The Python Library Reference

Versión 3.10.13

Guido van Rossum
and the Python development team

noviembre 14, 2023

Python Software Foundation
Email: docs@python.org

Índice general

1 Introducción **3**
1.1 Notas sobre la disponibilidad . . . 4

2 Funciones Built-in **5**

3 Constantes incorporadas **31**
3.1 Constantes agregadas por el módulo `site` . . . 32

4 Tipos Integrados **33**
4.1 Evaluar como valor verdadero/falso . . . 33
4.2 Operaciones booleanas — `and`, `or`, `not` . . . 34
4.3 Comparaciones . . . 34
4.4 Tipos numéricos — `int`, `float`, `complex` . . . 35
4.4.1 Operaciones de bits en números enteros . . . 36
4.4.2 Métodos adicionales de los enteros . . . 37
4.4.3 Métodos adicionales de Float . . . 38
4.4.4 Calculo del *hash* de tipos numéricos . . . 40
4.5 Tipos de iteradores . . . 41
4.5.1 Tipos Generador . . . 42
4.6 Tipos secuencia — `list`, `tuple`, `range` . . . 42
4.6.1 Operaciones comunes de las secuencias . . . 42
4.6.2 Tipos de secuencia inmutables . . . 44
4.6.3 Tipos de secuencia mutables . . . 44
4.6.4 Listas . . . 45
4.6.5 Tuplas . . . 46
4.6.6 Rangos . . . 46
4.7 Cadenas de caracteres — `str` . . . 48
4.7.1 Métodos de las cadenas de caracteres . . . 49
4.7.2 Formateo de cadenas al estilo `*printf*` . . . 57
4.8 Tipos de secuencias binarias — `bytes`, `bytearray` y `memoryview` . . . 59
4.8.1 Objetos de tipo Bytes . . . 59
4.8.2 Objetos de tipo *Bytearray* . . . 61
4.8.3 Operaciones de *bytes* y *bytearray* . . . 62
4.8.4 Usando el formateo tipo `printf` con bytes . . . 72
4.8.5 Vistas de memoria . . . 74
4.9 Conjuntos — `set`, `frozenset` . . . 81
4.10 Tipos Mapa — `dict` . . . 83
4.10.1 Objetos tipos vista de diccionario . . . 87
4.11 Tipos Gestores de Contexto . . . 88
4.12 Tipos de Anotaciones de Tipo — Generic Alias, Union . . . 89
4.12.1 Tipo Alias Genérico . . . 89

4.12.2 Tipo de conversión . . . 93
4.13 Otros tipos predefinidos . . . 94
4.13.1 Módulos . . . 94
4.13.2 Clases e instancias de clase . . . 95
4.13.3 Funciones . . . 95
4.13.4 Métodos . . . 95
4.13.5 Objetos código . . . 96
4.13.6 Objetos Tipo . . . 96
4.13.7 El objeto nulo (*Null*) . . . 96
4.13.8 El objeto puntos suspensivos (*Ellipsis*) . . . 96
4.13.9 El objeto *NotImplemented* . . . 96
4.13.10 Valores booleanos . . . 97
4.13.11 Objetos internos . . . 97
4.14 Atributos especiales . . . 97
4.15 Integer string conversion length limitation . . . 98
4.15.1 Affected APIs . . . 99
4.15.2 Configuring the limit . . . 99
4.15.3 Recommended configuration . . . 100

5 Excepciones incorporadas **101**
5.1 Exception context . . . 101
5.2 Inheriting from built-in exceptions . . . 102
5.3 Clases base . . . 102
5.4 Excepciones específicas . . . 103
5.4.1 Excepciones del sistema operativo . . . 108
5.5 Advertencias . . . 109
5.6 Jerarquía de excepción . . . 110

6 Servicios de procesamiento de texto **113**
6.1 `string` — Operaciones comunes de cadena de caracteres . . . 113
6.1.1 Constantes de cadenas . . . 113
6.1.2 Formato de cadena de caracteres personalizado . . . 114
6.1.3 Sintaxis de formateo de cadena . . . 115
6.1.4 Cadenas de plantillas . . . 122
6.1.5 Funciones de ayuda . . . 124
6.2 `re` — Operaciones con expresiones regulares . . . 124
6.2.1 Sintaxis de expresiones regulares . . . 125
6.2.2 Contenidos del módulo . . . 130
6.2.3 Objetos expresión regular . . . 135
6.2.4 Objetos de coincidencia . . . 136
6.2.5 Ejemplos de expresiones regulares . . . 139
6.3 `difflib` — Funciones auxiliares para calcular deltas . . . 144
6.3.1 Objetos *SequenceMatcher* . . . 149
6.3.2 `SequenceMatcher` Ejemplos . . . 151
6.3.3 Objetos *Differ* . . . 152
6.3.4 Ejemplo de *Differ* . . . 153
6.3.5 Una interfaz de línea de comandos para `difflib` . . . 154
6.4 `textwrap` — Envoltura y relleno de texto . . . 155
6.5 `unicodedata` — Base de datos Unicode . . . 158
6.6 `stringprep` — Preparación de cadenas de Internet . . . 160
6.7 `readline` — Interfaz readline de GNU . . . 162
6.7.1 Archivo de inicio . . . 162
6.7.2 Búfer de línea . . . 162
6.7.3 Archivo de historial . . . 163
6.7.4 Lista del historial . . . 163
6.7.5 Ganchos (*hooks*) de inicialización . . . 164
6.7.6 Terminación . . . 164
6.7.7 Ejemplo . . . 165

6.8 rlcompleter — Función de completado para GNU readline . . . 166
6.8.1 Objetos de Completado . . . 166

7 Servicios de datos binarios **167**
7.1 struct — Interpreta bytes como paquetes de datos binarios . . . 167
7.1.1 Funciones y Excepciones . . . 168
7.1.2 Cadenas de Formato . . . 168
7.1.3 Applications . . . 172
7.1.4 Clases . . . 173
7.2 codecs — Registro de códec y clases base . . . 174
7.2.1 Clases Base de Códec . . . 177
7.2.2 Codificaciones y Unicode . . . 183
7.2.3 Codificaciones estándar . . . 185
7.2.4 Codificaciones específicas de Python . . . 188
7.2.5 encodings.idna — Nombres de dominio internacionalizados en aplicaciones . . . 190
7.2.6 encodings.mbcs — Página de códigos ANSI de Windows . . . 190
7.2.7 encodings.utf_8_sig — Códec UTF-8 con firma BOM . . . 191

8 Tipos de datos **193**
8.1 datetime — Tipos básicos de fecha y hora . . . 193
8.1.1 Objetos conscientes (*aware*) y naífs (*naive*) . . . 194
8.1.2 Constantes . . . 194
8.1.3 Tipos disponibles . . . 194
8.1.4 Objetos timedelta . . . 195
8.1.5 Objeto date . . . 199
8.1.6 Objetos datetime . . . 203
8.1.7 Objetos time . . . 213
8.1.8 Objetos tzinfo . . . 217
8.1.9 Objetos timezone . . . 223
8.1.10 Comportamiento strftime() y strptime() . . . 224
8.2 zoneinfo — Soporte de zona horaria IANA . . . 229
8.2.1 Usando ZoneInfo . . . 229
8.2.2 Fuentes de datos . . . 230
8.2.3 La clase ZoneInfo . . . 231
8.2.4 Funciones . . . 233
8.2.5 Globales . . . 234
8.2.6 Excepciones y advertencias . . . 234
8.3 calendar — Funciones generales relacionadas con el calendario . . . 234
8.4 collections — Tipos de datos contenedor . . . 239
8.4.1 Objetos ChainMap . . . 239
8.4.2 Objetos Counter . . . 242
8.4.3 Objetos deque . . . 245
8.4.4 Objetos defaultdict . . . 248
8.4.5 namedtuple() Funciones *Factory* para Tuplas y Campos con Nombres . . . 250
8.4.6 Objetos OrderedDict . . . 253
8.4.7 Objetos UserDict . . . 255
8.4.8 Objetos UserList . . . 256
8.4.9 Objetos UserString . . . 256
8.5 collections.abc — Clases Base Abstractas para Contenedores . . . 257
8.5.1 Colecciones Clases Base Abstractas . . . 258
8.5.2 Colecciones Clases base abstractas - Descripciones detalladas . . . 260
8.5.3 Ejemplos y Recetas . . . 261
8.6 heapq — Algoritmo de colas montículos (*heap*) . . . 262
8.6.1 Ejemplos Básicos . . . 264
8.6.2 Notas de Aplicación de la Cola de Prioridades . . . 264
8.6.3 Teoría . . . 265
8.7 bisect — Algoritmo de bisección de arreglos . . . 266
8.7.1 Notas de rendimiento . . . 268

8.7.2 Búsqueda en listas ordenadas . . . 268
8.7.3 Ejemplos . . . 269
8.8 array — Arreglos eficientes de valores numéricos . . . 270
8.9 weakref — Referencias débiles . . . 272
8.9.1 Objetos de Referencias Débiles . . . 277
8.9.2 Ejemplo . . . 278
8.9.3 Objetos Finalizadores . . . 278
8.9.4 Comparando finalizadores con los métodos __del__() . . . 279
8.10 types — Creación de tipos dinámicos y nombres para tipos integrados . . . 280
8.10.1 Creación dinámica de tipos . . . 280
8.10.2 Tipos de Intérpretes Estándar . . . 281
8.10.3 Clases y funciones de utilidad adicionales . . . 285
8.10.4 Funciones de utilidad de corutina . . . 285
8.11 copy — Operaciones de copia superficial y profunda . . . 286
8.12 pprint — Impresión bonita de datos . . . 287
8.12.1 Objetos *PrettyPrinter* . . . 289
8.12.2 Ejemplo . . . 290
8.13 reprlib — Implementación repr() alternativa . . . 292
8.13.1 Objetos Repr . . . 293
8.13.2 Subclasificando Objetos Repr . . . 294
8.14 enum — Soporte para enumeraciones . . . 294
8.14.1 Contenido del Módulo . . . 294
8.14.2 Creando un Enum . . . 295
8.14.3 Acceso programático a los miembros de la enumeración y sus atributos . . . 296
8.14.4 Duplicando miembros y valores enum . . . 297
8.14.5 Garantizando valores de enumeración únicos . . . 297
8.14.6 Usando valores automáticos . . . 298
8.14.7 Iteración . . . 298
8.14.8 Comparaciones . . . 299
8.14.9 Miembros permitidos y atributos de enumeraciones . . . 299
8.14.10 Subclases restringidas de Enum . . . 300
8.14.11 Serialización . . . 301
8.14.12 API Funcional . . . 301
8.14.13 Enumeraciones derivadas . . . 302
8.14.14 Cuándo usar __new__() contra __init__() . . . 305
8.14.15 Ejemplos interesantes . . . 306
8.14.16 ¿Cómo son diferentes las Enums? . . . 310
8.15 graphlib —Funcionalidad para operar con estructuras de tipo-grafo . . . 313
8.15.1 Excepciones . . . 315

9 Módulos numéricos y matemáticos **317**
9.1 numbers — Clase base abstracta numérica . . . 317
9.1.1 La torre numérica . . . 317
9.1.2 Notas para implementadores de tipos . . . 318
9.2 math — Funciones matemáticas . . . 320
9.2.1 Teoría de números y funciones de representación . . . 321
9.2.2 Funciones logarítmicas y exponenciales . . . 324
9.2.3 Funciones trigonométricas . . . 325
9.2.4 Conversión angular . . . 326
9.2.5 Funciones hiperbólicas . . . 326
9.2.6 Funciones especiales . . . 326
9.2.7 Constantes . . . 327
9.3 cmath – Función matemática para números complejos . . . 328
9.3.1 Conversión a y desde coordenadas polares . . . 328
9.3.2 Funciones logarítmicas y de potencias . . . 329
9.3.3 Funciones trigonométricas . . . 329
9.3.4 Funciones hiperbólicas . . . 329
9.3.5 Funciones de clasificación . . . 330

9.3.6 Constantes . . . 330
9.4 decimal — Aritmética decimal de coma fija y coma flotante . . . 331
9.4.1 Tutorial de inicio rápido . . . 332
9.4.2 Objetos Decimal . . . 335
9.4.3 Objetos Context . . . 342
9.4.4 Constantes . . . 348
9.4.5 Modos de redondeo . . . 349
9.4.6 Señales . . . 349
9.4.7 Notas sobre la representación en coma flotante . . . 351
9.4.8 Trabajando con hilos . . . 352
9.4.9 Casos prácticos . . . 353
9.4.10 Preguntas frecuentes sobre Decimal . . . 355
9.5 fractions — Números racionales . . . 358
9.6 random —Generar números pseudoaleatorios . . . 360
9.6.1 Funciones de contabilidad . . . 361
9.6.2 Funciones para los bytes . . . 362
9.6.3 Funciones para enteros . . . 362
9.6.4 Funciones para secuencias . . . 362
9.6.5 Distribuciones para los nombres reales . . . 364
9.6.6 Generador alternativo . . . 365
9.6.7 Notas sobre la Reproducibilidad . . . 365
9.6.8 Ejemplos . . . 365
9.6.9 Recetas . . . 367
9.7 statistics — Funciones de estadística matemática . . . 368
9.7.1 Promedios y medidas de tendencia central . . . 369
9.7.2 Medidas de dispersión . . . 369
9.7.3 Estadísticas para relaciones entre dos entradas . . . 369
9.7.4 Detalles de las funciones . . . 370
9.7.5 Excepciones . . . 377
9.7.6 Objetos NormalDist . . . 377

10 Módulos de programación funcional **381**
10.1 itertools — Funciones que crean iteradores para bucles eficientes . . . 381
10.1.1 Funciones de itertools . . . 383
10.1.2 Fórmulas con itertools . . . 391
10.2 functools — Funciones de orden superior y operaciones sobre objetos invocables . . . 396
10.2.1 partial Objetos . . . 405
10.3 operator — Operadores estándar como funciones . . . 406
10.3.1 Asignación de operadores a funciones . . . 410
10.3.2 Operadores *In-place* . . . 411

11 Acceso a archivos y directorios **413**
11.1 pathlib — Rutas de sistemas orientada a objetos . . . 413
11.1.1 Uso básico . . . 414
11.1.2 Rutas puras . . . 415
11.1.3 Rutas concretas . . . 423
11.1.4 Correspondencia a herramientas en el módulo os . . . 430
11.2 os.path — Manipulaciones comunes de nombre de ruta . . . 431
11.3 fileinput — Iterar sobre líneas de múltiples flujos de entrada . . . 437
11.4 stat — Interpretación de los resultados de stat() . . . 439
11.5 filecmp— Comparaciones de Archivo y Directorio . . . 444
11.5.1 La clase dircmp . . . 445
11.6 tempfile — Generar archivos y directorios temporales . . . 446
11.6.1 Ejemplos . . . 450
11.6.2 Funciones y variables deprecadas . . . 450
11.7 glob — Expansión del patrón de nombres de ruta de estilo Unix . . . 451
11.8 fnmatch — Coincidencia de patrones de nombre de archivos Unix . . . 452
11.9 linecache — Acceso aleatorio a líneas de texto . . . 453

11.10 shutil — Operaciones de archivos de alto nivel . . . 454
11.10.1 Operaciones de directorios y archivos . . . 454
11.10.2 Operaciones de archivado . . . 460
11.10.3 Consulta el tamaño de la terminal de salida . . . 463

12 Persistencia de datos **465**
12.1 pickle — Serialización de objetos Python . . . 465
12.1.1 Relación con otros módulos de Python . . . 466
12.1.2 Formato de flujo de datos . . . 467
12.1.3 Interfaz del módulo . . . 467
12.1.4 ¿Qué se puede serializar (pickled) y deserializar (unpickled) con *pickle*? . . . 471
12.1.5 *Pickling* de Instancias de clases . . . 472
12.1.6 Reducción personalizada para tipos, funciones y otros objetos . . . 477
12.1.7 Búferes fuera de banda . . . 478
12.1.8 Restricción de Globals . . . 480
12.1.9 Performance . . . 481
12.1.10 Ejemplos . . . 481
12.2 copyreg — Registrar funciones de soporte de pickle . . . 481
12.2.1 Ejemplo . . . 482
12.3 shelve — Persistencia de objetos de Python . . . 482
12.3.1 Restricciones . . . 483
12.3.2 Ejemplo . . . 484
12.4 marshal — Serialización interna de objetos Python . . . 485
12.5 dbm — Interfaces para «bases de datos» de Unix . . . 486
12.5.1 dbm.gnu — La reinterpretación de GNU de dbm . . . 488
12.5.2 dbm.ndbm — Interfaz basada en ndbm . . . 489
12.5.3 dbm.dumb — Implementación de DBM portátil . . . 490
12.6 sqlite3 — DB-API 2.0 interfaz para bases de datos SQLite . . . 491
12.6.1 Tutorial . . . 491
12.6.2 Reference . . . 493
12.6.3 How-to guides . . . 506
12.6.4 Explanation . . . 512

13 Compresión de datos y archivado **513**
13.1 zlib — Compresión compatible con **gzip** . . . 513
13.2 gzip — Soporte para archivos **gzip** . . . 517
13.2.1 Ejemplos de uso . . . 519
13.2.2 Interfaz de Línea de Comandos . . . 519
13.3 bz2 — Soporte para compresión **bzip2** . . . 520
13.3.1 (De)compresión de archivos . . . 520
13.3.2 (De)compresión incremental . . . 521
13.3.3 (Des)comprimir en un solo paso . . . 522
13.3.4 Ejemplos de uso . . . 523
13.4 lzma — Compresión utilizando el algoritmo LZMA . . . 524
13.4.1 Leyendo y escribiendo ficheros comprimidos . . . 524
13.4.2 Comprimiendo y descomprimiendo datos en memoria . . . 525
13.4.3 Misceláneas . . . 527
13.4.4 Especificando cadenas de filtro personalizadas . . . 527
13.4.5 Ejemplos . . . 528
13.5 zipfile — Trabajar con archivos ZIP . . . 529
13.5.1 Objetos ZipFile . . . 530
13.5.2 Objetos de ruta . . . 534
13.5.3 Objetos PyZipFile . . . 535
13.5.4 Objetos ZipInfo . . . 536
13.5.5 Interfaz de línea de comandos . . . 538
13.5.6 Problemas de descompresión . . . 538
13.6 tarfile — Leer y escribir archivos tar . . . 539
13.6.1 Objetos *TarFile* . . . 542

13.6.2 Objetos TarInfo . . . 545
13.6.3 Extraction filters . . . 548
13.6.4 Interfaz de línea de comandos . . . 551
13.6.5 Ejemplos . . . 552
13.6.6 Formatos tar con soporte . . . 553
13.6.7 Problemas Unicode . . . 553

14 Formatos de archivo **555**
14.1 `csv` — Lectura y escritura de archivos CSV . . . 555
14.1.1 Contenidos del módulo . . . 556
14.1.2 Dialectos y parámetros de formato . . . 559
14.1.3 Objetos *Reader* . . . 560
14.1.4 Objetos *Writer* . . . 560
14.1.5 Ejemplos . . . 561
14.2 `configparser` — *Parser* para archivos de configuración . . . 562
14.2.1 Inicio Rápido . . . 562
14.2.2 Tipos de Datos Soportados . . . 564
14.2.3 Valores de contingencia . . . 565
14.2.4 Estructura Soportada para el Archivo INI . . . 565
14.2.5 Interpolación de valores . . . 566
14.2.6 Acceso por Protocolo de Mapeo . . . 568
14.2.7 Personalizando el Comportamiento del Parser . . . 568
14.2.8 Ejemplos de la API heredada . . . 573
14.2.9 Objetos ConfigParser . . . 575
14.2.10 Objetos RawConfigParser . . . 579
14.2.11 Excepciones . . . 579
14.3 `netrc` — procesado del fichero netrc . . . 580
14.3.1 Objetos netrc . . . 581
14.4 `plistlib` — Genera y analiza archivos `.plist` de Apple . . . 581
14.4.1 Ejemplos . . . 583

15 Servicios Criptográficos **585**
15.1 `hashlib` — Hashes seguros y resúmenes de mensajes . . . 585
15.1.1 Algoritmos de hash . . . 585
15.1.2 Resúmenes SHAKE de largo variable . . . 587
15.1.3 Derivación de clave . . . 588
15.1.4 BLAKE2 . . . 588
15.2 `hmac` — *Hash* con clave para autenticación de mensajes . . . 595
15.3 `secrets` — Genera números aleatorios seguros para trabajar con secretos criptográficos . . . 597
15.3.1 Números aleatorios . . . 597
15.3.2 Generando tokens . . . 598
15.3.3 Otras funciones . . . 598
15.3.4 Recetas y mejores prácticas . . . 599

16 Servicios genéricos del sistema operativo **601**
16.1 `os` — Interfaces misceláneas del sistema operativo . . . 601
16.1.1 Nombres de archivos, argumentos de la línea de comandos y variables de entorno . . . 602
16.1.2 Modo Python UTF-8 . . . 602
16.1.3 Parámetros de proceso . . . 603
16.1.4 Creación de objetos de tipo archivo . . . 609
16.1.5 Operaciones de descriptores de archivos . . . 609
16.1.6 Archivos y directorios . . . 619
16.1.7 Gestión de proceso . . . 641
16.1.8 Interfaz al planificador . . . 652
16.1.9 Información miscelánea del sistema . . . 654
16.1.10 Números al azar . . . 655
16.2 `io` — Herramientas principales para trabajar con *streams* . . . 656
16.2.1 Resumen . . . 657
16.2.2 Codificación de texto . . . 658

