clase en *tarfile*), 501
`Task` (clase en *asyncio*), 875
`task_done()` (método de *asyncio.Queue*), 894
`task_done()` (método de *multiprocessing.JoinableQueue*), 793
`task_done()` (método de *queue.Queue*), 857
`tau` (en el módulo *cmath*), 297
`tau` (en el módulo *math*), 294
`tb_locals` (atributo de *unittest.TestResult*), 1542
`tbreak` (*pdb* command), 1643
`tcdrain()` (en el módulo *termios*), 1887
`tcflow()` (en el módulo *termios*), 1887
`tcflush()` (en el módulo *termios*), 1887
`tcgetattr()` (en el módulo *termios*), 1887
`tcgetpgrp()` (en el módulo *os*), 574
`Tcl()` (en el módulo *tkinter*), 1430
`TCPServer` (clase en *socketserver*), 1294
`tcsendbreak()` (en el módulo *termios*), 1887
`tcsetattr()` (en el módulo *termios*), 1887
`tcsetpgrp()` (en el módulo *os*), 574
`tearDown()` (método de *unittest.TestCase*), 1528
`tearDownClass()` (método de *unittest.TestCase*), 1528
`tee()` (en el módulo *itertools*), 352
`tell()` (método de *aifc.aifc*), 1351
`tell()` (método de *chunk.Chunk*), 1359
`tell()` (método de *io.IOBBase*), 614
`tell()` (método de *io.TextIOBase*), 619
`tell()` (método de *mmap.mmap*), 1034
`tell()` (método de *sunau.AU_read*), 1354
`tell()` (método de *sunau.AU_write*), 1355
`tell()` (método de *wave.Wave_read*), 1357
`tell()` (método de *wave.Wave_write*), 1357
`Telnet` (clase en *telnetlib*), 1288
`telnetlib` (módulo), 1288
`TEMP`, 406
`temp_cwd()` (en el módulo *test.support*), 1619
`temp_dir()` (en el módulo *test.support*), 1619
`temp_umask()` (en el módulo *test.support*), 1619
`tempdir` (en el módulo *tempfile*), 406
`tempfile` (módulo), 403
`template` (atributo de *string.Template*), 108
`Template` (clase en *pipes*), 1892
`Template` (clase en *string*), 108
`temporary`
 file, 403
 file name, 403
`TemporaryDirectory()` (en el módulo *tempfile*), 404
`TemporaryFile()` (en el módulo *tempfile*), 403
`teredo` (atributo de *ipaddress.IPv6Address*), 1337
`TERM`, 710
`termattrs()` (en el módulo *curses*), 710
`terminal_size` (clase en *os*), 574
`terminate()` (método de *asyncio.asyncio.subprocess.Process*), 892
`terminate()` (método de *asyncio.SubprocessTransport*), 926
`terminate()` (método de *multiprocessing.pool.Pool*), 810
`terminate()` (método de *multiprocessing.Process*), 790
`terminate()` (método de *subprocess.Popen*), 845
`termios` (módulo), 1886
`termname()` (en el módulo *curses*), 710
`test` (atributo de *doctest.DocTestFailure*), 1518
`test` (atributo de *doctest.UnexpectedException*), 1518
`test` (módulo), 1612
`--test <tarfile>`
 `tarfile` command line option, 507

- test <zipfile>
 - zipfile command line option, 494
- test() (en el módulo cgi), 1206
- TEST_DATA_DIR (en el módulo test.support), 1615
- TEST_HOME_DIR (en el módulo test.support), 1615
- TEST_HTTP_URL (en el módulo test.support), 1616
- TEST_SUPPORT_DIR (en el módulo test.support), 1615
- TestCase (clase en unittest), 1528
- TestFailed, 1614
- testfile() (en el módulo doctest), 1508
- TESTFN (en el módulo test.support), 1615
- TESTFN_ENCODING (en el módulo test.support), 1615
- TESTFN_NONASCII (en el módulo test.support), 1615
- TESTFN_UNDECODABLE (en el módulo test.support), 1615
- TESTFN_UNENCODABLE (en el módulo test.support), 1615
- TESTFN_UNICODE (en el módulo test.support), 1615
- TestHandler (clase en test.support), 1627
- TestLoader (clase en unittest), 1539
- testMethodPrefix (atributo de unittest.TestLoader), 1541
- testmod() (en el módulo doctest), 1508
- testNamePatterns (atributo de unittest.TestLoader), 1541
- TestResult (clase en unittest), 1541
- tests (en el módulo imghdr), 1360
- testsource() (en el módulo doctest), 1517
- testsRun (atributo de unittest.TestResult), 1542
- TestSuite (clase en unittest), 1538
- test.support (módulo), 1614
- test.support.script_helper (módulo), 1627
- testzip() (método de zipfile.ZipFile), 489
- text (atributo de SyntaxError), 93
- text (atributo de traceback.TracebackException), 1749
- text (atributo de xml.etree.ElementTree.Element), 1153
- Text (clase en typing), 1489
- text (en el módulo msilib), 1866
- text mode, 19
- text() (en el módulo cgitb), 1209
- text() (método de msilib.Dialog), 1866
- text_factory (atributo de sqlite3.Connection), 454
- Textbox (clase en curses.textpad), 722
- TextCalendar (clase en calendar), 214
- textdomain() (en el módulo gettext), 1368
- textdomain() (en el módulo locale), 1383
- textinput() (en el módulo turtle), 1409
- TextIO (clase en typing), 1489
- TextIOBase (clase en io), 619
- TextIOWrapper (clase en io), 619
- TextTestResult (clase en unittest), 1543
- TextTestRunner (clase en unittest), 1543
- textwrap (módulo), 140
- TextWrapper (clase en textwrap), 142
- theme_create() (método de tkinter.ttk.Style), 1457
- theme_names() (método de tkinter.ttk.Style), 1458
- theme_settings() (método de tkinter.ttk.Style), 1457
- theme_use() (método de tkinter.ttk.Style), 1458
- THOUSEP (en el módulo locale), 1378
- Thread (clase en threading), 772
- thread() (método de imaplib.IMAP4), 1270
- thread_info (en el módulo sys), 1708
- thread_time() (en el módulo time), 628
- thread_time_ns() (en el módulo time), 628
- ThreadedChildWatcher (clase en asyncio), 937
- threading (módulo), 769
- threading_cleanup() (en el módulo test.support), 1623
- threading_setup() (en el módulo test.support), 1623
- ThreadingHTTPServer (clase en http.server), 1302
- ThreadingMixin (clase en socketserver), 1295
- ThreadingTCPServer (clase en socketserver), 1296
- ThreadingUDPServer (clase en socketserver), 1296
- ThreadPool (clase en multiprocessing.pool), 816
- ThreadPoolExecutor (clase en concurrent.futures), 831
- threads
 - POSIX, 862
- throw (2to3 fixer), 1611
- ticket_lifetime_hint (atributo de ssl.SSLSession), 1004
- tigetflag() (en el módulo curses), 710
- tigetnum() (en el módulo curses), 710
- tigetstr() (en el módulo curses), 711
- TILDE (en el módulo token), 1828
- tilt() (en el módulo turtle), 1401
- tiltangle() (en el módulo turtle), 1402
- time (atributo de ssl.SSLSession), 1004
- time (clase en datetime), 197
- time (módulo), 622
- time() (en el módulo time), 628
- time() (método de asyncio.loop), 901
- time() (método de datetime.datetime), 191
- Time2Internaldate() (en el módulo imaplib), 1266
- time_ns() (en el módulo time), 628
- timedelta (clase en datetime), 179
- TimedRotatingFileHandler (clase en logging.handlers), 695
- timegm() (en el módulo calendar), 216
- timeit (módulo), 1654
- timeit command line option
 - h, 1657
 - help, 1657
 - n N, 1657
 - number=N, 1657
 - p, 1657
 - process, 1657
 - r N, 1657
 - repeat=N, 1657
 - s S, 1657
 - setup=S, 1657