16.2.3 Interfaz de módulo de alto nivel . . . 658
16.2.4 Jerarquía de clases . . . 659
16.2.5 Rendimiento . . . 669
16.3 `time` — Acceso a tiempo y conversiones . . . 670
16.3.1 Las Funciones . . . 671
16.3.2 Constantes de ID de reloj . . . 678
16.3.3 Constantes de zona horaria . . . 679
16.4 `argparse` — Analizador sintáctico (*Parser*) para las opciones, argumentos y sub-comandos de la línea de comandos . . . 680
16.4.1 Ejemplo . . . 680
16.4.2 Objetos *ArgumentParser* . . . 682
16.4.3 El método *add_argument()* . . . 690
16.4.4 El método *parse_args()* . . . 700
16.4.5 Otras utilidades . . . 703
16.4.6 Actualizar el código de *optparse* . . . 711
16.5 `getopt` — Analizador de estilo C para opciones de línea de comando . . . 711
16.6 `logging` — Instalación de logging para Python . . . 714
16.6.1 Objetos Logger . . . 714
16.6.2 Niveles de logging . . . 718
16.6.3 Gestor de objetos . . . 718
16.6.4 Objetos formateadores . . . 720
16.6.5 Filtro de Objetos . . . 722
16.6.6 Objetos LogRecord . . . 722
16.6.7 Atributos LogRecord . . . 723
16.6.8 Objetos LoggerAdapter . . . 725
16.6.9 Seguridad del hilo . . . 725
16.6.10 Funciones a nivel de módulo . . . 725
16.6.11 Atributos a nivel de módulo . . . 730
16.6.12 Integración con el módulo de advertencias . . . 730
16.7 `logging.config` — Configuración de registro . . . 730
16.7.1 Funciones de configuración . . . 731
16.7.2 Security considerations . . . 733
16.7.3 Esquema del diccionario de configuración . . . 733
16.7.4 Formato de archivo de configuración . . . 739
16.8 `logging.handlers` — Gestores de *logging* . . . 742
16.8.1 StreamHandler . . . 742
16.8.2 FileHandler . . . 743
16.8.3 NullHandler . . . 743
16.8.4 WatchedFileHandler . . . 744
16.8.5 BaseRotatingHandler . . . 744
16.8.6 RotatingFileHandler . . . 745
16.8.7 TimedRotatingFileHandler . . . 746
16.8.8 SocketHandler . . . 747
16.8.9 DatagramHandler . . . 748
16.8.10 Gestor *SysLog* (*SysLogHandler*) . . . 749
16.8.11 Gestor de eventos *NTELog* (NTEventLogHandler) . . . 751
16.8.12 SMTPHandler . . . 751
16.8.13 MemoryHandler . . . 752
16.8.14 HTTPHandler . . . 753
16.8.15 QueueHandler . . . 753
16.8.16 QueueListener . . . 754
16.9 `getpass` — Entrada de contraseña portátil . . . 755
16.10 `curses` — Manejo de terminales para pantallas de celdas de caracteres . . . 756
16.10.1 Funciones . . . 757
16.10.2 Objetos de ventana . . . 763
16.10.3 Constantes . . . 770
16.11 `curses.textpad`— Widget de entrada de texto para programas de curses . . . 774
16.11.1 Objeto de caja de texto . . . 774

16.12 curses.ascii — Utilidades para los caracteres ASCII . . . 775
16.13 curses.panel — Una extensión de pila de panel para curses . . . 778
16.13.1 Funciones . . . 778
16.13.2 Objetos de Panel . . . 778
16.14 platform — Acceso a los datos identificativos de la plataforma subyacente . . . 779
16.14.1 Plataforma cruzada . . . 779
16.14.2 Plataforma Java . . . 781
16.14.3 Plataforma windows . . . 781
16.14.4 Plataforma macOS . . . 781
16.14.5 Plataformas Unix . . . 782
16.14.6 Plataformas Linux . . . 782
16.15 errno — Símbolos estándar del sistema errno . . . 782
16.16 ctypes — Una biblioteca de funciones foráneas para Python . . . 788
16.16.1 tutorial de ctypes . . . 788
16.16.2 referencia ctypes . . . 806

17 Ejecución concurrente **821**
17.1 threading — Paralelismo basado en hilos . . . 821
17.1.1 Datos locales del hilo . . . 824
17.1.2 Objetos tipo hilo . . . 824
17.1.3 Objetos tipo *lock* . . . 826
17.1.4 Objetos *Rlock* . . . 827
17.1.5 Objetos condicionales . . . 828
17.1.6 Objetos semáforo . . . 830
17.1.7 Objetos de eventos . . . 831
17.1.8 Objetos temporizadores . . . 832
17.1.9 Objetos de barrera . . . 833
17.1.10 Uso de *locks*, condiciones y semáforos en la declaración with . . . 834
17.2 multiprocessing — Paralelismo basado en procesos . . . 834
17.2.1 Introducción . . . 834
17.2.2 Referencia . . . 841
17.2.3 Pautas de programación . . . 869
17.2.4 Ejemplos . . . 872
17.3 multiprocessing.shared_memory — Shared memory for direct access across processes . . . 877
17.4 El paquete concurrent . . . 882
17.5 concurrent.futures — Lanzamiento de tareas paralelas . . . 882
17.5.1 Objetos Ejecutores . . . 882
17.5.2 ThreadPoolExecutor . . . 883
17.5.3 ProcessPoolExecutor . . . 885
17.5.4 Objetos Futuro . . . 886
17.5.5 Funciones del Módulo . . . 888
17.5.6 Clases de Excepciones . . . 888
17.6 subprocess — Gestión de subprocesos . . . 889
17.6.1 Uso del módulo subprocess . . . 889
17.6.2 Consideraciones sobre seguridad . . . 897
17.6.3 Objetos Popen . . . 898
17.6.4 Elementos auxiliares de Popen en Windows . . . 900
17.6.5 Antigua API de alto nivel . . . 902
17.6.6 Cómo reemplazar anteriores funciones con el módulo subprocess . . . 904
17.6.7 Funciones de llamada a la shell de retrocompatibilidad . . . 906
17.6.8 Notas . . . 907
17.7 sched — Eventos del planificador . . . 908
17.7.1 Objetos de Scheduler . . . 908
17.8 queue — clase de cola sincronizada . . . 909
17.8.1 Objetos de la cola . . . 910
17.8.2 Objetos de cola simple . . . 912
17.9 contextvars — Variables de Contexto . . . 912
17.9.1 Variables de Contexto . . . 913

17.9.2 Gestión de Contexto Manual . . . 914
17.9.3 Soporte asyncio . . . 915
17.10 _thread — API de bajo nivel para manejo de hilos . . . 916

18 Comunicación en redes y entre procesos **919**
18.1 asyncio — E/S Asíncrona . . . 919
18.1.1 Corrutinas y Tareas . . . 920
18.1.2 Streams . . . 934
18.1.3 Primitivas de sincronización . . . 941
18.1.4 Sub-procesos . . . 945
18.1.5 Colas . . . 950
18.1.6 Excepciones . . . 952
18.1.7 Bucle de eventos . . . 953
18.1.8 Futures . . . 975
18.1.9 Transportes y protocolos . . . 978
18.1.10 Políticas . . . 992
18.1.11 Soporte de plataforma . . . 995
18.1.12 Índice de API de alto nivel . . . 997
18.1.13 Índice de API de bajo nivel . . . 999
18.1.14 Desarrollando con asyncio . . . 1005
18.2 socket — interfaz de red de bajo nivel . . . 1008
18.2.1 Familias Socket . . . 1009
18.2.2 Contenido del módulo . . . 1011
18.2.3 Objetos Socket . . . 1022
18.2.4 Notas sobre los tiempos de espera del socket . . . 1029
18.2.5 Ejemplo . . . 1030
18.3 ssl —Empaquetador o wrapper TLS/SSL para objetos de tipo socket . . . 1033
18.3.1 Funciones, constantes y excepciones . . . 1034
18.3.2 Sockets SSL . . . 1046
18.3.3 Contextos SSL . . . 1050
18.3.4 Certificados . . . 1058
18.3.5 Ejemplos . . . 1060
18.3.6 Notas sobre los sockets no bloqueantes . . . 1062
18.3.7 Soporte de memoria BIO . . . 1063
18.3.8 Sesión SSL . . . 1065
18.3.9 Consideraciones de seguridad . . . 1065
18.3.10 TLS 1.3 . . . 1067
18.4 select — Esperando la finalización de E/S . . . 1068
18.4.1 Objetos de sondeo /dev/poll . . . 1070
18.4.2 Objetos de sondeo de *Edge* y *Level Trigger* (*epoll*) . . . 1071
18.4.3 Sondeo de objetos . . . 1072
18.4.4 Objetos Kqueue . . . 1073
18.4.5 Objetos Kevent . . . 1073
18.5 selectors — Multiplexación de E/S de alto nivel . . . 1075
18.5.1 Introducción . . . 1075
18.5.2 Clases . . . 1075
18.5.3 Ejemplos . . . 1077
18.6 signal — Establece gestores para eventos asíncronos . . . 1078
18.6.1 Reglas generales . . . 1078
18.6.2 Contenidos del módulo . . . 1079
18.6.3 Ejemplo . . . 1085
18.6.4 Nota sobre SIGPIPE . . . 1085
18.6.5 Note on Signal Handlers and Exceptions . . . 1086
18.7 mmap — Soporte de archivos mapeados en memoria . . . 1087
18.7.1 Constantes MADV_* . . . 1090
18.7.2 Constantes MAP_* . . . 1091

19 Manejo de Datos de Internet **1093**

19.1 email — Un paquete de manejo de correo electrónico y MIME . . . 1093
19.1.1 email.message: Representando un mensaje de correo electrónico . . . 1094
19.1.2 email.parser: Analizar mensajes de correo electrónico . . . 1102
19.1.3 email.generator: Generando documentos MIME . . . 1106
19.1.4 email.policy: Objetos *Policy* . . . 1109
19.1.5 email.errors: Clases de excepción y defecto . . . 1116
19.1.6 email.headerregistry: Objetos de encabezado personalizados . . . 1117
19.1.7 email.contentmanager: Gestión de contenido MIME . . . 1123
19.1.8 email: Ejemplos . . . 1125
19.1.9 email.message.Message: Representar un mensaje de correo electrónico usando la API compat32 . . . 1131
19.1.10 email.mime: Creación de correo electrónico y objetos MIME desde cero . . . 1140
19.1.11 email.header: Cabeceras internacionalizadas . . . 1142
19.1.12 email.charset: Representa conjunto de caracteres . . . 1145
19.1.13 email.encoders: Codificadores . . . 1147
19.1.14 email.utils: Utilidades misceláneas . . . 1148
19.1.15 email.iterators: Iteradores . . . 1150
19.2 json — Codificador y decodificador JSON . . . 1152
19.2.1 Uso básico . . . 1154
19.2.2 Codificadores y Decodificadores . . . 1156
19.2.3 Excepciones . . . 1158
19.2.4 Cumplimiento e interoperabilidad estándar . . . 1158
19.2.5 Interfaz de línea de comandos . . . 1160
19.3 mailbox — Manipular buzones de correo en varios formatos . . . 1161
19.3.1 Objetos :class:”Mailbox” . . . 1162
19.3.2 Objetos Message . . . 1170
19.3.3 Excepciones . . . 1177
19.3.4 Ejemplos . . . 1178
19.4 mimetypes — Mapea nombres de archivo a tipos MIME . . . 1179
19.4.1 Objetos MimeTypes . . . 1181
19.5 base64 — Codificaciones de datos Base16, Base32, Base64, y Base85 . . . 1182
19.5.1 Consideraciones de Seguridad . . . 1185
19.6 binhex — Codificar y decodificar archivos binhex4 . . . 1185
19.6.1 Notas . . . 1186
19.7 binascii — Convertir entre binario y ASCII . . . 1186
19.8 quopri — Codificar y decodificar datos MIME imprimibles entre comillas . . . 1188

20 Herramientas Para Procesar Formatos de Marcado Estructurado 1191
20.1 html — Compatibilidad con el Lenguaje de marcado de hipertexto . . . 1191
20.2 html.parser — Analizador simple de HTML y XHTML . . . 1192
20.2.1 Aplicación ejemplo de un analizador sintáctico (*parser*) de HTML . . . 1192
20.2.2 Métodos HTMLParser . . . 1193
20.2.3 Ejemplos . . . 1194
20.3 html.entities — Definiciones de entidades generales HTML . . . 1196
20.4 Módulos de procesamiento XML . . . 1197
20.4.1 Vulnerabilidades XML . . . 1197
20.4.2 El paquete defusedxml . . . 1198
20.5 xml.etree.ElementTree — La API XML de ElementTree . . . 1198
20.5.1 Tutorial . . . 1199
20.5.2 Soporte de XPath . . . 1204
20.5.3 Referencia . . . 1206
20.5.4 Soporte de XInclude . . . 1209
20.5.5 Referencia . . . 1210
20.6 xml.dom — El *API* del Modelo de Objetos del Documento . . . 1218
20.6.1 Contenido del Módulo . . . 1219
20.6.2 Objetos en el DOM . . . 1219
20.6.3 Conformidad . . . 1227
20.7 xml.dom.minidom — Implementación mínima del DOM . . . 1228

20.7.1 Objetos del DOM . . . 1229
20.7.2 Ejemplo de DOM . . . 1231
20.7.3 minidom y el estándar DOM . . . 1232
20.8 xml.dom.pulldom — Soporte para la construcción parcial de árboles DOM . . . 1233
20.8.1 Objetos DOMEventStream . . . 1234
20.9 XML.sax— Soporte para analizadores SAX2 . . . 1235
20.9.1 Objetos SAXException . . . 1236
20.10 xml.sax.handler — Base classes for SAX handlers . . . 1236
20.10.1 Objetos ContentHandler . . . 1239
20.10.2 Objetos DTDHandler . . . 1241
20.10.3 Objetos EntityResolver . . . 1241
20.10.4 Objetos ErrorHandler . . . 1241
20.10.5 Objetos DTDHandler . . . 1241
20.11 xml.sax.saxutils — Utilidades SAX . . . 1242
20.12 xml.sax.xmlreader — Interfaz para analizadores XML . . . 1243
20.12.1 Objetos XMLReader . . . 1244
20.12.2 Objetos IncrementalParser . . . 1245
20.12.3 Objetos localizadores . . . 1245
20.12.4 Objetos InputSource . . . 1246
20.12.5 La Interfaz Attributes . . . 1246
20.12.6 La Interfaz AttributesNS . . . 1247
20.13 xml.parsers.expat — Análisis rápido XML usando Expat . . . 1247
20.13.1 Objetos XMLParser . . . 1248
20.13.2 Excepciones de ExpatError . . . 1252
20.13.3 Ejemplo . . . 1252
20.13.4 Descripciones del modelo de contenido . . . 1253
20.13.5 Constantes de error de expansión . . . 1254

21 Protocolos y soporte de Internet 1257
21.1 webbrowser — Controlador de navegador web conveniente . . . 1257
21.1.1 Objetos controladores de navegador . . . 1260
21.2 wsgiref — Utilidades WSGI e implementación de referencia . . . 1260
21.2.1 wsgiref.util – Utilidades de entorno WSGI . . . 1260
21.2.2 wsgiref.headers – Herramientas para cabeceras de respuesta WSGI . . . 1262
21.2.3 wsgiref.simple_server– Un servidor HTTP WSGI simple . . . 1263
21.2.4 wsgiref.validate — Verificador de compatibilidad WSGI . . . 1264
21.2.5 wsgiref.handlers – Clases base servidor/*gateway* . . . 1265
21.2.6 Ejemplos . . . 1269
21.3 urllib — URL módulos de manipulación . . . 1270
21.4 urllib.request — Biblioteca extensible para abrir URLs . . . 1270
21.4.1 Objetos Request . . . 1275
21.4.2 Objetos OpenerDirector . . . 1276
21.4.3 Objetos BaseHandler . . . 1277
21.4.4 Objetos HTTPRedirectHandler . . . 1279
21.4.5 Objetos HTTPCookieProcessor . . . 1279
21.4.6 Objetos ProxyHandler . . . 1279
21.4.7 Objetos HTTPPasswordMgr . . . 1280
21.4.8 Objetos HTTPPasswordMgrWithPriorAuth . . . 1280
21.4.9 Objetos AbstractBasicAuthHandler . . . 1280
21.4.10 Objetos HTTPBasicAuthHandler . . . 1281
21.4.11 Objetos ProxyBasicAuthHandler . . . 1281
21.4.12 Objetos AbstractDigestAuthHandler . . . 1281
21.4.13 Objetos HTTPDigestAuthHandler . . . 1281
21.4.14 Objetos ProxyDigestAuthHandler . . . 1281
21.4.15 Objetos HTTPHandler . . . 1281
21.4.16 Objetos HTTPSHandler . . . 1281
21.4.17 Objetos FileHandler . . . 1281
21.4.18 Objetos DataHandler . . . 1282

21.4.19 Objetos FTPHandler . . . 1282
21.4.20 Objetos CacheFTPHandler . . . 1282
21.4.21 Objetos UnknownHandler . . . 1282
21.4.22 Objetos HTTPErrorProcessor . . . 1282
21.4.23 Ejemplos . . . 1282
21.4.24 Interfaz heredada . . . 1285
21.4.25 Restricciones `urllib.request` . . . 1287
21.5 `urllib.response` — Clases de respuesta usadas por urllib . . . 1288
21.6 `urllib.parse` — Analiza URL en componentes . . . 1288
21.6.1 Análisis de URL . . . 1289
21.6.2 URL parsing security . . . 1293
21.6.3 Análisis de bytes codificados ASCII . . . 1294
21.6.4 Resultados del análisis estructurado . . . 1294
21.6.5 Cita de URL . . . 1295
21.7 `urllib.error` — Clases de excepción lanzadas por urllib.request . . . 1297
21.8 `urllib.robotparser` — Analizador para robots.txt . . . 1298
21.9 `http` — Módulos HTTP . . . 1299
21.9.1 Códigos de estado HTTP . . . 1300
21.10 `http.client` — Cliente de protocolo HTTP . . . 1301
21.10.1 Objetos de `HTTPConnection` . . . 1304
21.10.2 Objetos de `HTTPResponse` . . . 1306
21.10.3 Ejemplos . . . 1307
21.10.4 Objetos de `HTTPMessage` . . . 1308
21.11 `ftplib` — cliente de protocolo FTP . . . 1308
21.11.1 Objetos FTP . . . 1310
21.11.2 Objetos FTP_TLS . . . 1313
21.12 `poplib` — Cliente de protocolo POP3 . . . 1313
21.12.1 Objetos POP3 . . . 1314
21.12.2 Ejemplo POP3 . . . 1316
21.13 `imaplib` — Protocolo del cliente IMAP4 . . . 1316
21.13.1 Objetos de IMAP4 . . . 1318
21.13.2 Ejemplo IMAP4 . . . 1323
21.14 `smtplib` — Cliente de protocolo SMTP . . . 1323
21.14.1 Objetos SMTP . . . 1325
21.14.2 Ejemplo SMTP . . . 1329
21.15 `uuid` — objetos UUID según **RFC 4122** . . . 1329
21.15.1 Ejemplo . . . 1332
21.16 `socketserver` — Un framework para servidores de red . . . 1333
21.16.1 Notas de creación del servidor . . . 1334
21.16.2 Objetos de servidor . . . 1335
21.16.3 Solicitar objetos de controlador . . . 1337
21.16.4 Ejemplos . . . 1337
21.17 `http.server` — Servidores HTTP . . . 1341
21.17.1 Security Considerations . . . 1346
21.18 `http.cookies` — Gestión del estado HTTP . . . 1347
21.18.1 Objetos de cookie . . . 1347
21.18.2 Objetos Morsel . . . 1348
21.18.3 Ejemplo . . . 1349
21.19 `http.cookiejar` — Manejo de cookies para clientes HTTP . . . 1350
21.19.1 Objetos CookieJar y FileCookieJar . . . 1352
21.19.2 Subclases FileCookieJar y co-operación con navegadores web . . . 1354
21.19.3 Objetos CookiePolicy . . . 1354
21.19.4 Objetos DefaultCookiePolicy . . . 1355
21.19.5 Objetos Cookie . . . 1357
21.19.6 Ejemplos . . . 1358
21.20 `xmlrpc` — Módulos XMLRPC para cliente y servidor . . . 1359
21.21 `xmlrpc.client` — acceso cliente XML-RPC . . . 1359
21.21.1 Objetos *ServerProxy* . . . 1361

21.21.2 Objetos *DateTime* . . . 1362
21.21.3 Objetos binarios . . . 1362
21.21.4 Objetos Faults . . . 1363
21.21.5 Objetos ProtocolError . . . 1364
21.21.6 Objetos MultiCall . . . 1364
21.21.7 Funciones de Conveniencia . . . 1365
21.21.8 Ejemplo de Uso de Cliente . . . 1366
21.21.9 Ejemplo de Uso de Cliente y Servidor . . . 1366
21.22 `xmlrpc.server` — Servidores básicos XML-RPC . . . 1366
21.22.1 Objetos SimpleXMLRPCServer . . . 1367
21.22.2 CGIXMLRPCRequestHandler . . . 1370
21.22.3 Documentando el servidor XMLRPC . . . 1371
21.22.4 Objetos DocXMLRPCServer . . . 1371
21.22.5 DocCGIXMLRPCRequestHandler . . . 1372
21.23 `ipaddress` — Biblioteca de manipulación IPv4 / IPv6 . . . 1372
21.23.1 Funciones de fábrica de conveniencia . . . 1372
21.23.2 Direcciones IP . . . 1373
21.23.3 Definiciones de red IP . . . 1377
21.23.4 Objetos de interfaz . . . 1383
21.23.5 Otras funciones de nivel de módulo . . . 1384
21.23.6 Excepciones personalizadas . . . 1385