-u, 1657
--unit=U, 1657
-v, 1657
--verbose, 1657
timeit() (en el módulo *timeit*), 1655
timeit() (método de *timeit.Timer*), 1656
timeout, 953
timeout (atributo de *socketserver.BaseServer*), 1297
timeout (atributo de *ssl.SSLSession*), 1004
timeout (atributo de *subprocess.TimeoutExpired*), 838
timeout() (método de *curses.window*), 717
TIMEOUT_MAX (en el módulo *_thread*), 863
TIMEOUT_MAX (en el módulo *threading*), 771
TimeoutError, 96, 791, 836, 896
TimeoutExpired, 838
Timer (clase en *threading*), 780
Timer (clase en *timeit*), 1655
TimerHandle (clase en *asyncio*), 913
times() (en el módulo *os*), 603
TIMESTAMP (atributo de *py_compile.PycInvalidationMode*), 1837
timestamp() (método de *datetime.datetime*), 193
timetuple() (método de *datetime.date*), 184
timetuple() (método de *datetime.datetime*), 192
timetz() (método de *datetime.datetime*), 191
timezone (clase en *datetime*), 207
timezone (en el módulo *time*), 631
--timing
 trace command line option, 1660
tipado de pato, 1938
tipo, 1946
title() (en el módulo *turtle*), 1412
title() (método de *bytearray*), 65
title() (método de *bytes*), 65
title() (método de *str*), 51
Tix, 1459
tix_addbitmapdir() (método de *tkinter.tix.tixCommand*), 1463
tix_cget() (método de *tkinter.tix.tixCommand*), 1462
tix_configure() (método de *tkinter.tix.tixCommand*), 1462
tix_filedialog() (método de *tkinter.tix.tixCommand*), 1463
tix_getbitmap() (método de *tkinter.tix.tixCommand*), 1462
tix_getimage() (método de *tkinter.tix.tixCommand*), 1463
tix_option_get() (método de *tkinter.tix.tixCommand*), 1463
tix_resetoptions() (método de *tkinter.tix.tixCommand*), 1463
tixCommand (clase en *tkinter.tix*), 1462
Tk, 1429
Tk (clase en *tkinter*), 1430
Tk (clase en *tkinter.tix*), 1459
Tk Option Data Types, 1437
Tkinter, 1429
tkinter (módulo), 1429
tkinter.scrolledtext (módulo), 1463
tkinter.tix (módulo), 1459
tkinter.ttk (módulo), 1440
TList (clase en *tkinter.tix*), 1461
TLS, 973
TLSv1 (atributo de *ssl.TLSVersion*), 985
TLSv1_1 (atributo de *ssl.TLSVersion*), 985
TLSv1_2 (atributo de *ssl.TLSVersion*), 985
TLSv1_3 (atributo de *ssl.TLSVersion*), 985
TLSVersion (clase en *ssl*), 985
TMP, 406
TMPDIR, 406
to_bytes() (método de *int*), 33
to_eng_string() (método de *decimal.Context*), 314
to_eng_string() (método de *decimal.Decimal*), 308
to_integral() (método de *decimal.Decimal*), 308
to_integral_exact() (método de *decimal.Context*), 314
to_integral_exact() (método de *decimal.Decimal*), 308
to_integral_value() (método de *decimal.Decimal*), 308
to_sci_string() (método de *decimal.Context*), 314
ToASCII() (en el módulo *encodings.idna*), 175
tobuf() (método de *tarfile.TarInfo*), 501
tobytes() (método de *array.array*), 245
tobytes() (método de *memoryview*), 70
today() (método de clase de *datetime.date*), 183
today() (método de clase de *datetime.datetime*), 187
tofile() (método de *array.array*), 246
tok_name (en el módulo *token*), 1826
token (atributo de *shlex.shlex*), 1426
Token (clase en *contextvars*), 859
token (módulo), 1826
token_bytes() (en el módulo *secrets*), 555
token_hex() (en el módulo *secrets*), 555
token_urlsafe() (en el módulo *secrets*), 556
TokenError, 1831
tokenize (módulo), 1830
tokenize command line option
 -e, 1832
 --exact, 1832
 -h, 1832
 --help, 1832
tokenize() (en el módulo *tokenize*), 1830
tolist() (método de *array.array*), 246
tolist() (método de *memoryview*), 71
tolist() (método de *parser.ST*), 1816
tomono() (en el módulo *audioop*), 1349
toordinal() (método de *datetime.date*), 185
toordinal() (método de *datetime.datetime*), 193
top() (método de *curses.panel.Panel*), 727
top() (método de *poplib.POP3*), 1264
top_panel() (en el módulo *curses.panel*), 726
--top-level-directory directory
 unittest-discover command line
 option, 1523

- `toprettyxml()` (método `xml.dom.minidom.Node`), 1173
`toreadonly()` (método de `memoryview`), 71
`tostereo()` (en el módulo `audioop`), 1349
`tostring()` (en el módulo `xml.etree.ElementTree`), 1151
`tostring()` (método de `array.array`), 246
`tostringlist()` (en el módulo `xml.etree.ElementTree`), 1151
`total_changes` (atributo de `sqlite3.Connection`), 455
`total_ordering()` (en el módulo `functools`), 359
`total_seconds()` (método de `datetime.timedelta`), 182
`totuple()` (método de `parser.ST`), 1816
`touch()` (método de `pathlib.Path`), 388
`touchline()` (método de `curses.window`), 718
`touchwin()` (método de `curses.window`), 718
`ToUnicode()` (en el módulo `encodings.idna`), 175
`tounicode()` (método de `array.array`), 246
`towards()` (en el módulo `turtle`), 1394
`toxml()` (método de `xml.dom.minidom.Node`), 1173
`tparm()` (en el módulo `curses`), 711
`--trace`
 trace command line option, 1660
`Trace` (clase en `trace`), 1661
`Trace` (clase en `tracemalloc`), 1670
`trace` (módulo), 1659
`trace` command line option
 -c, 1660
 -C, 1660
 --count, 1660
 --coverdir=<dir>, 1660
 -f, 1660
 --file=<file>, 1660
 -g, 1660
 --help, 1659
 --ignore-dir=<dir>, 1660
 --ignore-module=<mod>, 1660
 -l, 1660
 --listfuncs, 1660
 -m, 1660
 --missing, 1660
 --no-report, 1660
 -r, 1660
 -R, 1660
 --report, 1660
 -s, 1660
 --summary, 1660
 -t, 1660
 -T, 1660
 --timing, 1660
 --trace, 1660
 --trackcalls, 1660
 --version, 1659
`trace` function, 770, 1699, 1706
`trace()` (en el módulo `inspect`), 1770
`trace_dispatch()` (método de `bdb.Bdb`), 1634
`traceback`
 objeto, 1695, 1747
`traceback` (atributo de `tracemalloc.Statistic`), 1669
`traceback` (atributo de `tracemalloc.StatisticDiff`), 1670
`traceback` (atributo de `tracemalloc.Trace`), 1670
`Traceback` (clase en `tracemalloc`), 1670
`traceback` (módulo), 1747
`traceback_limit` (atributo de `tracemalloc.Snapshot`), 1669
`traceback_limit` (atributo de `wsgi-ref.handlers.BaseHandler`), 1217
`TracebackException` (clase en `traceback`), 1749
`tracebacklimit` (en el módulo `sys`), 1709
`tracebacks`
 in CGI scripts, 1208
`TracebackType` (clase en `types`), 256
`tracemalloc` (módulo), 1662
`tracer()` (en el módulo `turtle`), 1407
`traces` (atributo de `tracemalloc.Snapshot`), 1669
`--trackcalls`
 trace command line option, 1660
`transfercmd()` (método de `ftplib.FTP`), 1260
`transient_internet()` (en el módulo `test.support`), 1619
`TransientResource` (clase en `test.support`), 1626
`translate()` (en el módulo `fnmatch`), 409
`translate()` (método de `bytearray`), 59
`translate()` (método de `bytes`), 59
`translate()` (método de `str`), 51
`translation()` (en el módulo `gettext`), 1369
`transport` (atributo de `asyncio.StreamWriter`), 881
`Transport` (clase en `asyncio`), 922
`Transport Layer Security`, 973
`Tree` (clase en `tkinter.tix`), 1461
`TreeBuilder` (clase en `xml.etree.ElementTree`), 1157
`Treeview` (clase en `tkinter.ttk`), 1452
`triangular()` (en el módulo `random`), 329
`true`, 29
`True`, 29, 85
`True` (variable incorporada), 27
`truediv()` (en el módulo `operator`), 367
`trunc()` (en el módulo `math`), 291
`trunc()` (in module `math`), 32
`truncate()` (en el módulo `os`), 591
`truncate()` (método de `io.IOBase`), 614
`truth`
 value, 29
`truth()` (en el módulo `operator`), 366
`try`
 sentencia, 89
`ttk`, 1440
`tty`
 I/O control, 1886
`tty` (módulo), 1888
`ttynamename()` (en el módulo `os`), 574
`tupla` nombrada, 1943
`tuple`
 objeto, 39, 42