22 Servicios Multimedia **1387**
22.1 `wave` — Leer y escribir archivos WAV . . . 1387
22.1.1 Los objetos *Wave_read* . . . 1388
22.1.2 Los objetos *Wave_write* . . . 1388
22.2 `colorsys` — Conversiones entre sistemas de color . . . 1389

23 Internacionalización **1391**
23.1 `gettext` — Servicios de internacionalización multilingües . . . 1391
23.1.1 GNU API **`gettext`** . . . 1391
23.1.2 API basada en clases . . . 1393
23.1.3 Internacionalizando sus programas y módulos . . . 1397
23.1.4 Agradecimientos . . . 1400
23.2 `locale` — Servicios de internacionalización . . . 1400
23.2.1 Segundo plano, detalles, indicaciones, consejos y advertencias . . . 1406
23.2.2 Para escritores de extensión y programas que incrustan Python . . . 1407
23.2.3 Acceso a los catálogos de mensajes . . . 1407

24 *Frameworks* de programa **1409**
24.1 `turtle` — Gráficos con *Turtle* . . . 1409
24.1.1 Introducción . . . 1409
24.1.2 Reseña de los métodos disponibles para Turtle y *Screen* . . . 1411
24.1.3 Métodos de *RawTurtle/Turtle* Y sus correspondientes funciones . . . 1414
24.1.4 Métodos de *TurtleScreen/Screen* y sus correspondientes funciones . . . 1430
24.1.5 Clases públicas . . . 1437
24.1.6 Ayuda y configuración . . . 1438
24.1.7 `turtledemo` — Scripts de demostración . . . 1441
24.1.8 Cambios desde Python 2.6 . . . 1442
24.1.9 Cambios desde Python 3.0 . . . 1443
24.2 `cmd` — Soporte para intérpretes orientados a línea de comandos . . . 1443
24.2.1 Objetos Cmd . . . 1443
24.2.2 Ejemplo Cmd . . . 1445
24.3 `shlex` — Análisis léxico simple . . . 1448
24.3.1 objetos `shlex` . . . 1450
24.3.2 Reglas de análisis . . . 1452
24.3.3 Compatibilidad mejorada con intérprete de comandos . . . 1452

25 Interfaces gráficas de usuario con Tk **1455**

25.1 tkinter — Interface de Python para Tcl/Tk . . . 1455
25.1.1 Arquitectura . . . 1456
25.1.2 Módulos Tkinter . . . 1457
25.1.3 Guía de supervivencia de Tkinter . . . 1458
25.1.4 Modelo de subprocesamiento . . . 1461
25.1.5 Guía práctica . . . 1462
25.1.6 Gestor de archivos . . . 1467
25.2 tkinter.colorchooser — Diálogo de elección de color . . . 1468
25.3 tkinter.font — Envoltorio de fuente Tkinter . . . 1468
25.4 Diálogos Tkinter . . . 1469
25.4.1 tkinter.simpledialog —Diálogos de entrada estándar de Tkinter . . . 1469
25.4.2 Diálogos de selección de archivos . . . 1470
25.4.3 tkinter.commondialog — Plantillas de ventanas de diálogo . . . 1472
25.5 tkinter.messagebox — Indicadores de mensajes de Tkinter . . . 1472
25.6 tkinter.scrolledtext — Widget de texto desplazado . . . 1473
25.7 tkinter.dnd — Soporte de arrastrar y soltar . . . 1473
25.8 tkinter.ttk — Tk widgets temáticos . . . 1474
25.8.1 Uso de Ttk . . . 1474
25.8.2 Ttk widgets . . . 1475
25.8.3 Widget . . . 1475
25.8.4 Combobox . . . 1478
25.8.5 Spinbox . . . 1479
25.8.6 Notebook . . . 1480
25.8.7 Progressbar . . . 1483
25.8.8 Separator . . . 1484
25.8.9 Sizegrip . . . 1484
25.8.10 Treeview . . . 1484
25.8.11 Ttk Styling . . . 1490
25.9 tkinter.tix — Ampliación de widgets para Tk . . . 1493
25.9.1 Usando Tix . . . 1493
25.9.2 Widgets de Tix . . . 1494
25.9.3 Comandos Tix . . . 1496
25.10 IDLE . . . 1498
25.10.1 Menús . . . 1498
25.10.2 Editing and Navigation . . . 1502
25.10.3 Startup and Code Execution . . . 1505
25.10.4 Help and Preferences . . . 1509
25.10.5 idlelib . . . 1510

26 Herramientas de desarrollo 1511
26.1 typing — Soporte para *type hints* . . . 1511
26.1.1 Relevant PEPs . . . 1512
26.1.2 Alias de tipo . . . 1512
26.1.3 NewType . . . 1513
26.1.4 Callable . . . 1514
26.1.5 Genéricos . . . 1515
26.1.6 Tipos genéricos definidos por el usuario . . . 1515
26.1.7 El tipo Any . . . 1517
26.1.8 Subtipado nominal vs estructural . . . 1519
26.1.9 Contenido del módulo . . . 1519
26.2 pydoc — Generador de documentación y Sistema de ayuda en línea . . . 1544
26.3 Modo de desarrollo de Python . . . 1545
26.4 Efectos del modo de desarrollo de Python . . . 1545
26.5 Ejemplo de ResourceWarning . . . 1546
26.6 Ejemplo de error de descriptor de archivo incorrecto . . . 1547
26.7 doctest – Prueba ejemplos interactivos de Python . . . 1548
26.7.1 Uso simple: verificar ejemplos en docstrings . . . 1550
26.7.2 Uso Simple: Verificar ejemplos en un Archivo de Texto . . . 1551

26.7.3 Cómo funciona . . . 1552
26.7.4 API básica . . . 1559
26.7.5 API de unittest . . . 1560
26.7.6 API avanzada . . . 1563
26.7.7 Depuración . . . 1567
26.7.8 Plataforma improvisada . . . 1570
26.8 unittest — Infraestructura de tests unitarios . . . 1571
26.8.1 Ejemplo sencillo . . . 1572
26.8.2 Interfaz de línea de comandos . . . 1573
26.8.3 Descubrimiento de pruebas . . . 1574
26.8.4 Organización del código de pruebas . . . 1575
26.8.5 Reutilización de código de prueba anterior . . . 1577
26.8.6 Omitir tests y fallos esperados . . . 1578
26.8.7 Distinguiendo iteraciones de tests empleando subtests . . . 1579
26.8.8 Clases y funciones . . . 1580
26.8.9 Instalaciones para clases y módulos . . . 1599
26.8.10 Manejo de señales . . . 1600
26.9 unittest.mock — Biblioteca de objetos simulados . . . 1601
26.9.1 Guía rápida . . . 1602
26.9.2 La clase Mock . . . 1604
26.9.3 Parcheadores . . . 1620
26.9.4 MagicMock y el soporte de métodos mágicos . . . 1629
26.9.5 Ayudantes . . . 1632
26.10 unittest.mock — primeros pasos . . . 1641
26.10.1 Usando Mock . . . 1641
26.10.2 Decoradores de Parches . . . 1646
26.10.3 Otros Ejemplos . . . 1648
26.11 2to3 — Automated Python 2 to 3 code translation . . . 1660
26.11.1 Usando 2to3 . . . 1660
26.11.2 Fixers . . . 1662
26.11.3 lib2to3 — 2to3's library . . . 1665
26.12 test — Paquete de pruebas de regresión para Python . . . 1666
26.12.1 Escritura de pruebas unitarias para el paquete test . . . 1666
26.12.2 Ejecución de pruebas utilizando la interfaz de línea de comandos . . . 1668
26.13 test.support — Utilidades para el conjunto de pruebas de Python . . . 1668
26.14 test.support.socket_helper — Utilidades para pruebas de socket . . . 1677
26.15 test.support.script_helper —Utilidades para las pruebas de ejecución de Python . . . 1677
26.16 test.support.bytecode_helper — Herramientas de apoyo para comprobar la correcta generación de bytecode . . . 1679
26.17 test.support.threading_helper — Utilidades para pruebas con hilos . . . 1679
26.18 test.support.os_helper — Utilidades para pruebas de sistemas operativos . . . 1680
26.19 test.support.import_helper — Utilidades para pruebas de importación . . . 1682
26.20 test.support.warnings_helper — Utilidades para pruebas de advertencias . . . 1683

27 Depuración y perfilado 1685
27.1 Tabla de auditoría de eventos . . . 1685
27.2 bdb — Framework de depuración . . . 1689
27.3 faulthandler — Volcar el rastreo de Python . . . 1694
27.3.1 Volcar el rastreo . . . 1695
27.3.2 Estado del gestor de fallos . . . 1695
27.3.3 Volcar los rastreos después de un tiempo de espera . . . 1695
27.3.4 Volcar el rastreo en una señal del usuario . . . 1696
27.3.5 Problema con descriptores de archivo . . . 1696
27.3.6 Ejemplo . . . 1696
27.4 pdb — El Depurador de Python . . . 1696
27.4.1 Comandos del depurador . . . 1699
27.5 Los perfiladores de Python . . . 1703
27.5.1 Introducción a los perfiladores . . . 1703

27.5.2 Manual instantáneo de usuario . . . 1703
27.5.3 Referencia del módulo profile y cProfile . . . 1705
27.5.4 La clase Stats . . . 1707
27.5.5 ¿Qué es el perfil determinista? . . . 1709
27.5.6 Limitaciones . . . 1710
27.5.7 Calibración . . . 1710
27.5.8 Usando un temporizador personalizado . . . 1711
27.6 timeit — Mide el tiempo de ejecución de pequeños fragmentos de código . . . 1711
27.6.1 Ejemplos básicos . . . 1712
27.6.2 Interfaz de Python . . . 1712
27.6.3 Interfaz de línea de comandos . . . 1714
27.6.4 Ejemplos . . . 1715
27.7 trace — Rastrear la ejecución de la declaración de Python . . . 1716
27.7.1 Uso de la línea de comandos . . . 1716
27.7.2 Interfaz programática . . . 1718
27.8 tracemalloc— Rastrea la asignación de memoria . . . 1719
27.8.1 Ejemplos . . . 1719
27.8.2 API . . . 1724

28 Empaquetado y distribución de software 1731
28.1 distutils — Creación e instalación de módulos Python . . . 1731
28.2 ensurepip — Ejecutando el instalador pip . . . 1732
28.2.1 Interfaz de línea de comandos . . . 1732
28.2.2 API del módulo . . . 1733
28.3 venv — Creación de entornos virtuales . . . 1733
28.3.1 Creación de entornos virtuales . . . 1734
28.3.2 How venvs work . . . 1735
28.3.3 API . . . 1736
28.3.4 Un ejemplo de la extensión de EnvBuilder . . . 1739
28.4 zipapp — Gestiona archivadores zip ejecutables de Python . . . 1742
28.4.1 Ejemplo básico . . . 1743
28.4.2 Interfaz de línea de comando . . . 1743
28.4.3 API de Python . . . 1744
28.4.4 Ejemplos . . . 1745
28.4.5 Especificar el intérprete . . . 1745
28.4.6 Creando aplicaciones independientes con zipapp . . . 1746
28.4.7 El formato de archivado Zip de aplicaciones Python . . . 1748

29 Servicios en tiempo de ejecución de Python 1749
29.1 sys — Parámetros y funciones específicos del sistema . . . 1749
29.2 sysconfig — Proporciona acceso a la información de configuración de Python . . . 1769
29.2.1 Variables de configuración . . . 1769
29.2.2 Rutas de instalación . . . 1770
29.2.3 Otras funciones . . . 1771
29.2.4 Usando sysconfig como un script . . . 1772
29.3 builtins — Objetos incorporados . . . 1773
29.4 __main__ — Entorno de código de nivel máximo . . . 1773
29.4.1 __name__ == '__main__' . . . 1774
29.4.2 __main__.py en paquetes de Python . . . 1776
29.4.3 import __main__ . . . 1777
29.5 warnings — Control de advertencias . . . 1779
29.5.1 Categorías de advertencia . . . 1779
29.5.2 El filtro de advertencias . . . 1780
29.5.3 Eliminación temporal de las advertencias . . . 1782
29.5.4 Advertencias de prueba . . . 1783
29.5.5 Actualización del código para las nuevas versiones de las dependencias . . . 1783
29.5.6 Funciones Disponibles . . . 1784
29.5.7 Gestores de Contexto disponibles . . . 1785

29.6 dataclasses — Clases de datos . . . 1785
29.6.1 Contenidos del módulo . . . 1786
29.6.2 Procesamiento posterior a la inicialización . . . 1791
29.6.3 Variables de clase . . . 1792
29.6.4 Variable de solo inicialización . . . 1792
29.6.5 Instancias congeladas . . . 1793
29.6.6 Herencia . . . 1793
29.6.7 Reordenar los parámetros de solo palabras clave en __init__() . . . 1793
29.6.8 Funciones fábrica por defecto . . . 1794
29.6.9 Valores por defecto mutables . . . 1794
29.6.10 Descriptor-typed fields . . . 1795
29.7 contextlib — Utilidades para declaraciones de contexto with . . . 1796
29.7.1 Utilidades . . . 1796
29.7.2 Ejemplos y recetas . . . 1804
29.7.3 Gestores de contexto de uso único, reutilizables y reentrantes . . . 1807
29.8 abc — Clases de Base Abstracta . . . 1810
29.9 atexit — Gestores de Salida . . . 1814
29.9.1 Ejemplo con atexit . . . 1815
29.10 traceback — Imprimir o recuperar un seguimiento de pila . . . 1816
29.10.1 Objetos TracebackException . . . 1818
29.10.2 Objetos StackSummary . . . 1819
29.10.3 Objetos FrameSummary . . . 1820
29.10.4 Ejemplos de seguimiento de pila . . . 1820
29.11 __future__ — Definiciones de declaraciones futuras . . . 1822
29.12 gc — Interfaz del recolector de basura . . . 1824
29.13 inspect — Inspeccionar objetos vivos . . . 1827
29.13.1 Tipos y miembros . . . 1828
29.13.2 Recuperar el código fuente . . . 1831
29.13.3 Introspección de los invocables con el objeto Signature . . . 1832
29.13.4 Clases y funciones . . . 1837
29.13.5 La pila del interprete . . . 1840
29.13.6 Obteniendo atributos estáticamente . . . 1841
29.13.7 Estado actual de los Generadores y las Corutinas . . . 1842
29.13.8 Objetos de código Bit Flags . . . 1843
29.13.9 Interfaz de la línea de comando . . . 1844
29.14 site — Enlace (*hook*) de configuración específico del sitio . . . 1844
29.14.1 Configuración de *Readline* . . . 1845
29.14.2 Contenido del módulo . . . 1846
29.14.3 Interfaz de línea de comando . . . 1847

30 Intérpretes de Python personalizados 1849
30.1 code — Clases básicas de intérpretes . . . 1849
30.1.1 Objetos de intérprete interactivo . . . 1850
30.1.2 Objetos de consola interactiva . . . 1851
30.2 codeop — Compila código Python . . . 1851

31 Importando módulos 1853
31.1 zipimport — Importar módulos desde archivos zip . . . 1853
31.1.1 Objetos zipimporter . . . 1854
31.1.2 Ejemplos . . . 1855
31.2 pkgutil — Utilidad de extensión de paquete . . . 1855
31.3 modulefinder — Buscar módulos usados por un script . . . 1859
31.3.1 Ejemplo de uso de ModuleFinder . . . 1859
31.4 runpy — Localización y ejecución de módulos *Python* . . . 1860
31.5 importlib — La implementación de import . . . 1862
31.5.1 Introducción . . . 1862
31.5.2 Funciones . . . 1863
31.5.3 importlib.abc – Clases base abstractas relacionadas con la importación . . . 1865

31.5.4 importlib.resources – Recursos . . . 1872
31.5.5 importlib.machinery – Importadores y enlaces de ruta . . . 1874
31.5.6 importlib.util – Código de utilidad para importadores . . . 1878
31.5.7 Ejemplos . . . 1881
31.6 Usando importlib.metadata . . . 1884
31.6.1 Descripción general . . . 1884
31.6.2 API funcional . . . 1885
31.6.3 Distribuciones . . . 1888
31.6.4 Extendiendo el algoritmo de búsqueda . . . 1888

32 Servicios del lenguaje Python **1889**
32.1 ast — Árboles de sintaxis abstracta . . . 1889
32.1.1 Gramática abstracta . . . 1889
32.1.2 Clases Nodo . . . 1892
32.1.3 Ayudantes de ast . . . 1917
32.1.4 Banderas del compilador . . . 1921
32.1.5 Uso en línea de comandos . . . 1921
32.2 symtable — Acceso a la tabla de símbolos del compilador . . . 1922
32.2.1 Generando tablas de símbolos . . . 1922
32.2.2 Examinando la tabla de símbolos . . . 1922
32.3 token— Constantes usadas con árboles de sintaxis de Python . . . 1924
32.4 keyword — Pruebas para palabras clave en Python . . . 1927
32.5 tokenize — Conversor a tokens para código Python . . . 1928
32.5.1 Convirtiendo la entrada en *tokens* . . . 1928
32.5.2 Uso como línea de comandos . . . 1929
32.5.3 Ejemplos . . . 1930
32.6 tabnanny — Detección de indentación ambigua . . . 1932
32.7 pyclbr — Soporte para navegador de módulos Python . . . 1932
32.7.1 Objetos Function . . . 1933
32.7.2 Objetos Class . . . 1933
32.8 py_compile` — Compila archivos fuente Python . . . 1934
32.8.1 Interfaz de línea de comandos . . . 1935
32.9 compileall — Bibliotecas de Python de compilación de bytes . . . 1936
32.9.1 Uso de la línea de comandos . . . 1936
32.9.2 Funciones públicas . . . 1937
32.10 dis — Desensamblador para bytecode de Python . . . 1940
32.10.1 Análisis de bytecode . . . 1940
32.10.2 Funciones de análisis . . . 1941
32.10.3 Instrucciones bytecode de Python . . . 1943
32.10.4 Colecciones opcode . . . 1953
32.11 pickletools — Herramientas para desarrolladores pickle . . . 1953
32.11.1 Uso de la línea de comandos . . . 1954
32.11.2 Interfaz programática . . . 1954

33 Servicios Específicos para MS Windows **1957**
33.1 msvcrt — Rutinas útiles del entorno de ejecución MS VC++ . . . 1957
33.1.1 Operaciones con archivos . . . 1957
33.1.2 Consola E/S . . . 1958
33.1.3 Otras funciones . . . 1959
33.2 winreg — Acceso al registro de Windows . . . 1959
33.2.1 Funciones . . . 1959
33.2.2 Constantes . . . 1965
33.2.3 Objetos de control del registro . . . 1967
33.3 :mod:”winsound” — Interfaz de reproducción de sonido para Windows . . . 1968

34 Servicios específicos de Unix **1971**
34.1 posix — Las llamadas más comunes al sistema POSIX . . . 1971
34.1.1 Soporte de archivos grandes . . . 1971
34.1.2 Contenido notable del módulo . . . 1972

34.2 pwd — La base de datos de contraseñas . . . 1972
34.3 grp — La base de datos de grupo . . . 1973
34.4 termios —Control tty estilo POSIX . . . 1974
34.4.1 Ejemplo . . . 1974
34.5 tty — Funciones de control de terminal . . . 1975
34.6 pty — Utilidades para Pseudo-terminal . . . 1975
34.6.1 Ejemplo . . . 1976
34.7 fcntl — Las llamadas a sistema fcntl y ioctl . . . 1977
34.8 resource — Información sobre el uso de recursos . . . 1979
34.8.1 Límites de recursos . . . 1979
34.8.2 Utilización de recursos . . . 1982
34.9 syslog — Rutinas de la biblioteca syslog de Unix . . . 1983
34.9.1 Ejemplos . . . 1984