`tuple` (clase incorporada), 42
`Tuple` (en el módulo `typing`), 1494
`tuple2st()` (en el módulo `parser`), 1815
`tuple_params` (2to3 fixer), 1611
`Turtle` (clase en `turtle`), 1412
`turtle` (módulo), 1385
`turtledemo` (módulo), 1416
`turtles()` (en el módulo `turtle`), 1411
`TurtleScreen` (clase en `turtle`), 1412
`turtlesize()` (en el módulo `turtle`), 1401
`type`
 `Boolean`, 6
 función incorporada, 84
 objeto, 24
 operations on dictionary, 77
 operations on list, 40
`type` (atributo de `optparse.Option`), 1914
`type` (atributo de `socket.socket`), 969
`type` (atributo de `tarfile.TarInfo`), 502
`type` (atributo de `urllib.request.Request`), 1225
`Type` (clase en `typing`), 1485
`type` (clase incorporada), 24
`type_check_only()` (en el módulo `typing`), 1492
`TYPE_CHECKER` (atributo de `optparse.Option`), 1925
`TYPE_CHECKING` (en el módulo `typing`), 1495
`TYPE_COMMENT` (en el módulo `token`), 1829
`TYPE_IGNORE` (en el módulo `token`), 1829
`typeahead()` (en el módulo `curses`), 711
`typecode` (atributo de `array.array`), 244
`typecodes` (en el módulo `array`), 244
`TYPED_ACTIONS` (atributo de `optparse.Option`), 1927
`typed_subpart_iterator()` (en el módulo `email.iterators`), 1095
`TypedDict` (clase en `typing`), 1490
`TypeError`, 94
`types`
 built-in, 29
 immutable sequence, 39
 módulo, 84
 mutable sequence, 40
 operations on integer, 32
 operations on mapping, 77
 operations on numeric, 31
 operations on sequence, 38, 40
`types` (2to3 fixer), 1611
`TYPES` (atributo de `optparse.Option`), 1925
`types` (módulo), 253
`types_map` (atributo de `mimetypes.MimeTypes`), 1126
`types_map` (en el módulo `mimetypes`), 1125
`types_map_inv` (atributo de `mimetypes.MimeTypes`), 1126
`TypeVar` (clase en `typing`), 1483
`typing` (módulo), 1477
`TZ`, 628, 629
`tzinfo` (atributo de `datetime.datetime`), 190
`tzinfo` (atributo de `datetime.time`), 198
`tzinfo` (clase en `datetime`), 201
`tzname` (en el módulo `time`), 631

`tzname()` (método de `datetime.datetime`), 192
`tzname()` (método de `datetime.time`), 200
`tzname()` (método de `datetime.timezone`), 207
`tzname()` (método de `datetime.tzinfo`), 202
`tzset()` (en el módulo `time`), 628

U

`-u`
 timeit command line option, 1657
`ucd_3_2_0` (en el módulo `unicodedata`), 145
`udata` (atributo de `select.kevent`), 1013
`UDPServer` (clase en `socketserver`), 1294
`UF_APPEND` (en el módulo `stat`), 400
`UF_COMPRESSED` (en el módulo `stat`), 400
`UF_HIDDEN` (en el módulo `stat`), 400
`UF_IMMUTABLE` (en el módulo `stat`), 400
`UF_NODUMP` (en el módulo `stat`), 400
`UF_NOUNLINK` (en el módulo `stat`), 400
`UF_OPAQUE` (en el módulo `stat`), 400
`uid` (atributo de `tarfile.TarInfo`), 502
`UID` (clase en `plistlib`), 541
`uid()` (método de `imaplib.IMAP4`), 1270
`uidl()` (método de `poplib.POP3`), 1264
`u-LAW`, 1347, 1352, 1361
`ulaw2lin()` (en el módulo `audioop`), 1349
`umask()` (en el módulo `os`), 565
`unalias` (`pdb` command), 1645
`uname` (atributo de `tarfile.TarInfo`), 502
`uname()` (en el módulo `os`), 565
`uname()` (en el módulo `platform`), 729
`UNARY_INVERT` (opcode), 1845
`UNARY_NEGATIVE` (opcode), 1845
`UNARY_NOT` (opcode), 1845
`UNARY_POSITIVE` (opcode), 1845
`UnboundLocalError`, 94
`unbuffered I/O`, 19
`UNC paths`
 and `os.makedirs()`, 580
`UNCHECKED_HASH` (atributo de `py_compile.PycInvalidationMode`), 1837
`unconsumed_tail` (atributo de `zlib.Decompress`), 472
`unctrl()` (en el módulo `curses`), 711
`unctrl()` (en el módulo `curses.ascii`), 725
`Underflow` (clase en `decimal`), 316
`undisplay` (`pdb` command), 1645
`undo()` (en el módulo `turtle`), 1393
`undobufferentries()` (en el módulo `turtle`), 1405
`undoc_header` (atributo de `cmd.Cmd`), 1420
`unescape()` (en el módulo `html`), 1135
`unescape()` (en el módulo `xml.sax.saxutils`), 1184
`UnexpectedException`, 1518
`unexpectedSuccesses` (atributo de `unittest.TestResult`), 1541
`unfreeze()` (en el módulo `gc`), 1756
`unget_wch()` (en el módulo `curses`), 711
`ungetch()` (en el módulo `curses`), 711
`ungetch()` (en el módulo `msvcrt`), 1868