35 Módulos Reemplazados **1985**
35.1 aifc — Lee y escribe archivos AIFF y AIFC . . . 1985
35.2 asynchat — Gestor de comandos/respuestas en *sockets* asíncronos . . . 1987
35.2.1 Ejemplo de asynchat . . . 1989
35.3 asyncore — controlador de socket asincrónico . . . 1990
35.3.1 Ejemplo asyncore de cliente HTTP básico . . . 1993
35.3.2 Ejemplo asyncore de servidor de eco básico . . . 1994
35.4 audioop — Manipula datos de audio sin procesar . . . 1994
35.5 cgi — Soporte de Interfaz de Entrada Común (CGI) . . . 1997
35.5.1 Introducción . . . 1998
35.5.2 Usando el módulo CGI . . . 1998
35.5.3 Interfaz de Nivel Superior . . . 2000
35.5.4 Funciones . . . 2001
35.5.5 Preocuparse por la seguridad . . . 2002
35.5.6 Instalando su script de CGI en un sistema Unix . . . 2002
35.5.7 Probando su script de CGI . . . 2002
35.5.8 Depurando scripts de CGI . . . 2003
35.5.9 Problemas comunes y soluciones . . . 2004
35.6 cgitb — Administrador *traceback* para scripts CGI. . . . 2004
35.7 chunk — Lee los datos de los trozos de IFF . . . 2005
35.8 crypt — Función para verificar contraseñas Unix . . . 2006
35.8.1 Métodos de *hashing* . . . 2007
35.8.2 Atributos del módulo . . . 2007
35.8.3 Funciones del módulo . . . 2007
35.8.4 Ejemplos . . . 2008
35.9 :mod:”imghdr” — Determinar el tipo de imagen . . . 2008
35.10 imp — Acceda a import internamente . . . 2009
35.10.1 Ejemplos . . . 2013
35.11 mailcap — Manejo de archivos Mailcap . . . 2014
35.12 msilib — Leer y escribir archivos *Microsoft Installer* . . . 2015
35.12.1 Objetos Database . . . 2016
35.12.2 Objetos View . . . 2017
35.12.3 Objetos Summary Information . . . 2017
35.12.4 Objetos Record . . . 2018
35.12.5 Errores . . . 2018
35.12.6 Objetos CAB . . . 2018
35.12.7 Objetos Directory . . . 2019
35.12.8 Features . . . 2019
35.12.9 Clases GUI . . . 2020
35.12.10Tablas pre-calculadas . . . 2021
35.13 nis — Interfaz a Sun’s NIS (Páginas amarillas) . . . 2021
35.14 nntplib — Protocolo de cliente NNTP . . . 2022
35.14.1 Objetos NNTP . . . 2024
35.14.2 Funciones de utilidad . . . 2028

35.15 optparse — Analizador sintáctico (parser) para opciones de línea de comandos . . . 2028
35.15.1 Contexto . . . 2030
35.15.2 Tutorial . . . 2032
35.15.3 Guía de referencia . . . 2039
35.15.4 Retrollamadas de opción . . . 2049
35.15.5 Extendiendo el módulo optparse . . . 2053
35.16 ossaudiodev — Acceso a dispositivos de audio compatibles con OSS . . . 2055
35.16.1 Objetos de dispositivo de audio . . . 2056
35.16.2 Objetos del dispositivo mezclador . . . 2059
35.17 pipes — Interfaz para tuberías de shell . . . 2060
35.17.1 Objetos Template . . . 2060
35.18 smtpd — Servidor SMTP . . . 2061
35.18.1 Objetos SMTPServer . . . 2061
35.18.2 Objetos DebuggingServer . . . 2062
35.18.3 Objetos PureProxy . . . 2062
35.18.4 Objetos MailmanProxy . . . 2063
35.18.5 Objetos SMTPChannel . . . 2063
35.19 sndhdr — Determinar el tipo de archivo de sonido . . . 2064
35.20 spwd — La base de datos de contraseñas ocultas . . . 2065
35.21 sunau — Lectura y escritura de ficheros Sun AU . . . 2066
35.21.1 Objetos AU_read . . . 2067
35.21.2 Objetos AU_write . . . 2068
35.22 telnetlib — cliente Telnet . . . 2068
35.22.1 Objetos Telnet . . . 2069
35.22.2 Ejemplo de Telnet . . . 2071
35.23 uu — Codifica y decodifica archivos UUEncode . . . 2071
35.24 xdrlib — Codificar y decodificar datos XDR . . . 2072
35.24.1 Instancias de la clase *Packer* . . . 2072
35.24.2 Instancias de la clase *Unpacker* . . . 2073
35.24.3 Excepciones . . . 2075

36 Consideraciones de seguridad **2077**

A Glosario **2079**

B Acerca de estos documentos **2093**
B.1 Contribuidores de la documentación de Python . . . 2093

C Historia y Licencia **2095**
C.1 Historia del software . . . 2095
C.2 Términos y condiciones para acceder o usar Python . . . 2096
C.2.1 ACUERDO DE LICENCIA DE PSF PARA PYTHON | lanzamiento | . . . 2096
C.2.2 ACUERDO DE LICENCIA DE BEOPEN.COM PARA PYTHON 2.0 . . . 2097
C.2.3 ACUERDO DE LICENCIA CNRI PARA PYTHON 1.6.1 . . . 2098
C.2.4 ACUERDO DE LICENCIA CWI PARA PYTHON 0.9.0 HASTA 1.2 . . . 2099
C.2.5 LICENCIA BSD DE CLÁUSULA CERO PARA CÓDIGO EN EL PYTHON | lanzamiento | DOCUMENTACIÓN . . . 2099
C.3 Licencias y reconocimientos para software incorporado . . . 2100
C.3.1 Mersenne Twister . . . 2100
C.3.2 Sockets . . . 2101
C.3.3 Servicios de socket asincrónicos . . . 2101
C.3.4 Gestión de cookies . . . 2102
C.3.5 Seguimiento de ejecución . . . 2102
C.3.6 funciones UUencode y UUdecode . . . 2103
C.3.7 Llamadas a procedimientos remotos XML . . . 2103
C.3.8 test_epoll . . . 2104
C.3.9 Seleccionar kqueue . . . 2104
C.3.10 SipHash24 . . . 2105
C.3.11 strtod y dtoa . . . 2105

C.3.12 OpenSSL . . . 2106
C.3.13 expat . . . 2108
C.3.14 libffi . . . 2108
C.3.15 zlib . . . 2109
C.3.16 cfuhash . . . 2109
C.3.17 libmpdec . . . 2110
C.3.18 Conjunto de pruebas W3C C14N . . . 2110
C.3.19 Audioop . . . 2111

D Derechos de autor **2113**

Bibliografía **2115**

Índice de Módulos Python **2117**

Índice **2121**

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 un 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.

CAPÍTULO 1

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 *tienes* 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 que afirman «Disponibilidad: Unix» son compatibles con macOS, que se basa en un núcleo de Unix.

CAPÍTULO 2

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

A	**E**	**L**	**R**
abs()	*enumerate()*	*len()*	*range()*
aiter()	*eval()*	*list()*	*repr()*
all()	*exec()*	*locals()*	*reversed()*
any()			*round()*
anext()	**F**	**M**	
ascii()	*filter()*	*map()*	**S**
	float()	*max()*	*set()*
B	*format()*	*memoryview()*	*setattr()*
bin()	*frozenset()*	*min()*	*slice()*
bool()			*sorted()*
breakpoint()	**G**	**N**	*staticmethod()*
bytearray()	*getattr()*	*next()*	*str()*
bytes()	*globals()*		*sum()*
		O	*super()*
C	**H**	*object()*	
callable()	*hasattr()*	*oct()*	**T**
chr()	*hash()*	*open()*	*tuple()*
classmethod()	*help()*	*ord()*	*type()*
compile()	*hex()*		
complex()		**P**	**V**
	I	*pow()*	*vars()*
D	*id()*	*print()*	
delattr()	*input()*	*property()*	**Z**
dict()	*int()*		*zip()*
dir()	*isinstance()*		
divmod()	*issubclass()*		**_**
	iter()		*__import__()*

abs(*x*)

Retorna el valor absoluto de un número. El argumento puede ser un número entero, un número de coma flotante o un objeto que implemente `__abs__()`. Si el argumento es un número complejo, se retorna su magnitud.

aiter(*async_iterable*)

Retorna un *asynchronous iterator* para un *asynchronous iterable*. Equivalente a llamar `x.__aiter__()`.

Nota: A diferencia de *iter()*, *aiter()* no tiene una variante con 2 argumentos.

Nuevo en la versión 3.10.

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
```

awaitable anext(*async_iterator*[, *default*])

Cuando está esperado, retorna el siguiente elemento del *asynchronous iterator* otorgado, o *default* (por defecto) si se da y el iterador está agotado.

Esta es la variante asincrónica del *next()* incorporado, y se comporta de manera similar.

Esto llama al método __anext__() de *async_iterator*, retornando un *awaitable*. A la espera de esto se retorna el siguiente valor del iterador. Si se otorga *default* (por defecto), se retorna si el iterador está agotado, de lo contrario se lanza *StopAsyncIteration*.

Nuevo en la versión 3.10.

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*)

This function drops you into the debugger at the call site. Specifically, it calls *sys.breakpointhook()*, passing `args` and `kws` straight through. By default, `sys.breakpointhook()` calls *pdb.set_trace()* expecting no arguments. In this case, it is purely a convenience function so you don't have to explicitly import *pdb* or type as much code to enter the debugger. However, *sys.breakpointhook()* can be set to some other function and *breakpoint()* will automatically call that, allowing you to drop into the debugger of choice. If *sys.breakpointhook()* is not accessible, this function will raise *RuntimeError*.

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 <=

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.
- Si es un objeto conforme a interfaz de búfer, se utilizará un búfer de solo lectura del objeto para inicializar el arreglo de bytes.
- Si es un *iterable*, debe ser un iterable de enteros en el rango `0 <= 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 <= 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: Está 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 los desea, consulte `staticmethod()` en esta sección. Para obtener más información sobre los métodos de clase, consulte types.

Distinto en la versión 3.9: Los métodos de clase ahora pueden envolver otros *descriptores* como `property()`.

Distinto en la versión 3.10: Los métodos de clase ahora heredan los atributos de método (__module__, __name__, __qualname__, __doc__ and __annotations__) y tienen un nuevo atributo __wrapped__.

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 qué *opciones del compilador* deben activarse y cuáles características futuras deben permitirse. Si ninguno está presente (o ambos son cero), el código se compila con los mismos flags que afectan al código que llama *compile()*. Si se proporciona el argumento *flags* y *dont_inherit* no es (o es cero), las opciones del compilador y las declaraciones futuras especificadas por el argumento *flags* se utilizan además de las que se utilizarían de todos modos. Si *dont_inherit* es un número entero distinto de cero, entonces el argumento *flags* lo es: los indicadores (características futuras y opciones del compilador) en el código circundante se ignoran.

Las opciones del compilador y las declaraciones futuras se especifican mediante bits que pueden combinarse mediante OR bit a bit para especificar varias opciones. El campo de bits requerido para especificar una característica futura dada se puede encontrar como el atributo compiler_flag en la instancia _Feature en el módulo *__future__*. *Las flags del compilador* se pueden encontrar en el módulo *ast*, con el prefijo PyCF_.

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, filename.

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' ya no tiene que terminar necesariamente 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*)
This is a relative of `setattr()`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, 'foobar')` is equivalent to `del x.foobar`. *name* need not be a Python identifier (see `setattr()`).

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__()` personalizada.

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', 'calcsize', '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 metaclase 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 <= abs(a % b) < 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 analiza y evalúa como una expresión de Python (técnicamente hablando, 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 analizada. De esa forma se puede controlar qué módulos incorporados están disponibles para el código ejecutado insertando un diccionario propio `__builtins__` dentro de *globals* antes de pasarlo a la *eval()*. 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. Tener en cuenta que *eval()* no tiene acceso al *nested scopes* (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
```

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

Consejos: 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, respectivamente, lo cual puede ser útil para su uso en *eval()* o *exec()*.

Si la fuente dada es una cadena de caracteres, se eliminan los espacios y tabulaciones principales y finales.

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.

[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.

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*.

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be `'+'` or `'-'`; a `'+'` sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or positive or negative infinity. More precisely, the input must conform to the `floatvalue` production rule in the following grammar, after leading and trailing whitespace characters are removed:

```
sign          ::=   "+" | "-"
infinity      ::=   "Infinity" | "inf"
nan           ::=   "nan"
digitpart     ::=   digit (["_"] digit)*
number        ::=   [digitpart] "." digitpart | digitpart ["."]
exponent      ::=   ("e" | "E") ["+" | "-"] digitpart
floatnumber   ::=   number [exponent]
floatvalue    ::=   [sign] (floatnumber | infinity | nan)
```

Here `digit` is a Unicode decimal digit (character in the Unicode general category `Nd`). Case is not significant, so, for example, «inf», «Inf», «INFINITY», and «iNfINity» are all acceptable spellings for positive infinity.

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 lanzará 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 incorporados: *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*])
Return the value of the named attribute of *object*. *name* must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, *default* is returned if provided, otherwise `AttributeError` is raised. *name* need not be a Python identifier (see `setattr()`).

Nota: Dado que private name mangling ocurre en el momento de la compilación, uno debe enredar manualmente el nombre de un atributo privado (atributos con dos guiones bajos principales) para recuperarlo con `getattr()`.

---

globals()
Return the dictionary implementing the current module namespace. For code within functions, this is set when the function is defined and remains the same regardless of where the function is called.

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 aparece una barra(/) 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 signaturas 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()*.

Detalles de implementación de CPython: This is the address of the object in memory.

Lanza un *evento de auditoría* `builtins.id` con el argumento `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`.

Raises an *auditing event* `builtins.input/result` with the result after successfully reading input.

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.

If *x* is not a number or if *base* is given, then *x* must be a string, *bytes*, or *bytearray* instance representing an integer in radix *base*. Optionally, the string can be preceded by + or − (with no space in between), have leading zeros, be surrounded by whitespace, and have single underscores interspersed between digits.

A base-n integer string contains digits, each representing a value from 0 to n-1. The values 0–9 can be represented by any Unicode decimal digit. The values 10–35 can be represented by `a` to `z` (or `A` to `Z`). The default *base* is 10. The allowed bases are 0 and 2–36. Base-2, -8, and -16 strings can be optionally prefixed with `0b`/`0B`, `0o`/`0O`, or `0x`/`0X`, as with integer literals in code. For base 0, the string is interpreted in a similar way to an integer literal in code, in that the actual base is 2, 8, 10, or 16 as determined by the prefix. Base 0 also disallows leading zeros: `int('010', 0)` is not legal, while `int('010')` and `int('010', 8)` are.

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 3.10.7: *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) o *Tipo de conversión* de múltiples tipos , 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.

Distinto en la versión 3.10: *classinfo* puede ser una *Tipo de conversión*.

issubclass (*class*, *classinfo*)
Return `True` if *class* is a subclass (direct, indirect, or *virtual*) of *classinfo*. A class is considered a subclass of itself. *classinfo* may be a tuple of class objects (or recursively, other such tuples) or a *Tipo de conversión*, in which case return `True` if *class* is a subclass of any entry in *classinfo*. In any other case, a *TypeError* exception is raised.

Distinto en la versión 3.10: *classinfo* puede ser una *Tipo de conversión*.

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

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''):
        process_block(block)
```

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*).

Detalles de implementación de CPython: `len` aumenta *OverflowError* en longitudes mayores que *sys.maxsize*, como *range(2 ** 100)*.

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

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*)
Abre *file* y retorna un *file object* correspondiente. Si el archivo no se puede abrir, se lanza un `OSError`. Consulte tut-files para obtener más ejemplos de cómo utilizar esta función.

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 de caracteres 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 `'w+b'` abren y truncan el fichero. Los modos `'r+'` y `'r+b'` 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 descodificación. En modo de texto (por defecto, o cuando se incluye `'t'` en el argumento *mode*), los contenidos del fichero se retornan como `str`, tras descodificar los *bytes* usando una codificación dependiente de plataforma o usando el *encoding* especificado como argumento.

Hay un caracter adicional permitido para indicar un modo, `'U'`, que ya no tiene ningún efecto y que se considera obsoleto. Anteriormente permitía 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 is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer. Note that specifying a buffer size this way applies for binary buffered I/O, but `TextIOWrapper` (i.e., files opened with `mode='r+'`) would have another buffering. To disable buffering in `TextIOWrapper`, consider using the `write_through` flag for `io.TextIOWrapper.reconfigure()`. When no *buffering* argument is given, the default buffering policy works as follows:

- 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 descodificación – 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'` representará cualquier bytes incorrectos como unidades de código sustituto bajo que van desde U+DC80 a U+DCFF. Estas unidades de código sustituto volverán a convertirse en los mismos bytes cuando el gestor de errores `surrogateescape` sea usado al escribir datos. Esto es útil para el procesado 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 determines how to parse newline characters from the stream. It can be `None`, `''`, `'\n'`, `'\r'`, and `'\r\n'`. It works as follows:

- 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 `''`, 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 valoras 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 `''` o `'\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 del fichero 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, removed in version 3.10: 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`.

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For *int* operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `pow(10, 2)` returns `100`, but `pow(10, -2)` returns `0.01`. For a negative base of type *int* or *float* and a non-integral exponent, a complex result is delivered. For example, `pow(-9, 0.5)` returns a value close to `3j`.

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)`, dónde *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=None*, *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(...)`.

Output buffering is usually determined by *file*. However, if *flush* is true, the stream is forcibly flushed.

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 *dosctring* 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

    @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*)
Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`; otherwise, the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()` method. If `sys.displayhook()` is not accessible, this function will raise `RuntimeError`.

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 que admiten `round()`, los valores se redondean al múltiplo más cercano de 10 a la potencia menos *ndigits*; si dos múltiplos están igualmente cerca, el redondeo se realiza hacia la opción par (así, por ejemplo, tanto `round(0.5)` como `round(-0.5)` son `0`, y `round(1.5)` es `2`). Cualquier valor entero es válido para *ndigits* (positivo, cero o negativo). El valor retornado es un entero si se omite *ndigits* o `None`. De lo contrario, el valor retornado tiene el mismo tipo que *number*.

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 de caracteres, y un valor arbitrario. La cadena de caracteres 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`.

name need not be a Python identifier as defined in identifiers unless the object chooses to enforce that, for example in a custom `__getattribute__()` or via `__slots__`. An attribute whose name is not an identifier will not be accessible using the dot notation, but is accessible through `getattr()` etc..

Nota: Dado que private name mangling ocurre en el momento de la compilación, uno debe enredar manualmente el nombre de un atributo privado (atributos con dos guiones bajos principales) para recuperarlo con

getattr().

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 *NumPy* 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).

El algoritmo de ordenación utiliza sólo comparaciones de < entre items. Mientras que al definir un método `__lt__()` será suficiente para ordenar, **PEP 8** recomienda que las seis comparaciones rich comparisons se implementen. Esto ayudará a evitar errores al usar los mismos datos con otras herramientas de ordenado como *max()* que se basan en un método subyacente diferente. La implementación de las seis comparaciones también ayuda a evitar confusiones por comparaciones de tipos mixtos que pueden llamar reflejado al método `__gt__()`.

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()*‘). Además, puede ser llamado como una función regular (como `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:

```
def regular_function():
    ...

class C:
    method = staticmethod(regular_function)
```

Para más información sobre métodos estáticos, ver types.

Distinto en la versión 3.10: Los métodos estáticos ahora heredan los atributos del método (`__module__`, `__name__`, `__qualname__`, `__doc__` y `__annotations__`), tienen un nuevo atributo `__wrapped__` y ahora son invocables como funciones regulares.

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 iterabais, considera usar *itertools.chain()*.

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

class 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.

El segundo caso de uso es el soporte de herencia múltiple cooperativa en un entorno de ejecución dinámica. Este caso de uso es único en Python y no se encuentra en lenguajes estáticos compilados o que solo soportan herencia única. Esto hace posible implementar los «diagramas de diamantes», donde múltiples clases base implementan el mismo método. Las buenas prácticas de diseño dictan que estas implementaciones tengan la misma signatura de llamada en cada caso (porque el orden de llamadas se determina en tiempo de ejecución, porque ese orden se adapta a los cambios en la jerarquía de clase y porque ese orden puede incluir clases hermanas que son desconocidas antes de la ejecución).

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 punteados tales como `super().__getitem__(name)`. Lo hace implementando su propio método `__getattribute__()` 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 diseñas 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*, ***kwds*)
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.

Con tres argumentos, retorna un nuevo tipo objeto. Esta es esencialmente una forma dinámica de la declaración `class`. La cadena de caracteres *name* es el nombre de la clase y se convierte en el atributo *__name__*. La tupla *bases* contiene las clases base y se convierte en el atributo *__bases__*; si está vacío, se agrega *object*, la base última de todas las clases. El diccionario *dict* contiene definiciones de métodos y atributos para el cuerpo de la clase; se puede copiar o ajustar antes de convertirse en el atributo *__dict__*. Las siguientes dos declaraciones crean objetos idénticos *type*:

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

Ver también *Objetos Tipo*.

Los argumentos de palabras clave proporcionados al formulario de tres argumentos se pasan a la maquinaria de metaclase apropiada (generalmente `__init_subclass__()`) de la misma manera que lo harían las palabras clave en una definición de clase (además de *metaclase*).

Ver también 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.

Una excepción *TypeError* se genera si se especifica un objeto pero no tiene un atributo *__dict__* (por ejemplo, si su clase define el atributo `__slots__`).

zip(**iterables*, *strict=False*)

Iterar sobre varios iterables en paralelo, generando tuplas con un item para cada una.

Ejemplo:

```
>>> for item in zip([1, 2, 3], ['sugar', 'spice', 'everything nice']):
...     print(item)
...
(1, 'sugar')
(2, 'spice')
(3, 'everything nice')
```

Más formalmente: *zip()* retorna un iterador de tuplas, donde la tupla *i*-ésima contiene el elemento *i*-ésimo de cada uno de los argumentos iterables.

Otra forma de pensar en *zip()* es que convierte las filas en columnas y las columnas en filas. Esto es similar a transponer una matriz.

zip() es perezoso: los elementos no se procesarán hasta que el iterable sea iterado, por ejemplo, mediante un bucle `for` o envolviendo en un *list*.

Una cosa a considerar es que los iterables pasados a *zip()* podrían tener diferentes longitudes; a veces por diseño, y a veces por un error en el código que preparó estos iterables. Python ofrece tres enfoques diferentes para tratar este problema:

- De forma predeterminada, *zip()* se detiene cuando se agota el iterable más corto. Ignorará los elementos restantes en las iteraciones más largas, cortando el resultado a la longitud del iterable más corto:

```
>>> list(zip(range(3), ['fee', 'fi', 'fo', 'fum']))
[(0, 'fee'), (1, 'fi'), (2, 'fo')]
```

- *zip()* se usa a menudo en los casos en que se supone que los iterables son de igual longitud. En tales casos, se recomienda usar la opción `strict=True`. Su salida es la misma que la normal *zip()*:

```
>>> list(zip(('a', 'b', 'c'), (1, 2, 3), strict=True))
[('a', 1), ('b', 2), ('c', 3)]
```

  Unlike the default behavior, it raises a *ValueError* if one iterable is exhausted before the others:

```
>>> for item in zip(range(3), ['fee', 'fi', 'fo', 'fum'], strict=True):
...     print(item)
...
(0, 'fee')
(1, 'fi')
(2, 'fo')
Traceback (most recent call last):
  ...
ValueError: zip() argument 2 is longer than argument 1
```

  Sin el argumento `strict=True`, cualquier error que resulte en iterables de diferentes longitudes será silenciado, posiblemente manifestándose como un error difícil de encontrar en otra parte del programa.

- Las iteraciones más cortas se pueden acolchar con un valor constante para que todas las iteraciones tengan la misma longitud. Esto lo hace *itertools.zip_longest()*.

Casos extremos: con un único argumento iterable, *zip()* retorna un iterador de 1-tuplas. Sin argumentos, retorna un iterador vacío.

Consejos y trucos:

- 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, strict=True)`. 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()* en conjunción con el operador `*` puede usar para descomprimir (*unzip*) una lista:

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

Distinto en la versión 3.10: Añadido el argumento `strict`.

__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).

Distinto en la versión 3.9: Cuando se utilizan las opciones de línea de comando `-E` o `-I`, la variable de entorno `PYTHONCASEOK` ahora se ignora.

Notas al pie

CAPÍTULO 3

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
: Un objeto frecuentemente usado para representar la ausencia de un valor, tal como cuando no se entregan argumentos por defecto a una función. Las asignaciones a `None` son ilegales y lanzan un `SytaxError`. `None` es la única instancia del tipo `NoneType`.

NotImplemented
: Valor especial que debe ser retornado por los métodos especiales binarios (por ejemplo `__eq__()`, `__lt__()`, `__add__()`, `__rsub__()`, etc.) para indicar que la operación no está implementada con respecto al otro tipo; puede ser retornado por los métodos especiales binarios in situ (por ejemplo `__imul__()`, `__iand__()`, etc.) con el mismo propósito. No debe evaluarse en un contexto booleano. `NotImplemented` es la única instancia del tipo *types.NotImplementedType*.

Nota: Cuando un método binario (o in situ) retorna `NotImplemented`, el intérprete intentará la operación reflejada en el otro tipo (o algún otro recurso alternativo, según el operador). Si todos los intentos retornan `NotImplemented`, el intérprete lanzará una excepción apropiada. Retornar incorrectamente `NotImplemented` dará como resultado un mensaje de error engañoso o el valor de `NotImplemented` se retornará al código Python.

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.

Distinto en la versión 3.9: La evaluación de `NotImplemented` en un contexto booleano está en desuso. Si bien actualmente se evalúa como verdadero, emitirá un *DeprecationWarning*. Lanzará un *TypeError* en una versión futura de Python.

Ellipsis
: Lo mismo que la elipsis literal «`...`». Valor especial que se utiliza principalmente junto con la sintaxis de seg-

mentación extendida para tipos de datos de contenedor definidos por el usuario."Ellipsis" es la única instancia del tipo *types.EllipsisType*.

__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`

El módulo *site* (que se importa automáticamente durante el inicio, excepto si se proporciona la opción `-S` en la línea de comandos) agrega varias constantes al espacio de nombres integrado. Son útiles para el intérprete interactivo y no deben usarse en programas.

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
Objetos que al ser impresos o llamados imprimen el texto de derechos de autor o créditos, respectivamente.

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).

CAPÍTULO 4

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
`x or y`	if *x* is true, then *x*, else *y*	(1)
`x and y`	si *x* es falso, entonces *x*, si no, *y*	(2)
`not x`	si *x* es falso, entonces `True`, si no, `False`	(3)

Notas:

(1) Este operador usar lógica cortocircuitada, por lo que solo evalúa el segundo argumentos si el primero es falso.

(2) Este operador usa lógica cortocircuitada, por lo que solo evalúa el segundo argumentos 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 ``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 resumen las operaciones de comparación:

Operación	Significado
`<`	estrictamente menor que
`<=`	menor o igual que
`>`	estrictamente mayor que
`>=`	mayor o igual que
`==`	igual que
`!=`	diferente que
`is`	igualdad a nivel de identidad (Son el mismo objeto)
`is not`	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 lanzará una excepción de tipo *TypeError*.

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

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines enough of the methods `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` (in general, `__lt__()` and `__eq__()` are sufficient, if you want the conventional meanings of the comparison operators).

El comportamiento de los operadores `is` e `is not` no se puede personalizar; además, nunca lanzarán 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

There are three distinct numeric types: *integers*, *floating point numbers*, and *complex numbers*. In addition, Booleans are a subtype of integers. Integers have unlimited precision. Floating point numbers are usually implemented using `double` in C; information about the precision and internal representation of floating point numbers for the machine on which your program is running is available in *sys.float_info*. Complex numbers have a real and imaginary part, which are each a floating point number. To extract these parts from a complex number *z*, use `z.real` and `z.imag`. (The standard library includes the additional numeric types *fractions.Fraction*, for rationals, and *decimal.Decimal*, for floating-point numbers with user-definable precision.)

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 amplia 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	Notas	Documentación completa
`x + y`	suma de *x* e *y*		
`x - y`	resta de *x* e *y*		
`x * y`	multiplicación de *x* por *y*		
`x / y`	división de *x* por *y*		
`x // y`	división entera de *x* por *y*	(1)	
`x % y`	resto o residuo de *x* por *y*	(2)	
`-x`	valor de *x*, negado		
`+x`	valor de *x*, sin cambiar		
`abs(x)`	valor absoluto de la magnitud de *x*		*abs()*
`int(x)`	valor de *x* convertido a entero	(3)(6)	*int()*
`float(x)`	valor de *x* convertido a número de punto flotante	(4)(6)	*float()*
`complex(re, im)`	un número complejo, con parte real *re* y parte imaginaria *im*. El valor de *im* por defecto vale cero.	(6)	*complex()*
`c.conjugate()`	conjugado del número complejo *c*		
`divmod(x, y)`	el par de valores `(x // y, x % y)`	(2)	*divmod()*
`pow(x, y)`	*x* elevado a *y*	(5)	*pow()*
`x ** y`	*x* elevado a *y*	(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) Conversion from *float* to *int* truncates, discarding the fractional part. See functions *math.floor()* and *math.ceil()* for alternative conversions.

(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 https://www.unicode.org/Public/13.0.0/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
math.trunc(x)	*x* truncado a *Integral*
round(x[, n])	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.
math.floor(x)	el mayor número *Integral* que sea <= *x*
math.ceil(x)	el menor número *Integral* que sea >= *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
`x | y`	la operación *or* entre *x* e *y*	(4)
`x ^ y`	la operación *exclusive or* entre *x* e *y*	(4)
`x & y`	la operación *and* entre *x* e *y*	(4)
`x << n`	El valor *x* desplazado a la izquierda *n* bits	(1)(2)
`x >> n`	valor de *x* desplazado a la derecha *n* bits	(1)(3)
`~x`	invierte los bits de *x*	

Notas:

(1) Los desplazamientos negativos son ilegales y lanzarán una excepción de tipo `ValeError`.

(2) Un desplazamiento de *n* bits a la izquierda es equivalente a multiplicar por `pow(2, n)`.

(3) Un desplazamiento de *n* bits a la derecha es equivalente a efectuar la división de parte entera (floor) por `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) <= abs(x) < 2**k`. De igual manera, cuando `abs(x)` es lo suficientemente pequeño para tener un logaritmo redondeado correctamente, entonces `k = 1 + int(log(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.`**`bit_count`**`()`
Retorna el número de unos en la representación binaria del valor absoluto del entero. Esto también se conoce como el recuento de la población. Ejemplo:

```
>>> n = 19
>>> bin(n)
'0b10011'
>>> n.bit_count()
3
>>> (-n).bit_count()
3
```

Equivale a:

```
def bit_count(self):
    return bin(self).count("1")
```

Nuevo en la versión 3.10.

`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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\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 lanzará 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 lanzará 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 numero 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` *clase 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 lanza una excepción de tipo `OverflowError` y si se llama con *NaN* (*Not A Number*) lanza una excepción de tipo `ValueError`.

float.**is_integer**()
Retorna `True` si el valor en coma flotante se puede representar sin perdida 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 Calculo del *hash* de tipos numéricos

For numbers `x` and `y`, possibly of different types, it's a requirement that `hash(x) == hash(y)` whenever `x == y` (see the `__hash__()` method documentation for more details). For ease of implementation and efficiency across a variety of numeric types (including *int*, *float*, *decimal.Decimal* and *fractions.Fraction*) Python's hash for numeric types is based on a single mathematical function that's defined for any rational number, and hence applies to all instances of *int* and *fractions.Fraction*, and all finite instances of *float* and *decimal.Decimal*. Essentially, this function is given by reduction modulo `P` for a fixed prime `P`. The value of `P` is made available to Python as the `modulus` attribute of *sys.hash_info*.

Detalles de implementación de CPython: 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` modulo `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` se usan como valores hash para infinito positivo o infinito negativo (respectivamente).
- 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 obtenido 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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
    """Compute the hash of a float x."""

    if math.isnan(x):
        return object.__hash__(x)
    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.

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

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

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

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

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

Python 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__()* lanza 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 procesado 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 ABC *`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 numéricos[3]

Operación	Resultado	Notas
`x in s`	`True` si un elemento de *s* es igual a *x*, `False` en caso contrario	(1)
`x not in s`	`False` si un elemento de *s* es igual a *x*, `True` en caso contrario	(1)
`s + t`	la concatenación de *s* y *t*	(6)(7)
`s * n` o `n * s`	equivale a concatenar *s* consigo mismo *n* veces	(2)(7)
`s[i]`	El elemento *i-esimo* de *s*, empezando a contar en 0	(3)
`s[i:j]`	la rebanada de *s* desde *i* hasta *j*	(3)(4)
`s[i:j:k]`	la rebanada de *s* desde *i* hasta *j*, con paso *j*	(3)(5)
`len(s)`	longitud de *s*	
`min(s)`	el elemento más pequeño de *s*	
`max(s)`	el elemento más grande de *s*	
`s.index(x[, i[, j]])`	índice de la primera ocurrencia de *x* en *s* (en la posición *i* o superior, y antes de *j*)	(8)
`s.count(x)`	número total de ocurrencias de *x* en *s*	

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 si, y las dos secuencias deben ser del mismo tipo y de la misma longitud (Para más detalles, véase comparisons en la referencia del lenguaje).

Forward and reversed iterators over mutable sequences access values using an index. That index will continue to march forward (or backward) even if the underlying sequence is mutated. The iterator terminates only when an *`IndexError`* or a *`StopIteration`* is encountered (or when the index drops below zero).

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:

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

```
>>> "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 si, se puede hacer:

```
>>> 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 `len(s) + i` o `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 <= k < j`. Si *i* o *j* es mayor que `len(s)` se usa `len(s)`. Si *i* se omite o es `None`, se usa `0`. Si *j* se omite o es `None`, se usa `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 <= 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 `len(s)`, si fueran mayores. Si *k* es negativo, *i* y *j* se reducen de `len(s) - 1`. Si *i* o *j* se omiten o su valor es `None`, se convierten es 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` lanza 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 esta 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 todas 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 <= x <= 255`).

Operación	Resultado	Notas
`s[i] = x`	el elemento *i* de *s* es reemplazado por *x*	
`s[i:j] = t`	la rebanada de valores de *s* que van de *i* a *j* es reemplazada por el contenido del iterador *t*	
`del s[i:j]`	equivalente a `s[i:j] = []`	
`s[i:j:k] = t`	los elementos de `s[i:j:k]` son reemplazados por los elementos de *t*	(1)
`del s[i:j:k]`	borra los elementos de `s[i:j:k]` de la lista	
`s.append(x)`	añade *x* al final de la secuencia (Equivale a `s[len(s):len(s)] = [x]`)	
`s.clear()`	elimina todos los elementos de *s* (Equivale a `del s[:]`)	(5)
`s.copy()`	crea una copia superficial de *s* (Equivale a `s[:]`)	(5)
`s.extend(t)` o `s += t`	extiende *s* con los contenidos de *t* (En la mayoría de los casos equivale a `s[len(s):len(s)] = t`)	
`s *= n`	actualiza *s* con su contenido repetido *n* veces	(6)
`s.insert(i, x)`	inserta *x* en *s* en la posición indicada por el índice *i* (Equivale a `s[i:i] = [x]`)	
`s.pop()` o `s.pop(i)`	retorna el elemento en la posición indicada por *i*, y a la vez lo elimina de la secuencia *s*	(2)
`s.remove(x)`	elimina el primer elemento de *s* tal que `s[i]` sea igual a *x*	(3)
`s.reverse()`	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()` lanza 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 si 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 unicamente 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 explicita una nueva secuencia ordenada).

El método *sort()* es estable. Una 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.

Detalles de implementación de CPython: 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 lanza 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 si 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*])

The arguments to the range constructor must be integers (either built-in *int* or any object that implements the `__index__()` special method). If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. If *step* is zero, *ValueError* is raised.

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 lanzar 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))
[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:

- The linspace recipe shows how to implement a lazy version of range suitable for floating point applications.

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"'`
- Double quotes: `"allows embedded 'single' quotes"`
- Triples comillas: ya sea con comillas simples `'''Triples comillas simples'''` o dobles `"""Triples comillas dobles"""`

Las cadenas definidas con comillas tripes 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.

If neither *encoding* nor *errors* is given, `str(object)` returns `type(object).__str__(object)`, which is the «informal» or nicely printable string representation of *object*. For string objects, this is the string itself. If *object* does not have a `__str__()` method, then `str()` falls back to returning `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 ves 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.

If *sub* is empty, returns the number of empty strings between characters which is the length of the string plus one.

`str.`**`encode`**`(`*encoding='utf-8'*, *errors='strict'*`)`
Return the string encoded to `bytes`.

encoding defaults to `'utf-8'`; see *Codificaciones estándar* for possible values.

errors controls how encoding errors are handled. If `'strict'` (the default), a `UnicodeError` exception is raised. Other possible values are `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` and any other name registered via `codecs.register_error()`. See *Manejadores de errores* for details.

For performance reasons, the value of *errors* is not checked for validity unless an encoding error actually occurs, *Modo de desarrollo de Python* is enabled or a debug build is used.

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

Distinto en la versión 3.9: The value of the *errors* argument is now checked in *Modo de desarrollo de Python* and in debug mode.

`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 en 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 "indices 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)`), las 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
...
>>> '{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 lanza 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 identificar 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 minúsculas[4] están en minúsculas y hay, al menos, un carácter de ese tipo. En caso contrario, retorna `False`.

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 minúsculas[4] están en mayúsculas y hay, al menos, un carácter de ese tipo. En caso contrario, retorna `False`.

```
>>> 'BANANA'.isupper()
True
>>> 'banana'.isupper()
False
>>> 'baNana'.isupper()
False
>>> ' '.isupper()
False
```

str.**join**(*iterable*)
Retorna una cadena de caracteres formada por la concatenación de las cadenas en el *iterable*. Se lanza 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

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

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 minúsculas[4].

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 encuentren 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'
```

Véase *`str.removeprefix()`* para un método que removerá una única cadena de prefijo en lugar de todas las ocurrencias dentro de un set de caracteres. Por ejemplo:

```
>>> 'Arthur: three!'.lstrip('Arthur: ')
'ee!'
>>> 'Arthur: three!'.removeprefix('Arthur: ')
'three!'
```

`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.`**`removeprefix`**`(`*prefix*, `/``)`
Si la cadena de caracteres empieza con la cadena *prefix*, retorna `string[len(prefix):]`. De otra manera, retorna una copia de la cadena original:

```
>>> 'TestHook'.removeprefix('Test')
'Hook'
>>> 'BaseTestCase'.removeprefix('Test')
'BaseTestCase'
```

Nuevo en la versión 3.9.

`str.`**`removesuffix`**`(`*suffix*, `/``)`
Si la cadena de caracteres termina con la cadena *suffix*, retorna `string[:-len(suffix)]`. De otra manera, retorna una copia de la cadena original:

```
>>> 'MiscTests'.removesuffix('Tests')
'Misc'
>>> 'TmpDirMixin'.removesuffix('Tests')
'TmpDirMixin'
```

Nuevo en la versión 3.9.

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 lanza 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 encuentren 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'
```

Véase *str.removesuffix()* para un método que removerá una única cadena de sufijo en lugar de de todas las ocurrencias dentro de un set de caracteres. Por ejemplo:

```
>>> 'Monty Python'.rstrip(' Python')
'M'
>>> 'Monty Python'.removesuffix(' Python')
'Monty'
```

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=False*)

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
`\n`	Salto de línea
`\r`	Retorno de carro
`\r\n`	Retorno de carro + salto de línea
`\v` o `\x0b`	Tabulación de línea
`\f` o `\x0c`	Avance de página
`\x1c`	Separador de archivo
`\x1d`	Separador de grupo
`\x1e`	Separador de registro
`\x85`	Siguiente línea (Código de control *C1*)
`\u2028`	Separador de línea
`\u2029`	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 apóstrofos 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"
```

The *`string.capwords()`* function does not have this problem, as it splits words on spaces only.

Alternatively, a workaround for apostrophes can be constructed using regular expressions:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0).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 lanzar 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/minúsculas[4] 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 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:

Conversión	Significado	Notas
`'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 *repr()*).	(5)
`'s'`	Cadena de texto (Representará cualquier objeto usando la función *str()*).	(5)
`'a'`	Cadena de texto (Representará cualquier objeto usando la función *ascii()*).	(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 seria 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'`
- Double quotes: `b"still allows embedded 'single' quotes"`
- 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 <= x < 256` (Si se intenta violar esta restricción se lanzará 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 procesado 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 usase 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'
```

If you want to make the hex string easier to read, you can specify a single character separator *sep* parameter to include in the output. By default, this separator will be included between each byte. A second optional *bytes_per_sep* parameter controls the spacing. Positive values calculate the separator position from the right, negative values from the left.

```
>>> value = b'\xf0\xf1\xf2'
>>> value.hex('-')
'f0-f1-f2'
>>> value.hex('_', 2)
'f0_f1f2'
>>> b'UUDDLRLRAB'.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)`.

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 usase 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.

If *sub* is empty, returns the number of empty slices between characters which is the length of the bytes object plus one.

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

`bytes.`**`removeprefix`**`(`*prefix*`, /)`
`bytearray.`**`removeprefix`**`(`*prefix*`, /)`

Si los datos binarios comienzan con la cadena *prefix*, retorna `bytes[len(prefix):]`. De otra manera, retorna una copia de los datos binarios originales:

```
>>> b'TestHook'.removeprefix(b'Test')
b'Hook'
>>> b'BaseTestCase'.removeprefix(b'Test')
b'BaseTestCase'
```

El argumento *prefix* 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.

Nuevo en la versión 3.9.

`bytes.`**`removesuffix`**`(`*suffix*`, /)`
`bytearray.`**`removesuffix`**`(`*suffix*`, /)`

Si los datos binarios terminan con la cadena expresada en *suffix* y el argumento *suffix* no está vacío, retorna `bytes[:-len(suffix)]`. De otra manera, retorna una copia de los datos binarios originales:

```
>>> b'MiscTests'.removesuffix(b'Tests')
b'Misc'
>>> b'TmpDirMixin'.removesuffix(b'Tests')
b'TmpDirMixin'
```

El argumento *suffix* 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.

Nuevo en la versión 3.9.

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

Return the bytes decoded to a `str`.

encoding defaults to `'utf-8'`; see *Codificaciones estándar* for possible values.

errors controls how decoding errors are handled. If `'strict'` (the default), a `UnicodeError` exception is raised. Other possible values are `'ignore'`, `'replace'`, and any other name registered via `codecs.register_error()`. See *Manejadores de errores* for details.

For performance reasons, the value of *errors* is not checked for validity unless a decoding error actually occurs, *Modo de desarrollo de Python* is enabled or a debug build is used.

Nota: Passing the *encoding* argument to `str` allows decoding any *bytes-like object* directly, without needing to make a temporary `bytes` or `bytearray` object.

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

Distinto en la versión 3.9: The value of the *errors* argument is now checked in *Modo de desarrollo de Python* and in debug mode.