- `ungetmouse()` (en el módulo *curses*), 711
- `ungetwch()` (en el módulo *msvcrt*), 1868
- `unhexlify()` (en el módulo *binascii*), 1132
- `Unicode`, 144, 158
 - database, 144
- `unicode (2to3 fixer)`, 1611
- `unicodedata` (módulo), 144
- `UnicodeDecodeError`, 94
- `UnicodeEncodeError`, 94
- `UnicodeError`, 94
- `UnicodeTranslateError`, 94
- `UnicodeWarning`, 96
- `unidata_version` (en el módulo *unicodedata*), 145
- `unified_diff()` (en el módulo *difflib*), 133
- `uniform()` (en el módulo *random*), 329
- `UnimplementedFileMode`, 1252
- `Union` (clase en *ctypes*), 766
- `Union` (en el módulo *typing*), 1493
- `union()` (método de *frozenset*), 76
- `unique()` (en el módulo *enum*), 270
- `--unit=U`
 - timeit command line option, 1657
- `unittest` (módulo), 1519
- `unittest` command line option
 - `-b`, 1522
 - `--buffer`, 1522
 - `-c`, 1522
 - `--catch`, 1522
 - `-f`, 1522
 - `--failfast`, 1522
 - `-k`, 1522
 - `--locals`, 1522
- `unittest-discover` command line option
 - `-p`, 1522
 - `--pattern` pattern, 1522
 - `-s`, 1522
 - `--start-directory` directory, 1522
 - `-t`, 1523
 - `--top-level-directory` directory, 1523
 - `-v`, 1522
 - `--verbose`, 1522
- `unittest.mock` (módulo), 1548
- `universal newlines`
 - `bytearray.splitlines` method, 64
 - `bytes.splitlines` method, 64
 - `csv.reader` function, 512
 - `importlib.abc.InspectLoader.get_source` method, 1795
 - `io.IncrementalNewlineDecoder` class, 621
 - `io.TextIOWrapper` class, 620
 - `open()` built-in function, 18
 - `str.splitlines` method, 50
 - `subprocess` module, 839
- `UNIX`
 - file control, 1889
 - I/O control, 1889
- `unix_dialect` (clase en *csv*), 514
- `unix_shell` (en el módulo *test.support*), 1615
- `UnixDatagramServer` (clase en *socketserver*), 1294
- `UnixStreamServer` (clase en *socketserver*), 1294
- `unknown` (atributo de *uuid.SafeUUID*), 1291
- `unknown_decl()` (método de *html.parser.HTMLParser*), 1138
- `unknown_open()` (método de *urllib.request.BaseHandler*), 1228
- `unknown_open()` (método de *urllib.request.UnknownHandler*), 1232
- `UnknownHandler` (clase en *urllib.request*), 1224
- `UnknownProtocol`, 1252
- `UnknownTransferEncoding`, 1252
- `unlink()` (en el módulo *os*), 591
- `unlink()` (en el módulo *test.support*), 1616
- `unlink()` (método de *multiprocessing.shared_memory.SharedMemory*), 826
- `unlink()` (método de *pathlib.Path*), 388
- `unlink()` (método de *xml.dom.minidom.Node*), 1172
- `unload()` (en el módulo *test.support*), 1616
- `unlock()` (método de *mailbox.Babyl*), 1114
- `unlock()` (método de *mailbox.Mailbox*), 1109
- `unlock()` (método de *mailbox.Maildir*), 1111
- `unlock()` (método de *mailbox.mbox*), 1111
- `unlock()` (método de *mailbox.MH*), 1113
- `unlock()` (método de *mailbox.MMDF*), 1114
- `unpack()` (en el módulo *struct*), 154
- `unpack()` (método de *struct.Struct*), 158
- `unpack_archive()` (en el módulo *shutil*), 418
- `unpack_array()` (método de *xdrlib.Unpacker*), 538
- `unpack_bytes()` (método de *xdrlib.Unpacker*), 538
- `unpack_double()` (método de *xdrlib.Unpacker*), 538
- `UNPACK_EX` (opcode), 1850
- `unpack_farray()` (método de *xdrlib.Unpacker*), 538
- `unpack_float()` (método de *xdrlib.Unpacker*), 538
- `unpack_fopaque()` (método de *xdrlib.Unpacker*), 538
- `unpack_from()` (en el módulo *struct*), 154
- `unpack_from()` (método de *struct.Struct*), 158
- `unpack_fstring()` (método de *xdrlib.Unpacker*), 538
- `unpack_list()` (método de *xdrlib.Unpacker*), 538
- `unpack_opaque()` (método de *xdrlib.Unpacker*), 538
- `UNPACK_SEQUENCE` (opcode), 1850
- `unpack_string()` (método de *xdrlib.Unpacker*), 538
- `Unpacker` (clase en *xdrlib*), 536
- `unparsedEntityDecl()` (método de *xml.sax.handler.DTDHandler*), 1183
- `UnparsedEntityDeclHandler()` (método de *xml.parsers.expat.xmlparser*), 1193
- `Unpickler` (clase en *pickle*), 426
- `UnpicklingError`, 424
- `unquote()` (en el módulo *email.utils*), 1092
- `unquote()` (en el módulo *urllib.parse*), 1245
- `unquote_plus()` (en el módulo *urllib.parse*), 1245

- `unquote_to_bytes()` (en el módulo `urllib.parse`), 1245
- `unraisablehook()` (en el módulo `sys`), 1709
- `unregister()` (en el módulo `atexit`), 1746
- `unregister()` (en el módulo `faulthandler`), 1639
- `unregister()` (método de `select.devpoll`), 1009
- `unregister()` (método de `select.epoll`), 1010
- `unregister()` (método de `selectors.BaseSelector`), 1015
- `unregister()` (método de `select.poll`), 1011
- `unregister_archive_format()` (en el módulo `shutil`), 418
- `unregister_dialect()` (en el módulo `csv`), 512
- `unregister_unpack_format()` (en el módulo `shutil`), 419
- `unsafe` (atributo de `uuid.SafeUUID`), 1291
- `unset()` (método de `test.support.EnvironmentVarGuard`), 1626
- `unsetenv()` (en el módulo `os`), 566
- `UnstructuredHeader` (clase en `email.headerregistry`), 1062
- `unsubscribe()` (método de `imaplib.IMAP4`), 1271
- `UnsupportedOperation`, 611
- `until` (*pdb* command), 1644
- `untokenize()` (en el módulo `tokenize`), 1831
- `untouchwin()` (método de `curses.window`), 718
- `unused_data` (atributo de `bz2.BZ2Decompressor`), 478
- `unused_data` (atributo de `lzma.LZMADecompressor`), 483
- `unused_data` (atributo de `zlib.Decompress`), 472
- `unverifiable` (atributo de `urllib.request.Request`), 1225
- `unwrap()` (en el módulo `inspect`), 1769
- `unwrap()` (en el módulo `urllib.parse`), 1242
- `unwrap()` (método de `ssl.SSLSocket`), 988
- `up` (*pdb* command), 1643
- `up()` (en el módulo `turtle`), 1396
- `update()` (en el módulo `turtle`), 1408
- `update()` (método de `collections.Counter`), 221
- `update()` (método de `dict`), 80
- `update()` (método de `frozenset`), 77
- `update()` (método de `hashlib.hash`), 545
- `update()` (método de `hmac.HMAC`), 554
- `update()` (método de `http.cookies.Morsel`), 1310
- `update()` (método de `mailbox.Mailbox`), 1109
- `update()` (método de `mailbox.Maildir`), 1110
- `update()` (método de `trace.CoverageResults`), 1661
- `update_authenticated()` (método de `urllib.request.HTTPPasswordMgrWithPriorAuth`), 1230
- `update_lines_cols()` (en el módulo `curses`), 711
- `update_panels()` (en el módulo `curses.panel`), 726
- `update_visible()` (método de `mailbox.BabylMessage`), 1120
- `update_wrapper()` (en el módulo `functools`), 364
- `upper()` (método de `bytearray`), 65
- `upper()` (método de `bytes`), 65
- `upper()` (método de `str`), 52
- `urandom()` (en el módulo `os`), 609
- `URL`, 1202, 1238, 1247, 1302
 - `parsing`, 1238
 - `relative`, 1238
- `url` (atributo de `xmlrpc.client.ProtocolError`), 1325
- `url2pathname()` (en el módulo `urllib.request`), 1221
- `urlcleanup()` (en el módulo `urllib.request`), 1235
- `urldefrag()` (en el módulo `urllib.parse`), 1242
- `urlencode()` (en el módulo `urllib.parse`), 1245
- `URLError`, 1246
- `urljoin()` (en el módulo `urllib.parse`), 1242
- `urllib (2to3 fixer)`, 1611
- `urllib` (módulo), 1219
- `urllib.error` (módulo), 1246
- `urllib.parse` (módulo), 1238
- `urllib.request` (módulo), 1250
- `urllib.request` (módulo), 1220
- `urllib.response` (módulo), 1238
- `urllib.robotparser` (módulo), 1247
- `urlopen()` (en el módulo `urllib.request`), 1220
- `URLopener` (clase en `urllib.request`), 1235
- `urlparse()` (en el módulo `urllib.parse`), 1238
- `urlretrieve()` (en el módulo `urllib.request`), 1235
- `urlsafe_b64decode()` (en el módulo `base64`), 1128
- `urlsafe_b64encode()` (en el módulo `base64`), 1127
- `urlsplit()` (en el módulo `urllib.parse`), 1241
- `urlunparse()` (en el módulo `urllib.parse`), 1241
- `urlunsplit()` (en el módulo `urllib.parse`), 1242
- `urn` (atributo de `uuid.UUID`), 1292
- `use_default_colors()` (en el módulo `curses`), 711
- `use_env()` (en el módulo `curses`), 711
- `use_rawinput` (atributo de `cmd.Cmd`), 1420
- `UseForeignDTD()` (método de `xml.parsers.expat.xmlparser`), 1190
- `user`
 - `effective id`, 562
 - `id`, 563
 - `id`, `setting`, 565
- `USER`, 705
- `user()` (método de `poplib.POP3`), 1263
- `USER_BASE` (en el módulo `site`), 1775
- `user_call()` (método de `bdb.Bdb`), 1635
- `user_exception()` (método de `bdb.Bdb`), 1635
- `user_line()` (método de `bdb.Bdb`), 1635
- `user_return()` (método de `bdb.Bdb`), 1635
- `USER_SITE` (en el módulo `site`), 1775
- `--user-base`
 - `site` command line option, 1776
- `usercustomize` (módulo), 1774
- `UserDict` (clase en `collections`), 232
- `UserList` (clase en `collections`), 232
- `USERNAME`, 562, 705

- username (atributo de *email.headerregistry.Address*), 1066
- USERPROFILE, 391
- userptr() (método de *curses.panel.Panel*), 727
- user-site
site command line option, 1776
- UserString (clase en *collections*), 233
- UserWarning, 96
- USTAR_FORMAT (en el módulo *tarfile*), 497
- UTC, 622
- utc (atributo de *datetime.timezone*), 208
- utcfromtimestamp() (método de clase de *datetime.datetime*), 188
- utcnow() (método de clase de *datetime.datetime*), 188
- utcoffset() (método de *datetime.datetime*), 192
- utcoffset() (método de *datetime.time*), 200
- utcoffset() (método de *datetime.timezone*), 207
- utcoffset() (método de *datetime.tzinfo*), 201
- utctimetuple() (método de *datetime.datetime*), 193
- utf8 (atributo de *email.policy.EmailPolicy*), 1057
- utf8() (método de *poplib.POP3*), 1264
- utf8_enabled (atributo de *imaplib.IMAP4*), 1271
- utime() (en el módulo *os*), 591
- uu
módulo, 1130
- uu (módulo), 1133
- UUID (clase en *uuid*), 1291
- uuid (módulo), 1291
- uuid1, 1293
- uuid1() (en el módulo *uuid*), 1292
- uuid3, 1293
- uuid3() (en el módulo *uuid*), 1293
- uuid4, 1293
- uuid4() (en el módulo *uuid*), 1293
- uuid5, 1293
- uuid5() (en el módulo *uuid*), 1293
- UuidCreate() (en el módulo *msilib*), 1861
- ## V
- v
tarfile command line option, 507
timeit command line option, 1657
unittest-discover command line option, 1522
- v4_int_to_packed() (en el módulo *ipaddress*), 1345
- v6_int_to_packed() (en el módulo *ipaddress*), 1345
- valid_signals() (en el módulo *signal*), 1027
- validator() (en el módulo *wsgiref.validate*), 1214
- value
truth, 29
- value (atributo de *ctypes._SimpleCData*), 763
- value (atributo de *http.cookiejar.Cookie*), 1318
- value (atributo de *http.cookies.Morsel*), 1310
- value (atributo de *xml.dom.Attr*), 1168
- Value() (en el módulo *multiprocessing*), 800
- Value() (en el módulo *multiprocessing.sharedctypes*), 801
- Value() (método de *multiprocessing.managers.SyncManager*), 804
- value_decode() (método de *http.cookies.BaseCookie*), 1309
- value_encode() (método de *http.cookies.BaseCookie*), 1309
- ValueError, 94
- valuerefs() (método de *kref.WeakValueDictionary*), 248
- values
Boolean, 85
- values() (método de *contextvars.Context*), 861
- values() (método de *dict*), 80
- values() (método de *email.message.EmailMessage*), 1041
- values() (método de *email.message.Message*), 1079
- values() (método de *mailbox.Mailbox*), 1108
- values() (método de *types.MappingProxyType*), 257
- ValuesView (clase en *collections.abc*), 235
- ValuesView (clase en *typing*), 1487
- var (atributo de *contextvars.Token*), 859
- variable de clase, 1937
- variable de contexto, 1937
- variables de entorno
AUDIODEV, 1362
BROWSER, 1199, 1200
COLS, 711
COLUMNS, 711
COMSPEC, 603, 841
HOME, 391
HOMEDRIVE, 391
HOMEPATH, 391
http_proxy, 1220, 1233
IDLESTARTUP, 1471
KDEDIR, 1201
LANG, 1367, 1369, 1377, 1379
LANGUAGE, 1367, 1369
LC_ALL, 1367, 1369
LC_MESSAGES, 1367, 1369
LINES, 707, 711
LNAME, 705
LOGNAME, 562, 705
MIXERDEV, 1362
no_proxy, 1223
PAGER, 1496
PATH, 596, 597, 601, 602, 608, 1199, 1206, 1208, 1773
POSIXLY_CORRECT, 663
PYTHON_DOM, 1162
PYTHONASYNCIODEBUG, 912, 947
PYTHONBREAKPOINT, 1693
PYTHONDEVMODE, 1628
PYTHONDOCS, 1497
PYTHONDONTWRITEBYTECODE, 1694
PYTHONFAULTHANDLER, 1638
PYTHONHOME, 1627