`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 lanza 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 lanza 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 rebanado. 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 lanza 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   '
```

(continué en la próxima página)

(proviene de la página anterior)

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

La secuencia binaria que especifica el conjunto bytes a ser eliminados puede ser cualquier *objeto de tipo binario*. Véase `removeprefix()` para un método que removerá una única cadena de prefijo en lugar de de todas las ocurrencias dentro de un set de caracteres. Por ejemplo:

```
>>> b'Arthur: three!'.lstrip(b'Arthur: ')
b'ee!'
>>> b'Arthur: three!'.removeprefix(b'Arthur: ')
b'three!'
```

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 encuentren 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 que especifica el conjunto bytes a ser eliminados puede ser cualquier *objeto de tipo binario*. Véase `removesuffix()` para un método que removerá una única cadena de sufijo en lugar de de todas las ocurrencias dentro de un set de caracteres. Por ejemplo:

```
>>> b'Monty Python'.rstrip(b' Python')
b'M'
>>> b'Monty Python'.removesuffix(b' Python')
b'Monty'
```

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 en 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 secuencio 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 binaría, 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\n\nde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\n\nde 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 apóstrofos 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 apóstrofos 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 rellenada 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 formateo tipo `printf` con bytes

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). 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 *formateo* 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:

Conversión	Significado	Notas
`'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 `__bytes__()`).	(5)
`'s'`	`'s'` es un alias de `'b'` y solo debe ser usado para bases de código Python2/3.	(6)
`'a'`	Bytes (converts any Python object using `repr(obj).encode('ascii', 'backslashreplace')`).	(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 seria 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 - Añadiendo % formatea a 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(*object*)

Crea un *memoryview* que referencia *object*. La variable *object* 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 *object*. 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>
>>> 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')
```

One-dimensional memoryviews of *hashable* (read-only) types with formats "B", "b" or "c" are also hashable. The hash is defined as `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: One-dimensional memoryviews can now be sliced. One-dimensional memoryviews with formats "B", "b" or "c" are now *hashable*.

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 lanzará 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):
  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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
  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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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 `ndom = 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.

Los conjuntos (*sets*) se pueden construir de diferentes formas:

- Usando una lista de elementos separados por coma entre corchetes: `{'jack', 'sjoerd'}`
- Usando un *set comprehention*: `{c for c in 'abracadabra' if c not in 'abc'}`
- Usando un constructor de tipo: `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 si. 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 en 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*. Lanza 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. Lanza 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`).

A dictionary's keys are *almost* arbitrary values. Values that are not *hashable*, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Values that compare equal (such as `1`, `1.0`, and `True`) can be used interchangeably to index the same dictionary entry.

class dict(***kwargs*)
class dict(*mapping*, ***kwargs*)
class dict(*iterable*, ***kwargs*)
Retorna un diccionario creado a partir de un parámetro opcional por posición, y por una serie de parámetros por nombre, también opcionales.

Los diccionarios se pueden construir de diferentes formas:

- Usando una lista separada por comas de pares `key: value` entre llaves:`{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`

- Usando una comprensión de diccionario: `{}`, `{x: x ** 2 for x in range(10)}`
- Usando un constructor de tipo: `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 crear 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})
>>> f = dict({'one': 1, 'three': 3}, two=2)
>>> a == b == c == d == e == f
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*. Lanza 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 lanza 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 lanza *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*. Lanza 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 lanza 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 lanza 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()* lanza 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
```

d | other
Crea un nuevo diccionario con las claves y valores fusionados de *d* y *other*, por lo cual ambos deben ser diccionarios. Los valores de *other* tienen prioridad cuando *d* y *other* tienen claves compartidas.

Nuevo en la versión 3.9.

d |= other
Actualiza el diccionario *d* con las claves y valores de *other*, el cual podría ser ya sea un a *mapping* o un *iterable* de pares clave/valor. Los valores de *other* tienen prioridad cuando *d* y *other* tienen claves compartidas.

Nuevo en la versión 3.9.

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 "<", "<=", ">=", ">" lanzan 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 lanzar 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.

dictview.mapping
Retorna un *types.MappingProxyType* que envuelve el diccionario original al que se refiere la vista.

Nuevo en la versión 3.10.

Keys views are set-like since their entries are unique and *hashable*. If all values are hashable, so that `(key, value)` pairs are unique and hashable, then the items view is also set-like. (Values views are not treated as set-like since the entries are generally not unique.) For set-like views, all of the operations defined for the abstract base class *collections.abc.Set* are available (for example, ==, <, or ^).

Un ejemplo de uso de una vista de diccionario:

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

(continué en la próxima página)

(proviene de la página anterior)

```
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> 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'}

>>> # get back a read-only proxy for the original dictionary
>>> values.mapping
mappingproxy({'bacon': 1, 'spam': 500})
>>> values.mapping['spam']
500
```

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 continua con la primera sentencia inmediatamente después del cuerpo. En caso contrario la excepción producida continua 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 lanzarse 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.

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

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 Tipos de Anotaciones de Tipo — Generic Alias, Union

Los tipos principales integrados para *type annotations* son *Generic Alias* y *Union*.

4.12.1 Tipo Alias Genérico

`GenericAlias` objects are generally created by subscripting a class. They are most often used with container classes, such as *list* or *dict*. For example, `list[int]` is a `GenericAlias` object created by subscripting the `list` class with the argument *int*. `GenericAlias` objects are intended primarily for use with *type annotations*.

Nota: It is generally only possible to subscript a class if the class implements the special method `__class_getitem__()`.

A `GenericAlias` object acts as a proxy for a *generic type*, implementing *parameterized generics*.

For a container class, the argument(s) supplied to a subscription of the class may indicate the type(s) of the elements an object contains. For example, `set[bytes]` can be used in type annotations to signify a *set* in which all the elements are of type *bytes*.

For a class which defines `__class_getitem__()` but is not a container, the argument(s) supplied to a subscription of the class will often indicate the return type(s) of one or more methods defined on an object. For example, *regular expressions* can be used on both the *str* data type and the *bytes* data type:

- If `x = re.search('foo', 'foo')`, `x` will be a *re.Match* object where the return values of `x.group(0)` and `x[0]` will both be of type *str*. We can represent this kind of object in type annotations with the `GenericAlias re.Match[str]`.
- If `y = re.search(b'bar', b'bar')`, (note the `b` for *bytes*), `y` will also be an instance of `re.Match`, but the return values of `y.group(0)` and `y[0]` will both be of type *bytes*. In type annotations, we would represent this variety of *re.Match* objects with `re.Match[bytes]`.

`GenericAlias` objects are instances of the class *types.GenericAlias*, which can also be used to create `GenericAlias` objects directly.

T[X, Y, ...]

Creates a GenericAlias representing a type T parameterized by types *X*, *Y*, and more depending on the T used. For example, a function expecting a *list* containing *float* elements:

```
def average(values: list[float]) -> float:
    return sum(values) / len(values)
```

Otro ejemplo para el *mapping* de objetos, usando un *dict*, el cual es un tipo genérico que espera dos parámetros de tipo que representan el tipo de la clave y el tipo del valor. En este ejemplo, la función espera un dict con claves de tipo *str* y valores de tipo *int*:

```
def send_post_request(url: str, body: dict[str, int]) -> None:
    ...
```

Las funciones integradas *isinstance()* y *issubclass()* no aceptan tipos GenericAlias como segundo argumento:

```
>>> isinstance([1, 2], list[str])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

The Python runtime does not enforce *type annotations*. This extends to generic types and their type parameters. When creating a container object from a GenericAlias, the elements in the container are not checked against their type. For example, the following code is discouraged, but will run without errors:

```
>>> t = list[str]
>>> t([1, 2, 3])
[1, 2, 3]
```

Además, los tipos de los parámetros de tipos genéricos se borran durante la creación de objetos:

```
>>> t = list[str]
>>> type(t)
<class 'types.GenericAlias'>

>>> l = t()
>>> type(l)
<class 'list'>
```

Llamando a *repr()* o *str()* en un genérico se muestra el tipo parametrizado:

```
>>> repr(list[int])
'list[int]'

>>> str(list[int])
'list[int]'
```

The __getitem__() method of generic containers will raise an exception to disallow mistakes like dict[str][str]:

```
>>> dict[str][str]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: There are no type variables left in dict[str]
```

However, such expressions are valid when *type variables* are used. The index must have as many elements as there are type variable items in the GenericAlias object's *__args__*.

```
>>> from typing import TypeVar
>>> Y = TypeVar('Y')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> dict[str, Y][int]
dict[str, int]
```

Standard Generic Classes

The following standard library classes support parameterized generics. This list is non-exhaustive.

- *tuple*
- *list*
- *dict*
- *set*
- *frozenset*
- *type*
- *collections.deque*
- *collections.defaultdict*
- *collections.OrderedDict*
- *collections.Counter*
- *collections.ChainMap*
- *collections.abc.Awaitable*
- *collections.abc.Coroutine*
- *collections.abc.AsyncIterable*
- *collections.abc.AsyncIterator*
- *collections.abc.AsyncGenerator*
- *collections.abc.Iterable*
- *collections.abc.Iterator*
- *collections.abc.Generator*
- *collections.abc.Reversible*
- *collections.abc.Container*
- *collections.abc.Collection*
- *collections.abc.Callable*
- *collections.abc.Set*
- *collections.abc.MutableSet*
- *collections.abc.Mapping*
- *collections.abc.MutableMapping*
- *collections.abc.Sequence*
- *collections.abc.MutableSequence*
- *collections.abc.ByteString*
- *collections.abc.MappingView*
- *collections.abc.KeysView*
- *collections.abc.ItemsView*

- *collections.abc.ValuesView*
- *contextlib.AbstractContextManager*
- *contextlib.AbstractAsyncContextManager*
- *dataclasses.Field*
- *functools.cached_property*
- *functools.partialmethod*
- *os.PathLike*
- *queue.LifoQueue*
- *queue.Queue*
- *queue.PriorityQueue*
- *queue.SimpleQueue*
- *re.Pattern*
- *re.Match*
- *shelve.BsdDbShelf*
- *shelve.DbfilenameShelf*
- *shelve.Shelf*
- *types.MappingProxyType*
- *weakref.WeakKeyDictionary*
- *weakref.WeakMethod*
- *weakref.WeakSet*
- *weakref.WeakValueDictionary*

Special Attributes of `GenericAlias` objects

Todos los genéricos parametrizados implementan atributos especiales de solo lectura.

genericalias.**__origin__**
: Este atributo apunta a la clase genérica no parametrizada:

```
>>> list[int].__origin__
<class 'list'>
```

genericalias.**__args__**
: This attribute is a *tuple* (possibly of length 1) of generic types passed to the original `__class_getitem__()` of the generic class:

```
>>> dict[str, list[int]].__args__
(<class 'str'>, list[int])
```

genericalias.**__parameters__**
: Este atributo es una tupla (posiblemente vacía) computada perezosamente con las variables de tipo únicas encontradas en `__args__`:

```
>>> from typing import TypeVar

>>> T = TypeVar('T')
>>> list[T].__parameters__
(~T,)
```

Nota: Un objeto `GenericAlias` con *`typing.ParamSpec`* parámetros puede no tener los `__parameters__` correctos después de la sustitución porque *`typing.ParamSpec`* está destinado principalmente a la verificación de tipos estáticos.

Ver también:

PEP 484 - Type Hints Introducing Python's framework for type annotations.

PEP 585 - Type Hinting Generics In Standard Collections Introducing the ability to natively parameterize standard-library classes, provided they implement the special class method `__class_getitem__()`.

Genéricos*, *user-defined generics* and *`typing.Generic` Documentation on how to implement generic classes that can be parameterized at runtime and understood by static type-checkers.

Nuevo en la versión 3.9.

4.12.2 Tipo de conversión

Un objeto de conversión contiene el valor de la operación `|` (bit a bit o) en varios *type objects*. Estos tipos están destinados principalmente a *type annotations*. La expresión de tipo de conversión permite una sintaxis de sugerencia de tipo más limpia en comparación con *`typing.Union`*.

`X | Y | ...`
Define un objeto de conversión que contiene tipos *X*, *Y*, etc. `X | Y` significa X o Y. Es equivalente a `typing.Union[X, Y]`. Por ejemplo, la siguiente función espera un argumento de tipo *`int`* or *`float`*:

```
def square(number: int | float) -> int | float:
    return number ** 2
```

`union_object == other`
Los objetos de conversión se pueden probar para determinar su igualdad con otros objetos de conversión. Detalles:

- Las conversiones de conversión se aplanan:

```
(int | str) | float == int | str | float
```

- Se eliminan los tipos redundantes:

```
int | str | int == int | str
```

- Al comparar conversiones, se ignora el orden:

```
int | str == str | int
```

- Es compatible con *`typing.Union`*:

```
int | str == typing.Union[int, str]
```

- Los tipos opcionales se pueden escribir como una unión con `None`:

```
str | None == typing.Optional[str]
```

`isinstance(obj, union_object)`

`issubclass(obj, union_object)`
Las llamadas a *`isinstance()`* y *`issubclass()`* también son compatibles con un objeto de conversión:

```
>>> isinstance("", int | str)
True
```

Sin embargo, los objetos de unión que contienen *parameterized generics* no se pueden utilizar:

```
>>> isinstance(1, int | list[int])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: isinstance() argument 2 cannot contain a parameterized generic
```

El tipo expuesto por el usuario para el objeto de conversión puede ser accedido desde *types.UnionType* y usado por chequeos *isinstance()*. Un objeto no puede ser instancia por el tipo:

```
>>> import types
>>> isinstance(int | str, types.UnionType)
True
>>> types.UnionType()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create 'types.UnionType' instances
```

Nota: Se agregó el método `__or__()` para objetos de tipo para admitir la sintaxis `X | Y`. Si una metaclase implementa `__or__()`, la Conversión puede anularlo:

```
>>> class M(type):
...     def __or__(self, other):
...         return "Hello"
...
>>> class C(metaclass=M):
...     pass
...
>>> C | int
'Hello'
>>> int | C
int | __main__.C
```

Ver también:

PEP 604 — PEP propone la sintaxis `X | Y` y tipo Conversión.

Nuevo en la versión 3.10.

4.13 Otros tipos predefinidos

El intérprete soporta otros tipos de objetos variados. La mayoría de ellos solo implementan una o dos operaciones.

4.13.1 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 recomiendo 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'>`.

4.13.2 Clases e instancias de clase

Véase objects y class para más información.

4.13.3 Funciones

Los objetos de tipo función se crean mediante definiciones de función. La única operación posible con un objeto de tipo función en 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.

4.13.4 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 lance 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.

4.13.5 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`.

Al acceder a `__code__` se lanza un *evento de auditoría* `object.__getattr__` con argumentos `obj` y `"__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.

4.13.6 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'>`.

4.13.7 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`.

4.13.8 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 `...`.

4.13.9 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`.

4.13.10 Valores booleanos

Los valores booleanos o lógicos son los dos objetos constantes `False` y `True`. Su 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.

4.13.11 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*).

4.14 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 *metaclase* 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. La lista está en orden de definición. Por ejemplo:

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

4.15 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 3.10.7.

4.15.1 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*.

4.15.2 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 3.10.7.

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.

4.15.3 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

CAPÍTULO 5

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 lanzar 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.

5.1 Exception context

When raising a new exception while another exception is already being handled, the new exception's `__context__` attribute is automatically set to the handled exception. An exception may be handled when an `except` or `finally` clause, or a `with` statement, is used.

This implicit exception context can be supplemented with an explicit cause by using `from` with `raise`:

```
raise new_exc from original_exc
```

La expresión que sigue a `from` debe ser una excepción o `None`. Se establecerá como `__cause__` en la excepción generada. La configuración de `__cause__` también establece implícitamente el atributo `__suppress_context__` en `True`, de modo que el uso de `raise new_exc from None` reemplaza efectivamente la excepción anterior con la nueva para fines de visualización (por ejemplo, conversión de *KeyError* a *AttributeError*), dejando la excepción anterior disponible en `__context__` para introspección durante la depuración.

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.

5.2 Inheriting from built-in exceptions

User code can create subclasses that inherit from an exception type. It's recommended to only subclass one exception type at a time to avoid any possible conflicts between how the bases handle the `args` attribute, as well as due to possible memory layout incompatibilities.

Detalles de implementación de CPython: Most built-in exceptions are implemented in C for efficiency, see: Objects/exceptions.c. Some have custom memory layouts which makes it impossible to create a subclass that inherits from multiple exception types. The memory layout of a type is an implementation detail and might change between Python versions, leading to new conflicts in the future. Therefore, it's recommended to avoid subclassing multiple exception types altogether.

5.3 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 rastreo para la excepción y retorna el objeto de excepción. Se usaba con más frecuencia antes de que las características de encadenamiento de excepciones de **PEP 3134** estuvieran disponibles. El siguiente ejemplo muestra cómo podemos convertir una instancia de `SomeException` en una instancia de `OtherException` mientras se conserva el rastreo. Una vez generado, el marco actual se inserta en el rastreo del `OtherException`, como habría sucedido con el rastreo del `SomeException` original si hubiéramos permitido que se propagara al llamador.

```
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 lanzar directamente por *codecs.lookup()*.

5.4 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*.)

Los atributos `name` y `obj` se pueden configurar utilizando argumentos de solo palabras clave para el constructor. Cuando se establecen, representan el nombre del atributo al que se intentó acceder y el objeto al que se accedió para dicho atributo, respectivamente.

Distinto en la versión 3.10: Se agregaron los atributos `name` y `obj`.

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.

Nota: Catching a *KeyboardInterrupt* requires special consideration. Because it can be raised at unpredictable points, it may, in some circumstances, leave the running program in an inconsistent state. It is generally best to allow *KeyboardInterrupt* to end the program as quickly as possible or avoid raising it entirely. (See *Note on Signal Handlers and Exceptions*.)

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.

El atributo `name` se puede establecer utilizando un argumento de solo palabra clave para el constructor. Cuando se establece, representa el nombre de la variable a la que se intentó acceder.

Distinto en la versión 3.10: Se agregó el atributo `name`.

exception NotImplementedError
Esta excepción se deriva de *RuntimeError*. En las clases base definidas por el usuario, los métodos abstractos deberían lanzar 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 de archivo original pasado a la función, en lugar del nombre codificado o decodificado desde *filesystem encoding and error handler*. Además, se agregaron el argumento y atributo del constructor *filename2*.

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 lanzar *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 lanza 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 lanzar mediante `__anext__()` de un objeto *asynchronous iterator* para detener la iteración.

Nuevo en la versión 3.5.

exception SyntaxError(*message*, *details*)

Se genera cuando el analizador encuentra un error de sintaxis. Esto puede ocurrir en una instrucción `import`, en una llamada a las funciones integradas *compile()*, *exec()* o *eval()*, o al leer el script inicial o la entrada estándar (también de forma interactiva).

El *str()* de la instancia de excepción retorna solo el mensaje de error. Detalles es una tupla cuyos miembros también están disponibles como atributos separados.

filename

El nombre del archivo en el que se produjo el error de sintaxis.

lineno

En qué número de línea del archivo se produjo el error. Tiene un índice 1: la primera línea del archivo tiene un `lineno` de 1.

offset

La columna de la línea donde ocurrió el error. Está indexado a 1: el primer carácter de la línea tiene un `offset` de 1.

text

El texto del código fuente involucrado en el error.

end_lineno

En qué número de línea del archivo termina el error. Está indexado a 1: la primera línea del archivo tiene un `lineno` de 1.

end_offset

Finaliza la columna de la línea final donde ocurrió el error. Está indexado a 1: el primer carácter de la línea tiene un `offset` de 1.

Para errores en los campos de cadena f, el mensaje tiene el prefijo «cadena f:» y las compensaciones son compensaciones en un texto construido a partir de la expresión de reemplazo. Por ejemplo, compilar f'Bad {ab} field "da como resultado este atributo args: (" f-string: … ", (" ", 1, 2," (ab) n ", 1, 5)) .

Distinto en la versión 3.10: Se agregaron los atributos *end_lineno* y *end_offset*.

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 lanzar 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 lanzar.

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 codificación. 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.

5.4.1 Excepciones del sistema operativo

Las siguientes excepciones son subclases de *OSError*, se generan según el código de error del sistema.

exception BlockingIOError
Raised when an operation would block on an object (e.g. socket) set for non-blocking operation. Corresponds to errno *EAGAIN*, *EALREADY*, *EWOULDBLOCK* and *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 modulo *io*.

exception ChildProcessError
Raised when an operation on a child process failed. Corresponds to errno *ECHILD*.

exception ConnectionError
Una clase base para problemas relacionados con la conexión.

Las subclases son *BrokenPipeError*, *ConnectionAbortedError*, *ConnectionRefusedError* y *ConnectionResetError*.

exception BrokenPipeError
A subclass of *ConnectionError*, raised when trying to write on a pipe while the other end has been closed, or trying to write on a socket which has been shutdown for writing. Corresponds to errno *EPIPE* and *ESHUTDOWN*.

exception ConnectionAbortedError
A subclass of *ConnectionError*, raised when a connection attempt is aborted by the peer. Corresponds to errno *ECONNABORTED*.

exception ConnectionRefusedError
A subclass of *ConnectionError*, raised when a connection attempt is refused by the peer. Corresponds to errno *ECONNREFUSED*.

exception ConnectionResetError
A subclass of *ConnectionError*, raised when a connection is reset by the peer. Corresponds to errno *ECONNRESET*.

exception FileExistsError
Raised when trying to create a file or directory which already exists. Corresponds to errno *EEXIST*.

exception FileNotFoundError
Raised when a file or directory is requested but doesn't exist. Corresponds to 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 de señales lanza una excepción (ver **PEP 475** para la justificación), en lugar de lanzar *InterruptedError*.

exception IsADirectoryError
Raised when a file operation (such as *os.remove()*) is requested on a directory. Corresponds to errno *EISDIR*.

exception NotADirectoryError
Raised when a directory operation (such as *os.listdir()*) is requested on something which is not a directory. On most POSIX platforms, it may also be raised if an operation attempts to open or traverse a non-directory file as if it were a directory. Corresponds to errno *ENOTDIR*.

exception PermissionError
Raised when trying to run an operation without the adequate access rights - for example filesystem permissions. Corresponds to `errno` *EACCES* and *EPERM*.

exception ProcessLookupError
Raised when a given process doesn't exist. Corresponds to `errno` *ESRCH*.

exception TimeoutError
Raised when a system function timed out at the system level. Corresponds to `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

5.5 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.

Ignorado por los filtros de advertencia predeterminados, excepto en el módulo `__main__` (**PEP 565**). Habilitando *Modo Desarrollo de Python* muestra esta advertencia.

La política de obsolescencia se describe en **PEP 387**.

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.

Ignorado por los filtros de advertencia predeterminados. Habilitando *Modo Desarrollo de Python* muestra esta advertencia.

La política de obsolescencia se describe en **PEP 387**.

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.

Ignorado por los filtros de advertencia predeterminados. Habilitando *Modo Desarrollo de Python* muestra esta advertencia.

exception UnicodeWarning
Clase base para advertencias relacionadas con Unicode.

exception EncodingWarning
Clase base para advertencias relacionadas con codificaciones.

Consulte *EncodingWarning opcional* para obtener más detalles.

Nuevo en la versión 3.10.

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 predeterminados. Habilitando *Modo Desarrollo de Python* muestra esta advertencia.

Nuevo en la versión 3.2.

5.6 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- EncodingWarning
           +-- ResourceWarning
```

CAPÍTULO 6

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 descriptas `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

El método `str.format()` y la clase `Formatter` comparten la misma sintaxis para las cadenas de caracteres de formato (aunque en el caso de `Formatter` las subclases pueden definir su propia sintaxis de cadena de caracteres de formato). La sintaxis es similar a la de literales de cadena con formato pero hay algunas diferencias.