- PYTHONINTMAXSTRDIGITS, 87, 1702
 PYTHONIOENCODING, 1708
 PYTHONLEGACYWINDOWSFSENCODING, 1707
 PYTHONLEGACYWINDOWSTDIO, 1708
 PYTHONNOUSERSITE, 1775
 PYTHONPATH, 1206, 1627, 1703
 PYTHONPYCACHEPREFIX, 1694
 PYTHONSTARTUP, 150, 1471, 1702, 1774
 PYTHONTRACEMALLOC, 1662, 1666
 PYTHONTRACEMALLOC`comienzo,
 configura la variable del
 entorno a ``25`, 1662
 PYTHONUSERBASE, 1775
 PYTHONUSERSITE, 1627
 PYTHONUTF8, 1708
 PYTHONWARNINGS, 1717
 SOURCE_DATE_EPOCH, 1837, 1839
 SSL_CERT_FILE, 1006
 SSL_CERT_PATH, 1006
 SSLKEYLOGFILE, 975
 SystemRoot, 842
 TEMP, 406
 TERM, 710
 TMP, 406
 TMPDIR, 406
 TZ, 628, 629
 USER, 705
 USERNAME, 562, 705
 USERPROFILE, 391
 VIRTUAL_ENV, 1677
 variance (atributo de *statistics.NormalDist*), 339
 variance() (en el módulo *statistics*), 337
 variant (atributo de *uuid.UUID*), 1292
 vars() (función incorporada), 24
 vbar (atributo de *tkinter.scrolledtext.ScrolledText*), 1464
 VBAR (en el módulo *token*), 1827
 VBAREQUAL (en el módulo *token*), 1828
 Vec2D (clase en *turtle*), 1413
 venv (módulo), 1675
 --verbose
 tarfile command line option, 507
 timeit command line option, 1657
 unittest-discover command line
 option, 1522
 VERBOSE (en el módulo *re*), 117
 verbose (en el módulo *tabnanny*), 1834
 verbose (en el módulo *test.support*), 1615
 verify() (método de *smtplib.SMTP*), 1281
 verify_client_post_handshake() (método
 de *ssl.SSLSocket*), 988
 verify_code (atributo de
 ssl.SSLCertVerificationError), 976
 VERIFY_CRL_CHECK_CHAIN (en el módulo *ssl*), 980
 VERIFY_CRL_CHECK_LEAF (en el módulo *ssl*), 980
 VERIFY_DEFAULT (en el módulo *ssl*), 980
 verify_flags (atributo de *ssl.SSLContext*), 997
 verify_generated_headers (atributo de
 email.policy.Policy), 1055
 verify_message (atributo de
 ssl.SSLCertVerificationError), 976
 verify_mode (atributo de *ssl.SSLContext*), 997
 verify_request() (método de *socketserver.BaseServer*), 1298
 VERIFY_X509_STRICT (en el módulo *ssl*), 980
 VERIFY_X509_TRUSTED_FIRST (en el módulo *ssl*),
 980
 VerifyFlags (clase en *ssl*), 980
 VerifyMode (clase en *ssl*), 980
 --version
 trace command line option, 1659
 version (atributo de
 email.headerregistry.MIMEVersionHeader),
 1064
 version (atributo de *http.client.HTTPResponse*), 1255
 version (atributo de *http.cookiejar.Cookie*), 1318
 version (atributo de *ipaddress.IPv4Address*), 1334
 version (atributo de *ipaddress.IPv4Network*), 1339
 version (atributo de *ipaddress.IPv6Address*), 1336
 version (atributo de *ipaddress.IPv6Network*), 1342
 version (atributo de *urllib.request.URLopener*), 1236
 version (atributo de *uuid.UUID*), 1292
 version (en el módulo *curses*), 718
 version (en el módulo *marshal*), 442
 version (en el módulo *sqlite3*), 448
 version (en el módulo *sys*), 1709
 version() (en el módulo *ensurepip*), 1675
 version() (en el módulo *platform*), 729
 version() (método de *ssl.SSLSocket*), 988
 version_info (en el módulo *sqlite3*), 448
 version_info (en el módulo *sys*), 1710
 version_string() (método de
 http.server.BaseHTTPRequestHandler),
 1305
 vformat() (método de *string.Formatter*), 100
 virtual
 Environments, 1675
 VIRTUAL_ENV, 1677
 visit() (método de *ast.NodeVisitor*), 1822
 vista de diccionario, 1938
 vline() (método de *curses.window*), 718
 voidcmd() (método de *ftplib.FTP*), 1260
 volume (atributo de *zipfile.ZipInfo*), 493
 vonmisesvariate() (en el módulo *random*), 329
- ## W
- W_OK (en el módulo *os*), 576
 wait() (en el módulo *asyncio*), 873
 wait() (en el módulo *concurrent.futures*), 835
 wait() (en el módulo *multiprocessing.connection*), 813
 wait() (en el módulo *os*), 603
 wait() (método de
 asyncio.subprocess.Process), 891
 wait() (método de *asyncio.Condition*), 887
 wait() (método de *asyncio.Event*), 886
 wait() (método de *multiprocessing.pool.AsyncResult*),
 811

- `wait()` (método de `subprocess.Popen`), 844
- `wait()` (método de `threading.Barrier`), 781
- `wait()` (método de `threading.Condition`), 777
- `wait()` (método de `threading.Event`), 779
- `wait3()` (en el módulo `os`), 604
- `wait4()` (en el módulo `os`), 605
- `wait_closed()` (método de `asyncio.Server`), 915
- `wait_closed()` (método de `asyncio.StreamWriter`), 882
- `wait_for()` (en el módulo `asyncio`), 872
- `wait_for()` (método de `asyncio.Condition`), 888
- `wait_for()` (método de `threading.Condition`), 777
- `wait_threads_exit()` (en el módulo `test.support`), 1620
- `waitid()` (en el módulo `os`), 604
- `waitpid()` (en el módulo `os`), 604
- `walk()` (en el módulo `ast`), 1822
- `walk()` (en el módulo `os`), 592
- `walk()` (método de `email.message.EmailMessage`), 1043
- `walk()` (método de `email.message.Message`), 1082
- `walk_packages()` (en el módulo `pkgutil`), 1785
- `walk_stack()` (en el módulo `traceback`), 1749
- `walk_tb()` (en el módulo `traceback`), 1749
- `want` (atributo de `doctest.Example`), 1512
- `warn()` (en el módulo `warnings`), 1720
- `warn_explicit()` (en el módulo `warnings`), 1720
- `Warning`, 96, 460
- `warning()` (en el módulo `logging`), 677
- `warning()` (método de `logging.Logger`), 668
- `warning()` (método de `xml.sax.handler.ErrorHandler`), 1183
- `warnings`, 1715
- `warnings` (módulo), 1715
- `WarningsRecorder` (clase en `test.support`), 1627
- `warnoptions` (en el módulo `sys`), 1710
- `wasSuccessful()` (método de `unittest.TestResult`), 1542
- `WatchedFileHandler` (clase en `logging.handlers`), 693
- `wave` (módulo), 1355
- `WCONTINUED` (en el módulo `os`), 605
- `WCOREDUMP()` (en el módulo `os`), 605
- `WeakKeyDictionary` (clase en `weakref`), 248
- `WeakMethod` (clase en `weakref`), 248
- `weakref` (módulo), 246
- `WeakSet` (clase en `weakref`), 248
- `WeakValueDictionary` (clase en `weakref`), 248
- `webbrowser` (módulo), 1199
- `weekday()` (en el módulo `calendar`), 216
- `weekday()` (método de `datetime.date`), 185
- `weekday()` (método de `datetime.datetime`), 193
- `weekheader()` (en el módulo `calendar`), 216
- `weibullvariate()` (en el módulo `random`), 329
- `WEXITED` (en el módulo `os`), 604
- `WEXITSTATUS()` (en el módulo `os`), 606
- `wfile` (atributo de `http.server.BaseHTTPRequestHandler`), 1303
- `what()` (en el módulo `imghdr`), 1360
- `what()` (en el módulo `sndhdr`), 1361
- `whathdr()` (en el módulo `sndhdr`), 1361
- `whatis` (`pdb` command), 1645
- `when()` (método de `asyncio.TimerHandle`), 914
- `where` (`pdb` command), 1642
- `which()` (en el módulo `shutil`), 415
- `whichdb()` (en el módulo `dbm`), 442
- `while`
 - sentencia, 29
- `whitespace` (atributo de `shlex.shlex`), 1425
- `whitespace` (en el módulo `string`), 100
- `whitespace_split` (atributo de `shlex.shlex`), 1426
- `Widget` (clase en `tkinter.ttk`), 1443
- `width` (atributo de `textwrap.TextWrapper`), 142
- `width()` (en el módulo `turtle`), 1396
- `WIFCONTINUED()` (en el módulo `os`), 605
- `WIFEXITED()` (en el módulo `os`), 605
- `WIFSIGNALED()` (en el módulo `os`), 605
- `WIFSTOPPED()` (en el módulo `os`), 605
- `win32_edition()` (en el módulo `platform`), 729
- `win32_is_iot()` (en el módulo `platform`), 729
- `win32_ver()` (en el módulo `platform`), 729
- `WinDLL` (clase en `ctypes`), 754
- `window manager` (widgets), 1436
- `window()` (método de `curses.panel.Panel`), 727
- `window_height()` (en el módulo `turtle`), 1411
- `window_width()` (en el módulo `turtle`), 1411
- `Windows ini file`, 518
- `WindowsError`, 95
- `WindowsPath` (clase en `pathlib`), 382
- `WindowsProactorEventLoopPolicy` (clase en `asyncio`), 936
- `WindowsRegistryFinder` (clase en `importlib.machinery`), 1800
- `WindowsSelectorEventLoopPolicy` (clase en `asyncio`), 936
- `winerror` (atributo de `OSError`), 92
- `WinError()` (en el módulo `ctypes`), 762
- `WINFUNCTYPE()` (en el módulo `ctypes`), 758
- `winreg` (módulo), 1868
- `WinSock`, 1008
- `winsound` (módulo), 1877
- `winver` (en el módulo `sys`), 1710
- `WITH_CLEANUP_FINISH` (opcode), 1849
- `WITH_CLEANUP_START` (opcode), 1849
- `with_hostmask` (atributo de `ipaddress.IPv4Interface`), 1344
- `with_hostmask` (atributo de `ipaddress.IPv4Network`), 1339
- `with_hostmask` (atributo de `ipaddress.IPv6Interface`), 1344
- `with_hostmask` (atributo de `ipaddress.IPv6Network`), 1342
- `with_name()` (método de `pathlib.PurePath`), 381
- `with_netmask` (atributo de `ipaddress.IPv4Interface`), 1344

- `with_netmask` (atributo de `ipaddress.IPv4Network`), 1339
- `with_netmask` (atributo de `ipaddress.IPv6Interface`), 1344
- `with_netmask` (atributo de `ipaddress.IPv6Network`), 1342
- `with_prefixlen` (atributo de `ipaddress.IPv4Interface`), 1344
- `with_prefixlen` (atributo de `ipaddress.IPv4Network`), 1339
- `with_prefixlen` (atributo de `ipaddress.IPv6Interface`), 1344
- `with_prefixlen` (atributo de `ipaddress.IPv6Network`), 1342
- `with_pymalloc` () (en el módulo `test.support`), 1616
- `with_suffix` () (método de `pathlib.PurePath`), 381
- `with_traceback` () (método de `BaseException`), 90
- `WNOHANG` (en el módulo `os`), 605
- `WNOWAIT` (en el módulo `os`), 604
- `wordchars` (atributo de `shlex.shlex`), 1425
- World Wide Web, 1199, 1238, 1247
- `wrap` () (en el módulo `textwrap`), 140
- `wrap` () (método de `textwrap.TextWrapper`), 143
- `wrap_bio` () (método de `ssl.SSLContext`), 994
- `wrap_future` () (en el módulo `asyncio`), 918
- `wrap_socket` () (en el módulo `ssl`), 979
- `wrap_socket` () (método de `ssl.SSLContext`), 994
- `wrapper` () (en el módulo `curses`), 711
- `WrapperDescriptorType` (en el módulo `types`), 255
- `wraps` () (en el módulo `functools`), 364
- `WRITABLE` (en el módulo `tkinter`), 1440
- `writable` () (método de `asyncore.dispatcher`), 1019
- `writable` () (método de `io.IOBase`), 614
- `write` () (en el módulo `os`), 574
- `write` () (en el módulo `turtle`), 1399
- `write` () (método de `asyncio.StreamWriter`), 881
- `write` () (método de `asyncio.WriteTransport`), 925
- `write` () (método de `codecs.StreamWriter`), 165
- `write` () (método de `code.InteractiveInterpreter`), 1778
- `write` () (método de `configparser.ConfigParser`), 533
- `write` () (método de `email.generator.BytesGenerator`), 1051
- `write` () (método de `email.generator.Generator`), 1052
- `write` () (método de `io.BufferedIOBase`), 616
- `write` () (método de `io.BufferedWriter`), 618
- `write` () (método de `io.RawIOBase`), 615
- `write` () (método de `io.TextIOBase`), 619
- `write` () (método de `mmap.mmap`), 1034
- `write` () (método de `ossaudiodev.oss_audio_device`), 1362
- `write` () (método de `ssl.MemoryBIO`), 1004
- `write` () (método de `ssl.SSLSocket`), 986
- `write` () (método de `telnetlib.Telnet`), 1290
- `write` () (método de `xml.etree.ElementTree.ElementTree`), 1156
- `write` () (método de `zipfile.ZipFile`), 489
- `write_byte` () (método de `mmap.mmap`), 1034
- `write_bytes` () (método de `pathlib.Path`), 388
- `write_docstringdict` () (en el módulo `turtle`), 1415
- `write_eof` () (método de `asyncio.StreamWriter`), 881
- `write_eof` () (método de `asyncio.WriteTransport`), 925
- `write_eof` () (método de `ssl.MemoryBIO`), 1004
- `write_history_file` () (en el módulo `readline`), 148
- `write_results` () (método de `trace.CoverageResults`), 1661
- `write_text` () (método de `pathlib.Path`), 388
- `write_through` (atributo de `io.TextIOWrapper`), 620
- `writeall` () (método de `ossaudiodev.oss_audio_device`), 1362
- `writelines` () (método de `aifc.aifc`), 1352
- `writelines` () (método de `sunau.AU_write`), 1355
- `writelines` () (método de `wave.Wave_write`), 1357
- `writelinesraw` () (método de `aifc.aifc`), 1352
- `writelinesraw` () (método de `sunau.AU_write`), 1355
- `writelinesraw` () (método de `wave.Wave_write`), 1357
- `writeheader` () (método de `csv.DictWriter`), 516
- `writelines` () (método de `asyncio.StreamWriter`), 881
- `writelines` () (método de `asyncio.WriteTransport`), 925
- `writelines` () (método de `codecs.StreamWriter`), 165
- `writelines` () (método de `io.IOBase`), 614
- `writePlist` () (en el módulo `plistlib`), 540
- `writePlistToBytes` () (en el módulo `plistlib`), 541
- `writepy` () (método de `zipfile.PyZipFile`), 491
- `writer` (atributo de `formatter.formatter`), 1858
- `writer` () (en el módulo `csv`), 512
- `writerow` () (método de `csv.csvwriter`), 516
- `writerows` () (método de `csv.csvwriter`), 516
- `writestr` () (método de `zipfile.ZipFile`), 489
- `WriteTransport` (clase en `asyncio`), 922
- `writev` () (en el módulo `os`), 574
- `writexml` () (método de `xml.dom.minidom.Node`), 1173
- `WrongDocumentErr`, 1170
- `ws_comma` (2to3 fixer), 1611
- `wsgi_file_wrapper` (atributo de `wsgi-ref.handlers.BaseHandler`), 1217
- `wsgi_multiprocess` (atributo de `wsgi-ref.handlers.BaseHandler`), 1216
- `wsgi_multithread` (atributo de `wsgi-ref.handlers.BaseHandler`), 1216
- `wsgi_run_once` (atributo de `wsgi-ref.handlers.BaseHandler`), 1216
- `wsgiref` (módulo), 1209
- `wsgiref.handlers` (módulo), 1215
- `wsgiref.headers` (módulo), 1211
- `wsgiref.simple_server` (módulo), 1212
- `wsgiref.util` (módulo), 1210
- `wsgiref.validate` (módulo), 1214

WSGIRequestHandler (clase en `wsgiref.simple_server`), 1213
 WSGIServer (clase en `wsgiref.simple_server`), 1213
 wShowWindow (atributo de `subprocess.STARTUPINFO`), 846
 WSTOPPED (en el módulo `os`), 604
 WSTOPSIG() (en el módulo `os`), 606
 wstring_at() (en el módulo `ctypes`), 762
 WTERMSIG() (en el módulo `os`), 606
 WUNTRACED (en el módulo `os`), 605
 WWW, 1199, 1238, 1247
 server, 1202, 1302

X

X (en el módulo `re`), 117

-x regex
 compileall command line option, 1838

X509 certificate, 997

X_OK (en el módulo `os`), 576

xatom() (método de `imaplib.IMAP4`), 1271

XATTR_CREATE (en el módulo `os`), 595

XATTR_REPLACE (en el módulo `os`), 595

XATTR_SIZE_MAX (en el módulo `os`), 595

xcor() (en el módulo `turtle`), 1394

XDR, 536

xdrlib (módulo), 536

xhdr() (método de `nnplib.NNTP`), 1277

XHTML, 1136

XHTML_NAMESPACE (en el módulo `xml.dom`), 1162

xml (módulo), 1140

XML() (en el módulo `xml.etree.ElementTree`), 1152

XML_ERROR_ABORTED (en el módulo `xml.parsers.expat.errors`), 1198

XML_ERROR_ASYNC_ENTITY (en el módulo `xml.parsers.expat.errors`), 1196

XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF (en el módulo `xml.parsers.expat.errors`), 1196

XML_ERROR_BAD_CHAR_REF (en el módulo `xml.parsers.expat.errors`), 1196

XML_ERROR_BINARY_ENTITY_REF (en el módulo `xml.parsers.expat.errors`), 1196

XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_DUPLICATE_ATTRIBUTE (en el módulo `xml.parsers.expat.errors`), 1196

XML_ERROR_ENTITY_DECLARED_IN_PE (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_EXTERNAL_ENTITY_HANDLING (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_FEATURE_REQUIRES_XML_DTD (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_FINISHED (en el módulo `xml.parsers.expat.errors`), 1198

XML_ERROR_INCOMPLETE_PE (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_INCORRECT_ENCODING (en el módulo `xml.parsers.expat.errors`), 1196

XML_ERROR_INVALID_TOKEN (en el módulo `xml.parsers.expat.errors`), 1196

XML_ERROR_JUNK_AFTER_DOC_ELEMENT (en el módulo `xml.parsers.expat.errors`), 1196

XML_ERROR_MISPLACED_XML_PI (en el módulo `xml.parsers.expat.errors`), 1196

XML_ERROR_NO_ELEMENTS (en el módulo `xml.parsers.expat.errors`), 1196

XML_ERROR_NO_MEMORY (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_NOT_STANDALONE (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_NOT_SUSPENDED (en el módulo `xml.parsers.expat.errors`), 1198

XML_ERROR_PARAM_ENTITY_REF (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_PARTIAL_CHAR (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_PUBLICID (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_RECURSIVE_ENTITY_REF (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_SUSPEND_PE (en el módulo `xml.parsers.expat.errors`), 1198

XML_ERROR_SUSPENDED (en el módulo `xml.parsers.expat.errors`), 1198

XML_ERROR_SYNTAX (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_TAG_MISMATCH (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_TEXT_DECL (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_UNBOUND_PREFIX (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_UNCLOSED_CDATA_SECTION (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_UNCLOSED_TOKEN (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_UNDECLARING_PREFIX (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_UNDEFINED_ENTITY (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_UNEXPECTED_STATE (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_UNKNOWN_ENCODING (en el módulo `xml.parsers.expat.errors`), 1197

XML_ERROR_XML_DECL (en el módulo `xml.parsers.expat.errors`), 1197

XML_NAMESPACE (en el módulo `xml.dom`), 1162

xmlcharrefreplace_errors() (en el módulo `codecs`), 163

XmlDeclHandler() (método de `xml.parsers.expat.xmlparser`), 1192

xml.dom (módulo), 1161

xml.dom.minidom (módulo), 1171

xml.dom.pulldom (módulo), 1175

xml.etree.ElementInclude.default_loader() (en el módulo `xml.etree.ElementTree`), 1153

- `xml.etree.ElementInclude.include()` (en el módulo `xml.etree.ElementTree`), 1153
- `xml.etree.ElementTree` (módulo), 1142
- `XMLFilterBase` (clase en `xml.sax.saxutils`), 1184
- `XMLGenerator` (clase en `xml.sax.saxutils`), 1184
- `XMLID()` (en el módulo `xml.etree.ElementTree`), 1152
- `XMLNS_NAMESPACE` (en el módulo `xml.dom`), 1162
- `XMLParser` (clase en `xml.etree.ElementTree`), 1158
- `xml.parsers.expat` (módulo), 1189
- `xml.parsers.expat.errors` (módulo), 1196
- `xml.parsers.expat.model` (módulo), 1195
- `XMLParserType` (en el módulo `xml.parsers.expat`), 1189
- `XMLPullParser` (clase en `xml.etree.ElementTree`), 1160
- `XMLReader` (clase en `xml.sax.xmlreader`), 1185
- `xmlrpc.client` (módulo), 1320
- `xmlrpc.server` (módulo), 1327
- `xml.sax` (módulo), 1177
- `xml.sax.handler` (módulo), 1179
- `xml.sax.saxutils` (módulo), 1184
- `xml.sax.xmlreader` (módulo), 1185
- `xor()` (en el módulo `operator`), 367
- `xover()` (método de `nnplib.NNTP`), 1277
- `xpath()` (método de `nnplib.NNTP`), 1278
- `xrange` (2to3 fixer), 1611
- `xreadlines` (2to3 fixer), 1611
- `xview()` (método de `tkinter.ttk.Treeview`), 1455
- Y**
- `ycor()` (en el módulo `turtle`), 1395
- `year` (atributo de `datetime.date`), 184
- `year` (atributo de `datetime.datetime`), 190
- `Year` 2038, 622
- `yeardatescalendar()` (método de `calendar.Calendar`), 213
- `yeardays2calendar()` (método de `calendar.Calendar`), 213
- `yeardayscalendar()` (método de `calendar.Calendar`), 213
- `YESEXPR` (en el módulo `locale`), 1379
- `YIELD_FROM` (opcode), 1848
- `YIELD_VALUE` (opcode), 1848
- `yiq_to_rgb()` (en el módulo `colorsys`), 1359
- `yview()` (método de `tkinter.ttk.Treeview`), 1455
- Z**
- Zen de Python, 1947
- `ZeroDivisionError`, 95
- `zfill()` (método de `bytearray`), 66
- `zfill()` (método de `bytes`), 66
- `zfill()` (método de `str`), 52
- `zip` (2to3 fixer), 1611
- `zip()` (función incorporada), 24
- `ZIP_BZIP2` (en el módulo `zipfile`), 486
- `ZIP_DEFLATED` (en el módulo `zipfile`), 486
- `zip_longest()` (en el módulo `itertools`), 353
- `ZIP_LZMA` (en el módulo `zipfile`), 486
- `ZIP_STORED` (en el módulo `zipfile`), 486
- `zipapp` (módulo), 1684
- `zipapp` command line option
- `-c`, 1684
 - `--compress`, 1684
 - `-h`, 1685
 - `--help`, 1685
 - `--info`, 1684
 - `-m <mainfn>`, 1684
 - `--main=<mainfn>`, 1684
 - `-o <output>`, 1684
 - `--output=<output>`, 1684
 - `-p <interpreter>`, 1684
 - `--python=<interpreter>`, 1684
- `ZipFile` (clase en `zipfile`), 486
- `zipfile` (módulo), 485
- `zipfile` command line option
- `-c <zipfile> <source1> ... <sourceN>`, 494
 - `--create <zipfile> <source1> ... <sourceN>`, 494
 - `-e <zipfile> <output_dir>`, 494
 - `--extract <zipfile> <output_dir>`, 494
 - `-l <zipfile>`, 494
 - `--list <zipfile>`, 494
 - `-t <zipfile>`, 494
 - `--test <zipfile>`, 494
- `zipimport` (módulo), 1781
- `zipimporter` (clase en `zipimport`), 1782
- `ZipImportError`, 1782
- `ZipInfo` (clase en `zipfile`), 485
- `zlib` (módulo), 469
- `ZLIB_RUNTIME_VERSION` (en el módulo `zlib`), 472
- `ZLIB_VERSION` (en el módulo `zlib`), 472