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 `'{} {}'.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 {}"                   # 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 del 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][typ
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:

Opción	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 "+000000120". Esta opción de alineación solo es válida para tipos numéricos. Se convierte en el valor predeterminado cuando "0" 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:

Opción	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).
espacio	indica que el espacio inicial debe ser usado para números positivos y el signo menos para números negativos.

La opción `'#'` hace que la «forma alternativa» se utilice para la conversión. La forma alternativa se define de forma diferente para diferentes tipos. Esta opción solo es válida para los tipos entero, flotante y complejo. Para los enteros, cuando se utiliza la salida binaria, octal o hexadecimal, esta opción agrega el respectivo prefijo `'0b'`, `'0o'` o `'0x'` al valor de salida. Para los flotantes y complejos, la forma alternativa hace que el resultado de la conversión siempre contenga un carácter de punto decimal, incluso si no hay dígitos que lo sigan. Normalmente, un carácter de punto decimal aparece en el resultado de estas conversiones sólo si un dígito lo sigue. Además, para las conversiones `'g'` y `'G'`, los ceros finales no se eliminan del resultado.

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 `'_'` señaliza 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 formateo. 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 `'='`.

Distinto en la versión 3.10: Anteponer el campo *width* por `'0'` ya no afecta el alineamiento por defecto para cadenas de caracteres.

The *precision* is a decimal integer indicating how many digits should be displayed after the decimal point for presentation types `'f'` and `'F'`, or before and after the decimal point for presentation types `'g'` or `'G'`. For string presentation types the field indicates the maximum field size - in other words, how many characters will be used from the field content. The *precision* is not allowed for integer presentation types.

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:

Tipo	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. En caso de que `'#'` se especifique, el prefijo `'0x'` convertirá en `'0X'` también.
`'n'`	Número. Es lo mismo que `'d'`, excepto que usa la configuración regional actual para insertar el número apropiado de caracteres separadores.
None	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.

Los tipos de presentación disponibles para valores *`float`* y *`Decimal`* son:

Tipo	Significado
`'e'`	Notación científica. Para una precisión dada `p`, formatea el número en notación científica con la letra `e` separando el coeficiente del exponente. El coeficiente tiene un dígito antes, y `p` dígitos después, del punto decimal, para un total de `p + 1` dígitos significativos. Cuando se no da una precisión, usa una precisión de `6` dígitos después del punto decimal para *float*, y muestra todos los dígitos del coeficiente para *Decimal*. Si no hay dígitos después del punto decimal, el punto decimal también es removido a no ser que se use la opción `#`.
`'E'`	Notación científica. Igual que `'e'` excepto que utiliza una "E" mayúscula como carácter separador.
`'f'`	Notación de punto fijo. Para una precisión dada `p`, formatea el número como un valor decimal con exactamente `p` dígitos siguiendo el punto decimal. Cuando no se da una precisión, usa una precisión de `6` dígitos después del punto decimal para *float*, y usa una precisión tan grande como sea necesaria para mostrar todos los dígitos del coeficiente para *Decimal*. Si no hay dígitos después del punto decimal, el punto decimal también es removido a no ser que se use la opción `#`.
`'F'`	Notación de punto fijo. Igual que `'f'`, pero convierte (nulos) `nan` a `NAN` e `inf` a `INF`.
`'g'`	Formato general. Para una precisión dada `p >= 1`, redondea el número a `p` dígitos significativos y luego formatea el resultado como formato de punto fijo o en notación científica, dependiendo de su magnitud. Una precisión de `0` es tratada como equivalente a una precisión de `1`. Las reglas precisas son las siguientes: supongamos que el resultado formateado con el tipo de presentación `'e'` y la precisión `p-1` tiene exponente `exp`. Entonces, si `m <= exp < p`, donde `m` es -4 para *floats* y -6 para *Decimals*, el número se formatea con el tipo de presentación `'f'` y la precisión `p-1-exp`. De lo contrario, el número se formatea con el tipo de presentación `'e'` y precisión `p-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 `'#'`. Si no se da una precisión, usa una precisión de `6` dígitos significativos para *float*. Para *Decimal* el coeficiente del resultado se construye usando los dígitos del coeficiente del valor; se usa notación científica para valores menores a `1e-6` en valor absoluto y valores donde el valor posicional del dígito menos significativo es mayor a 1, de otra forma se usa notación de punto fijo. Infinito positivo y negativo, cero positivo y negativo, y nulos (*nans*) son respectivamente formateados como `inf`, `-inf`, `0`, `-0` y `nan`, 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.
None	Para *float* esto es lo mismo que `'g'`, excepto que cuando se usa notación de punto fijo para formatear el resultado, siempre incluye al menos un dígito pasado el punto decimal. La precisión usada es tan larga como sea necesaria para representar el valor dado fielmente. Para *Decimal*, esto es lo mismo que `'g'` o `'G'` dependiendo del valor de `context.capitals` para el contexto decimal actual. El efecto general es el de igualar la salida de *str()* al ser alterada por los otros modificadores de formato.

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                  '
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> '{:>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'
>>> '{: 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}'
↪{: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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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
   10     A    12  1010
   11     B    13  1011
```

6.1.4 Cadenas de plantillas

Template strings provide simpler string substitutions as described in **PEP 292**. A primary use case for template strings is for internationalization (i18n) since in that context, the simpler syntax and functionality makes it easier to translate than other built-in string formatting facilities in Python. As an example of a library built on template strings for i18n, see the flufl.i18n package.

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={}*, */*, ***kwds*)

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 *kwds* son dados y hay elementos duplicados, los *placeholders* (comodines) de *kwds* tienen prioridad.

safe_substitute(*mapping={}*, */*, ***kwds*)

Igual que *substitute()*, excepto que si faltan comodines de *mapping* y *kwds*, 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'
>>> 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.`**`capwords`**(*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 `'aaaaaaa'`, `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.
- To match a literal `']'` inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both `[()[\]{}]` and `[]()[{}]` will match a right bracket, as well as left bracket, braces, and parentheses.
- 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 *quote* (cita)	Formas de hacer referencia
en el mismo patrón en sí mismo	• `(?P=quote)` (como se muestra) • `\1`
cuando se procesa el objeto de la coincidencia *m*	• `m.group('quote')` • `m.end('quote')` (etc.)
en una cadena pasada al argumento *repl* de `re.sub()`	• `\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 `'elel'` (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 *[^ tnrfv]*.

\w

Para los patrones de Unicode (str): Matches Unicode word characters; this includes alphanumeric characters (as defined by *str.isalnum()*) as well as the underscore (_). If the *ASCII* flag is used, only `[a-zA-Z0-9_]` is matched.

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:

```
\a      \b      \f      \n
\N      \r      \t      \u
\U      \v      \x      \\
```

(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.

Flags

Distinto en la versión 3.6: Ahora las constantes de indicadores son instancias de `RegexFlag`, que es una subclase de *enum.IntFlag*.

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 Ü 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)`.

  Notar que cuando los patrones Unicode `[a-z]` o `[A-Z]` se usan en combinación con el flag `IGNORECASE`\`, coincidirán con las 52 letras ASCII y 4 letras adicionales no ASCII: “İ” (U+0130, letra mayúscula latina I con punto arriba), “ı” (U+0131, letra minúscula latina sin punto i), “ſ” (U+017F, letra minúscula latina s larga) y “K” (U+212A, signo Kelvin). Si se usa el indicador *ASCII*, sólo las letras de la “a” a la “z” y de la “A” a la “Z” coinciden.

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**
This flag allows you to write regular expressions that look nicer and are more readable by allowing you to visually separate logical sections of the pattern and add comments. Whitespace within the pattern is ignored, except when in a character class, or when preceded by an unescaped backslash, or within tokens like `*?`, `(?:` or `(?P<...>`. For example, `(? :` and `* ?` are not allowed. When a line contains a # that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such # through the end of the line are ignored.

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)`.

Functions

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.**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 de *pattern* en *string*, como una lista de strings o tuplas. El *string* se escanea de izquierda a derecha y las coincidencias se retornan en el orden en que se encuentran. Las coincidencias vacías se incluyen en el resultado.

El resultado depende del número de grupos detectados en el patrón. Si no hay grupos, retorna una lista de strings que coincidan con el patrón completo. Si existe exactamente un grupo, retorna una lista de strings que coincidan con ese grupo. Si hay varios grupos presentes, retorna una lista de tuplas de strings que coinciden con los grupos. Los grupos que no son detectados no afectan la forma del resultado.

```
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.findall(r'(\w+)=(\d+)', 'set width=20 and height=10')
[('width', '20'), ('height', '10')]
```

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('https://www.python.org'))
https://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.

Exceptions

`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
↪".
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> 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 `scanf()`	Expresión regular
`%c`	`.`
`%5c`	`.{5}`
`%d`	`[-+]?\d+`
`%e, %E, %f, %g`	`[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?`
`%i`	`[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)`
`%o`	`[-+]?[0-7]+`
`%s`	`\S+`
`%u`	`\d+`
`%x, %X`	`[-+]?(0[xX])?[\dA-Fa-f]+`

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 offers different primitive operations based on regular expressions:

- *`re.match()`* checks for a match only at the beginning of the string
- *`re.search()`* checks for a match anywhere in the string (this is what Perl does by default)
- *`re.fullmatch()`* checks for entire string to be a match

Por ejemplo:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<re.Match object; span=(2, 3), match='c'>
>>> re.fullmatch("p.*n", "python") # Match
<re.Match object; span=(0, 6), match='python'>
>>> re.fullmatch("r.*n", "python") # No match
```

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'],
['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 reoprt yuor asnebces potlmrpy.'
```

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\b", 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\b", 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='\\'>
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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('(?P<%s>%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
        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;
'''
```

(continué en la próxima página)

(proviene de la página anterior)

```
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

Este módulo proporciona clases y funciones para comparar secuencias. Se puede utilizar, por ejemplo, para comparar archivos y puede producir información sobre diferencias de archivo en varios formatos, incluidos HTML y contexto y diferencias unificadas. Para comparar directorios y archivos, consulte también el módulo *filecmp*.

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 lineas 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 compli-cada 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 lineas de texto y producir diferencias o deltas en una forma legible por humanos. Differ usa *SequenceMatcher* tanto para comparar secuencias de lineas, como para comparar secuencias de caracteres entre lineas similares.

Cada linea 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 linea por linea del texto, con cambios interlineales e intralineales. 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 lineas serán divididas y ajustadas al ancho de columna, su valor por defecto es None, donde las lineas 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 *tolines* (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 intralineales 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 lineas 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 lineas 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
! 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* (comúnmente 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 lineas 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 lineas 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 Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in Dr. Dobb's Journal in July, 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 primer 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=0*, *ahi=None*, *blo=0*, *bhi=None*)

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 porporcionó *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)`.

Distinto en la versión 3.9: Se agregaron argumentos predeterminados.

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 quíntuplas (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'`	`a[i1:i2]` debe ser reemplazado por `b[j1:j2]`.
`'delete'`	`a[i1:i2]` debe ser eliminado. Nótese que en este caso `j1 == j2`.
`'insert'`	`b[j1:j2]` debe ser insertado en `a[i1:i1]`. Nótese que en este caso `i1 == i2`.
`'equal'`	`a[i1:i2] == b[j1:j2]` (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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 linea 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 lineas, 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 lineas 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)
    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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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*, ***, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder='[...]'*)

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 de palabras clave opcionales corresponden a los atributos de instancia de *TextWrapper*, que se documentan a continuación.

Ver el método *TextWrapper.wrap()* para más detalles sobre el comportamiento de *wrap()*.

textwrap.**fill**(*text*, *width=70*, ***, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder='[...]'*)

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*, ***, *fix_sentence_endings=False*, *break_long_words=True*, *break_on_hyphens=True*, *placeholder='[...]'*)

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)
'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) Si es verdadero, después de la expansión por tabulador pero antes de envolver, el método *wrap()* reemplazará cada carácter de espacio en blanco con un espacio sencillo. Los caracteres de los espacios en blanco reemplazados son los siguientes: tab, nueva línea, tab vertical, *formfeed* y *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.

Dado que el algoritmo de detección de oraciones se basa en `string.lowercase` para la definición de «letra en minúscula», y en la convención de utilizar dos espacios después de un punto para separar las oraciones en la misma línea, es específico para los textos en inglés.

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
(predeterminado: `True`) Si es verdadero, el ajuste se producirá preferiblemente en espacios en blanco y justo después de los guiones en palabras compuestas, como es habitual en inglés. Si es falso, solo los espacios en blanco se considerarán lugares potencialmente buenos para los saltos de línea, pero debe establecer *break_long_words* en falso si desea palabras verdaderamente insecables. El comportamiento predeterminado en versiones anteriores era permitir siempre romper palabras con guiones.

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 la UCD versión 13.0.0.

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 nombre[1] y secuencias con nombre[2].

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).

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

[1] https://www.unicode.org/Public/13.0.0/ucd/NameAliases.txt
[2] https://www.unicode.org/Public/13.0.0/ucd/NamedSequences.txt

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 modulo *`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.

Detalles de implementación de CPython: 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**()
Obtenga el índice inicial o final del alcance de finalización. Estos índices son los argumentos *start* y *end* pasados a la devolución de llamada `rl_attempted_completion_function` de la biblioteca subyacente. Los valores pueden ser diferentes en el mismo escenario de edición de entrada según la implementación de la línea de lectura de C subyacente. Ejemplo: se sabe que libedit se comporta de manera diferente a libreadline.

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 eli-

mina 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)
    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")):
```

(continué en la próxima página)

(proviene de la página anterior)

```
        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`*.

CAPÍTULO 7

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

This module converts between Python values and C structs represented as Python `bytes` objects. Compact *format strings* describe the intended conversions to/from Python values. The module's functions and objects can be used for two largely distinct applications, data exchange with external sources (files or network connections), or data transfer between the Python application and the C layer.

Nota: When no prefix character is given, native mode is the default. It packs or unpacks data based on the platform and compiler on which the Python interpreter was built. The result of packing a given C struct includes pad bytes which maintain proper alignment for the C types involved; similarly, alignment is taken into account when unpacking. In contrast, when communicating data between external sources, the programmer is responsible for defining byte ordering and padding between elements. See *Orden de Bytes, Tamaño y Alineación* for details.

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*)
Iteratively unpack from the buffer *buffer* according to the format string *format*. This function returns an iterator which will read equally sized chunks from the buffer until all its contents have been consumed. The buffer's size in bytes must be a multiple of the size required by the format, as reflected by `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

Format strings describe the data layout when packing and unpacking data. They are built up from *format characters*, which specify the type of data being packed/unpacked. In addition, special characters control the *byte order, size and alignment*. Each format string consists of an optional prefix character which describes the overall properties of the data and one or more format characters which describe the actual data values and padding.

Orden de Bytes, Tamaño y Alineación

By default, C types are represented in the machine's native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler). This behavior is chosen so that the bytes of a packed struct correspond exactly to the memory layout of the corresponding C struct. Whether to use native byte ordering and padding or standard formats depends on the application.

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:

Carácter	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 `'@'`.

Native byte order is big-endian or little-endian, depending on the host system. For example, Intel x86, AMD64 (x86-64), and Apple M1 are little-endian; IBM z and many legacy architectures are big-endian. Use `sys.byteorder` to check the endianness of your system.

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 *Caracteres de formato*.

Nótese la diferencia entre `'@'` y `'='` : ambos utilizan el orden de bytes nativo, pero el tamaño y la alineación de este último está estandarizado.

La forma `'!'` representa el orden de bytes en la red, el cuál siempre es big-endian tal como se define en 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 conteo repetido de ceros. Véase *Ejemplos*.

Caracteres de formato

Los caracteres de formato tienen el siguiente significado; la conversión entre los valores de 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		(7)
`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		(9)
`p`	`char[]`	bytes		(8)
`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) The `'?'` conversion code corresponds to the `_Bool` type defined by C99. If this type is not available, it is simulated using a `char`. In standard mode, it is always represented by one byte.

(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: Agregado el uso del método `__index__()` para los no enteros.

(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 nativo de bytes (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.

(7) When packing, `'x'` inserts one NUL byte.

(8) 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.

(9) For the `'s'` format character, the count is interpreted as the length of the bytes, not a repeat count like for the other format characters; for example, `'10s'` means a single 10-byte string mapping to or from a single Python byte string, while `'10c'` means 10 separate one byte character elements (e.g., `cccccccccc`) mapping to or from ten different Python byte objects. (See *Ejemplos* for a concrete demonstration of the difference.) If a count is not given, it defaults to 1. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting bytes object always has exactly the specified number of bytes. As a special case, `'0s'` means a single, empty string (while `'0c'` means 0 characters).

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.

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: Anteriormente, algunos de los formatos enteros ajustaban los valores fuera de rango y lanzaban *DeprecationWarning* en vez de *struct.error*.

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: Native byte order examples (designated by the `'@'` format prefix or lack of any prefix character) may not match what the reader's machine produces as that depends on the platform and compiler.

Pack and unpack integers of three different sizes, using big endian ordering:

```
>>> from struct import *
>>> pack(">bhl", 1, 2, 3)
b'\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('>bhl', b'\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('>bhl')
7
```

Attempt to pack an integer which is too large for the defined field:

```
>>> pack(">h", 99999)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
struct.error: 'h' format requires -32768 <= number <= 32767
```

Demonstrate the difference between `'s'` and `'c'` format characters:

```
>>> pack("@ccc", b'1', b'2', b'3')
b'123'
>>> pack("@3s", b'123')
b'123'
```

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)
```

The ordering of format characters may have an impact on size in native mode since padding is implicit. In standard mode, the user is responsible for inserting any desired padding. Note in the first `pack` call below that three NUL bytes were added after the packed `'#'` to align the following integer on a four-byte boundary. In this example, the output was produced on a little endian machine:

```
>>> pack('@ci', b'#', 0x12131415)
b'#\x00\x00\x00\x15\x14\x13\x12'
>>> pack('@ic', 0x12131415, b'#')
b'\x15\x14\x13\x12#'
>>> calcsize('@ci')
8
>>> calcsize('@ic')
5
```

The following format `'llh0l'` results in two pad bytes being added at the end, assuming the platform's longs are aligned on 4-byte boundaries:

```
>>> pack('@llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

Ver también:

Módulo ***array*** Almacenamiento binario empaquetado de datos homogéneos.

Module ***json*** JSON encoder and decoder.

Module ***pickle*** Python object serialization.

7.1.3 Applications

Two main applications for the *struct* module exist, data interchange between Python and C code within an application or another application compiled using the same compiler (*native formats*), and data interchange between applications using agreed upon data layout (*standard formats*). Generally speaking, the format strings constructed for these two domains are distinct.

Native Formats

When constructing format strings which mimic native layouts, the compiler and machine architecture determine byte ordering and padding. In such cases, the `@` format character should be used to specify native byte ordering and data sizes. Internal pad bytes are normally inserted automatically. It is possible that a zero-repeat format code will be needed at the end of a format string to round up to the correct byte boundary for proper alignment of consective chunks of data.

Consider these two simple examples (on a 64-bit, little-endian machine):

```
>>> calcsize('@lhl')
24
>>> calcsize('@llh')
18
```

Data is not padded to an 8-byte boundary at the end of the second format string without the use of extra padding. A zero-repeat format code solves that problem:

```
>>> calcsize('@llh0l')
24
```

The `'x'` format code can be used to specify the repeat, but for native formats it is better to use a zero-repeat format like `'0l'`.

By default, native byte ordering and alignment is used, but it is better to be explicit and use the `'@'` prefix character.

Standard Formats

When exchanging data beyond your process such as networking or storage, be precise. Specify the exact byte order, size, and alignment. Do not assume they match the native order of a particular machine. For example, network byte order is big-endian, while many popular CPUs are little-endian. By defining this explicitly, the user need not care about the specifics of the platform their code is running on. The first character should typically be < or > (or `!`). Padding is the responsibility of the programmer. The zero-repeat format character won't work. Instead, the user must explicitly add `'x'` pad bytes where needed. Revisiting the examples from the previous section, we have:

```
>>> calcsize('<qh6xq')
24
>>> pack('<qh6xq', 1, 2, 3) == pack('@lhl', 1, 2, 3)
True
>>> calcsize('@llh')
```

(continué en la próxima página)

(proviene de la página anterior)

```
18
>>> pack('@llh', 1, 2, 3) == pack('<qqh', 1, 2, 3)
True
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
24
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
True
```

The above results (executed on a 64-bit machine) aren't guaranteed to match when executed on different machines. For example, the examples below were executed on a 32-bit machine:

```
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
12
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
False
```

7.1.4 Clases

El módulo *struct* también define el siguiente tipo:

class struct.**Struct**(*format*)

Return a new Struct object which writes and reads binary data according to the format string *format*. Creating a Struct object once and calling its methods is more efficient than calling module-level functions with the same format since the format string is only compiled once.

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

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry, which manages the codec and error handling lookup process. Most standard codecs are *text encodings*, which encode text to bytes (and decode bytes to text), but there are also codecs provided that encode text to text, and bytes to bytes. Custom codecs may encode and decode between arbitrary types, but some module features are restricted to be used specifically with *text encodings* or with codecs that encode to `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 con guiones y espacios convertidos a guiones bajos, y que retorne un objeto `CodecInfo`. En caso de que una función de búsqueda no pueda encontrar una codificación dada, debería retornar `None`.

Distinto en la versión 3.9: Guiones y espacios se convierten a guiones bajos.

codecs.**unregister**(*search_function*)
Anula el registro de una función de búsqueda de códecs y elimina el caché del registro. Si la función de búsqueda no está registrada, no hace nada.

Nuevo en la versión 3.10.

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: If *encoding* is not `None`, then the underlying encoded files are always opened in binary mode. No automatic conversion of `'\n'` is done on reading and writing. The *mode* argument may be any binary mode acceptable to the built-in *open()* function; the `'b'` is automatically added.

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 códecs también necesitan definir cómo manejará los errores de codificación y decodificación.

Manejadores de errores

To simplify and standardize error handling, codecs may implement different error handling schemes by accepting the *errors* string argument:

```
>>> 'German ß, ♬'.encode(encoding='ascii', errors='backslashreplace')
b'German \\xdf, \\u266c'
>>> 'German ß, ♬'.encode(encoding='ascii', errors='xmlcharrefreplace')
b'German &#223;, &#9836;'
```

The following error handlers can be used with all Python *Codificaciones estándar* codecs:

Valor	Significado
`'strict'`	Raise *UnicodeError* (or a subclass), this is the default. Implemented in *strict_errors()*.
`'ignore'`	Ignore los datos mal formados y continúe sin previo aviso. Implementado en *ignore_errors()*.
`'replace'`	Replace with a replacement marker. On encoding, use `?` (ASCII character). On decoding, use `\ufffd` (U+FFFD, the official REPLACEMENT CHARACTER). Implemented in *replace_errors()*.
`'backslashreplace'`	Replace with backslashed escape sequences. On encoding, use hexadecimal form of Unicode code point with formats `\xhh` `\uxxxx` `\Uxxxxxxxx`. On decoding, use hexadecimal form of byte value with format `\xhh`. Implemented in *backslashreplace_errors()*.
`'surrogateescape'`	En la decodificación, reemplace 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).

The following error handlers are only applicable to encoding (within *text encodings*):

Valor	Significado
`'xmlcharrefreplace'`	Replace with XML/HTML numeric character reference, which is a decimal form of Unicode code point with format `&#num;` Implemented in *xmlcharrefreplace_errors()*.
`'namereplace'`	Replace with `\N{...}` escape sequences, what appears in the braces is the Name property from Unicode Character Database. Implemented in *namereplace_errors()*.

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	Allow encoding and decoding surrogate code point (`U+D800` - `U+DFFF`) as normal code point. Otherwise these codecs treat the presence of surrogate code point in *str* as an error.

Nuevo en la versión 3.1: Los manejadores de errores `'surrogateescape'` y `'surrogatepass'`.

Distinto en la versión 3.4: The `'surrogatepass'` error handler now works with utf-16* and utf-32* codecs.

Nuevo en la versión 3.5: El controlador de errores `'namereplace'`.

Distinto en la versión 3.5: The `'backslashreplace'` error handler now works with decoding and translating.

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 lanzará *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*)
Implements the `'strict'` error handling.

Each encoding or decoding error raises a *UnicodeError*.

`codecs.`**`ignore_errors`**(*exception*)
Implements the `'ignore'` error handling.

Malformed data is ignored; encoding or decoding is continued without further notice.

`codecs.`**`replace_errors`**(*exception*)
Implements the `'replace'` error handling.

Substitutes `?` (ASCII character) for encoding errors or `\ufffd` (U+FFFD, the official REPLACEMENT CHARACTER) for decoding errors.

`codecs.`**`backslashreplace_errors`**(*exception*)
Implements the `'backslashreplace'` error handling.

Malformed data is replaced by a backslashed escape sequence. On encoding, use the hexadecimal form of Unicode code point with formats `\xhh` `\uxxxx` `\Uxxxxxxxx`. On decoding, use the hexadecimal form of byte value with format `\xhh`.

Distinto en la versión 3.5: Works with decoding and translating.

`codecs.`**`xmlcharrefreplace_errors`**(*exception*)
Implements the `'xmlcharrefreplace'` error handling (for encoding within *text encoding* only).

The unencodable character is replaced by an appropriate XML/HTML numeric character reference, which is a decimal form of Unicode code point with format `&#num;` .

`codecs.`**`namereplace_errors`**(*exception*)
Implements the `'namereplace'` error handling (for encoding within *text encoding* only).

The unencodable character is replaced by a `\N{...}` escape sequence. The set of characters that appear in the braces is the Name property from Unicode Character Database. For example, the German lowercase letter `'ß'` will be converted to byte sequence `\N{LATIN SMALL LETTER SHARP S}`.

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='strict'*)

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='strict'*)

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 códecs 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=False*)
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 códecs 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=False*)
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*)
Writes the concatenated iterable of strings to the stream (possibly by reusing the *write()* method). Infinite or very large iterables are not supported. The standard bytes-to-bytes codecs do not support this method.

reset()
Restablece los búfers de códec utilizados para mantener el estado interno.

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=- 1*, *chars=- 1*, *firstline=False*)
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=None*, *keepends=True*)
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=None*, *keepends=True*)
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 búfers de códec utilizados para mantener el estado interno.

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

Strings are stored internally as sequences of code points in range `U+0000`–`U+10FFFF`. (See **PEP 393** for more details about the implementation.) Once a string object is used outside of CPU and memory, endianness and how these arrays are stored as bytes become an issue. As with other codecs, serialising a string into a sequence of bytes is known as *encoding*, and recreating the string from the sequence of bytes is known as *decoding*. There are a variety of different text serialisation codecs, which are collectivity referred to as *text encodings*.

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.

All of these encodings can only encode 256 of the 1114112 code points defined in Unicode. A simple and straightforward way that can store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities: store the bytes in big endian or in little endian order. These two encodings are called `UTF-32-BE` and `UTF-32-LE` respectively. Their disadvantage is that if e.g. you use `UTF-32-BE` on a little endian machine you will always have to swap bytes on encoding and decoding. `UTF-32` avoids this problem: bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a `UTF-16` or `UTF-32` byte sequence, there's the so called BOM («Byte Order Mark»). This is the Unicode character `U+FEFF`. This character can be prepended to every `UTF-16` or `UTF-32` byte sequence. The byte swapped version of this character (`0xFFFE`) is an illegal character that may not appear in a Unicode text. So when the first character in a `UTF-16` or `UTF-32` byte sequence appears to be a `U+FFFE` the bytes have to be swapped on decoding. Unfortunately the character `U+FEFF` had a second purpose as a `ZERO WIDTH NO-BREAK SPACE`: a character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using `U+FEFF` as a `ZERO WIDTH NO-BREAK SPACE` has been deprecated (with `U+2060` (`WORD JOINER`) assuming this role). Nevertheless Unicode software still must be able to handle `U+FEFF` in both roles: as a BOM it's a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a string; as a `ZERO WIDTH NO-BREAK SPACE` it's a normal character that will be decoded like any other.

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
`U-00000000` ... `U-0000007F`	0xxxxxxx
`U-00000080` ... `U-000007FF`	110xxxxx 10xxxxxx
`U-00000800` ... `U-0000FFFF`	1110xxxx 10xxxxxx 10xxxxxx
`U-00010000` ... `U-0010FFFF`	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

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"*).

Without external information it's impossible to reliably determine which encoding was used for encoding a string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python calls `"utf-8-sig"`) for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: `0xef`, `0xbb`, `0xbf`) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

> 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 códecs integrados, ya sea implementados como funciones C o con diccionarios como tablas de mapeo. La siguiente tabla enumera los códecs 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'`.

Detalles de implementación de CPython: Algunas codificaciones comunes pueden omitir la maquinaria de búsqueda de códecs 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	Aliases	Lenguajes
ascii	646, us-ascii	Inglés
big5	big5-tw, csbig5	Chino Tradicional
big5hkscs	big5-hkscs, hkscs	Chino Tradicional
cp037	IBM037, IBM039	Inglés
cp273	273, IBM273, csIBM273	Alemán Nuevo en la versión 3.4.
cp424	EBCDIC-CP-HE, IBM424	Hebreo
cp437	437, IBM437	Inglés
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

continué en la próxima página

Tabla 1 – proviene de la página anterior

Códec	Aliases	Lenguajes
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
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

continué en la próxima página

Tabla 1 – proviene de la página anterior

Códec	Aliases	Lenguajes
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, mac_centeuro	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
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	Aliases	Significado
idna		Implementar **RFC 3490**, ver también `encodings.idna`. Solo se admite `errors='strict'`.
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.
palmos		Codificación de PalmOS 3.5.
punycode		Implementar **RFC 3492**. Los códecs con estado no son compatibles.
raw_unicode_escape		Codificación Latin-1 con `\uXXXX` y `\UXXXXXXXX` para otros puntos de código. Las barras invertidas existentes no se escapan de ninguna manera. Se usa en el protocolo Python *pickle*.
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	Aliases	Significado	Codificador / decodificador
base64_codec[1]	base64, base_64	Convierta el operando a MIME base64 multilínea (el resultado siempre incluye un `'\n'` final). Distinto en la versión 3.4: acepta cualquier *bytes-like object* como entrada para codificar y decodificar	`base64.encodebytes()` / `base64.decodebytes()`
bz2_codec	bz2	Comprime el operando usando bz2.	`bz2.compress()` / `bz2.decompress()`
hex_codec	hex	Convierte el operando en representación hexadecimal, con dos dígitos por byte.	`binascii.b2a_hex()` / `binascii.a2b_hex()`
quopri_codec	quopri, quoted-printable, quoted_printable	Convierte el operando a MIME citado imprimible.	`quopri.encode()` con `quotetabs=True` / `quopri.decode()`
uu_codec	uu	Convierte el operando usando uuencode.	`uu.encode()` / `uu.decode()`
zlib_codec	zip, zlib	Comprime el operando usando gzip.	`zlib.compress()` / `zlib.decompress()`

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	Aliases	Significado
rot_13	rot13	Retorna el cifrado César (*Caesar-cypher*) 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`.

[1] Además de *objetos similares a bytes*, `'base64_codec'` también acepta instancias solo ASCII de `str` para decodificación

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.

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 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.mbcs — 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 antepondrá 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.

CAPÍTULO 8

Tipos de 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

Código fuente: 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.

Module `zoneinfo` Concrete time zones representing the IANA time zone database.

Paquete dateutil Biblioteca de terceros con zona horaria ampliada y soporte de análisis.

8.1.1 Objetos conscientes (*aware*) y naífs (*naive*)

Los objetos de fecha y hora pueden clasificarse como conscientes (*aware*) o naífs (*naive*) dependiendo de si incluyen o no información de zona horaria.

Con suficiente conocimiento de los ajustes de tiempo políticos y algorítmicos aplicables, como la zona horaria y la información del horario de verano, un objeto **consciente** puede ubicarse en relación con otros objetos conscientes. Un objeto consciente representa un momento específico en el tiempo que no está abierto a interpretación.[1]

Un objeto **ingenuo** no contiene suficiente información para ubicarse sin ambigüedades en relación con otros objetos de fecha/hora. Si un objeto ingenuo representa la hora universal coordinada (UTC), la hora local o la hora en alguna otra zona horaria depende exclusivamente del programa, al igual que depende del programa si un número en particular representa metros, millas o masa. Los objetos ingenuos son fáciles de entender y trabajar con ellos, a costa de ignorar algunos aspectos de la realidad.

Para las aplicaciones que requieren objetos conscientes, los objetos `datetime` y `time` tienen un atributo de información de zona horaria opcional, `tzinfo`, que se puede establecer en una instancia de una subclase de la clase abstracta `tzinfo`. Estos objetos `tzinfo` capturan información sobre el desplazamiento de la hora UTC, el nombre de la zona horaria y si el horario de verano está en vigor.

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 `datetimeMAXYEAR` es‘ *9999*‘.

8.1.3 Tipos disponibles

`class` `datetime.`**`date`**
Una fecha naíf (*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`**
Una duración que expresa la diferencia entre dos instancias `date`, `time` o `datetime` a una resolución de microsegundos.

`class` `datetime.`**`tzinfo`**
Una clase base abstracta para objetos de información de zona horaria. Estos son utilizados por las clases `datetime` y `time` para proporcionar una noción personalizable de ajuste de hora (por ejemplo, para tener en cuenta la zona horaria y / o el horario de verano).

[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.
- Objects of these types are *hashable*, meaning that they can be used as dictionary keys.
- Los objetos de este tipo admiten el *pickling* eficiente a través del módulo *pickle*.

Determinando si un objeto es Consciente (*Aware*) o Naíf (*Naive*)

Los objetos del tipo *date* son siempre naíf (*naive*).

Un objeto de tipo *time* o *datetime* puede ser consciente (*aware*) o naíf (*naive*).

Un objeto *datetime* *d* es consciente si se cumplen los dos siguientes:

1. `d.tzinfo` no es `None`
2. `d.tzinfo.utcoffset(d)` no retorna `None`

De lo contrario, *d* es naíf (*naive*).

A *time* object *t* es consciente si ambos de los siguientes se mantienen:

1. `t.tzinfo` no es `None`
2. `t.tzinfo.utcoffset(None)` no retorna `None`.

De lo contrario, *t* es naíf (*naive*).

La distinción entre los objetos consciente (*aware*) y naíf (*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 <= microsegundos < 1000000`
- `0 <= segundos< 3600*24` (el número de segundos en un día)
- `-999999999 <= days <= 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).

Si el valor normalizado de los días se encuentra fuera del rango indicado, se lanza *`OverflowError`*.

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`**
: El objeto más negativo en *`timedelta`*, `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
`days`	Entre -999999999 y 999999999 inclusive
`seconds`	Entre 0 y 86399 inclusive
`microseconds`	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 multiplicado por un entero. Después *t1* // i == *t2* es verdadero, siempre que `i != 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 *timedelta.resolution* usando redondeo de medio a par.
`f = t2 / t3`	División (3) de la duración total *t2* por unidad de intervalo *t3*. Retorna un objeto `float`.
`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 *timedelta.resolution* usando redondeo de medio a par.
`t1 = t2 // i` o `t1 = t2 // t3`	El piso (*floor*) 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 `timedelta`. (3)
`q, r = 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 `timedelta`.
`+t1`	Retorna un objeto `timedelta` con el mismo valor. (2)
`-t1`	equivalente a `timedelta`(*-t1.days*, *-t1.seconds*, *-t1.microseconds*), y a *t1** -1. (1)(4)
`abs(t)`	equivalente a +*t* cuando `t.days>= 0`, y a -**t** cuando `t.days < 0`. (2)
`str(t)`	Retorna una cadena de caracteres en la forma `[D day[s], ][H]H:MM:SS[.UUUUUU]`, donde D es negativo para negativo `t`. (5)
`repr(t)`	Retorna una representación de cadena del objeto `timedelta` 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
`date2 = date1 + timedelta`	*date2* will be `timedelta.days` days after *date1*. (1)
`date2 = date1 - timedelta`	Calcula *date2* tal que `date2 + timedelta == date1`. (2)
`timedelta = date1 - date2`	(3)
`date1 < date2`	*date1* se considera menor que *date2* cuando *date1* precede a *date2* en el tiempo. (4)

Notas:

(1) *date2* se mueve hacia adelante en el tiempo si `timedelta.days > 0`, o hacia atrás si `timedelta.days < 0`. Después `date2 - date1 == timedelta.days`. `timedelta.seconds` y `timedelta.microseconds` se ignoran. *OverflowError* se lanza si `date2.year` sería menor que *MINYEAR* o mayor que *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 un objeto *named tuple* con tres componentes: `year`, `week` y `weekday`.

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()
datetime.IsoCalendarDate(year=2004, week=1, weekday=1)
>>> date(2004, 1, 4).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=7)
```

Distinto en la versión 3.9: El resultado cambió de una tupla a un *named tuple*.

`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] See R. H. van Gent's guide to the mathematics of the ISO 8601 calendar for a good explanation.

```
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 naíf (*naive*): *datetime*. Se puede obtener una fecha y hora UTC actual consciente (*aware*) llamando a datetime.now (timezone.utc). Ver también *now()*.

Advertencia: Debido a que los objetos naífs (*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(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 naíf (*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 naíf (*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 entre1970 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 naíf (*naive*) 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=timezone.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 proléptico gregoriano, donde el 1 de enero del año 1 tiene ordinal 1. *ValueError* se genera a menos que `1 <= ordinal <= 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 pensado 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
datetime2 = datetime1 + timedelta	(1)
datetime2 = datetime1 - timedelta	(2)
timedelta = datetime1 - datetime2	(3)
datetime1 < datetime2	Compara *datetime* to *datetime*. (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 naíf(*naive*), o si ambos son conscientes(*aware*). Si uno es consciente y el otro es naíf, *TypeError* aparece.

Si ambos son naíf (*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 naíf (*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 un de los elementos comparados es naíf (*naive*) y el otro lo sabe, se genera un *TypeError* si se intenta una comparación de órdenes. Para las comparaciones de igualdad, las instancias naíf 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 naíf (*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 naíf (*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:** Because naive `datetime` objects are treated by many `datetime` methods as local times, it is preferred to use aware datetimes to represent times in UTC; as a result, using *datetime.utctimetuple()* may give misleading results. If you have a naive `datetime` representing UTC, use `datetime.replace(tzinfo=timezone.utc)` to make it aware, at which point you can use *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()*.

Se supone que las instancias Naíf *datetime* representan la hora local y este método se basa en la función de plataforma C `mktime()` para realizar la conversión. Dado que *datetime* admite un rango de valores más amplio que `mktime()` en muchas plataformas, este método puede lanzar *OverflowError* para tiempos lejanos en el pasado o en el futuro.

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 naíf *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 *named tuple* con tres componentes: `year`, `week`, y `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.ffffff".

Nota: Los componentes de tiempo excluidos están truncados, no redondeados.

ValueError lanzará 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[.ffffff]]]
```

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` lanzará 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 detonado 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
# https://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.stdoffset
        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:

zoneinfo 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).

`zoneinfo` brings the *IANA timezone database* (also known as the Olson database) to Python, and its usage is recommended.

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.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 ± es el signo de `offset`, HH y MM son dos dígitos de `offset.hours` y `offset.minutes` respectivamente.

Distinto en la versión 3.6: Name generated from `offset=timedelta(0)` is now plain `'UTC'`, not `'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()`:

	`strftime`	`strptime`
Uso	Convierte objetos en una cadena de caracteres de acuerdo con un formato dado	*parsear* una cadena en un objeto *datetime* con el formato correspondiente
Tipo de método	Método de instancia	Método de clase
Método de	*date*; *datetime*; *time*	*datetime*
Firma	`strftime(format)`	`strptime(date_string, format)`

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.	*Sun, Mon, …, Sat (en_US)*; *So, Mo, …, Sa (de_DE)*	(1)
%A	Día de la semana como nombre completo de la localidad.	*Sunday, Monday, …, Saturday (en_US)*; *Sonntag, Montag, …, Samstag (de_DE)*	(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.	*Jan, Feb, …, Dec (en_US)*; *Jan, Feb, …, Dez (de_DE)*	(1)
%B	Mes como nombre completo según la configuración regional.	*January, February, …, December (en_US)*; *Januar, Februar, …, Dezember (de_DE)*	(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 (en_US); am, pm (de_DE)	(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	Microsecond as a decimal number, zero-padded to 6 digits.	000000, 000001, …, 999999	(5)
%z	Desplazamiento (*offset*) UTC en la forma ±HHMM[SS[.ffffff]] (cadena de	(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 año con siglo que representa el año que contiene la mayor parte de la semana ISO (`%V`).	0001, 0002, …, 2013, 2014, …, 9998, 9999	(8)
`%u`	ISO 8601 día de la semana como un número decimal donde 1 es lunes.	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 `strftime()`. 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 lanzará un *ValueError*.

El conjunto completo de códigos de formato admitidos varía según las plataformas, porque Python llama a la función `strftime()` de la biblioteca de la plataforma C, y las variaciones de la plataforma son comunes. Para ver el conjunto completo de códigos de formato admitidos en su plataforma, consulte la documentación *strftime(3)*. También existen diferencias entre plataformas en el manejo de especificadores de formato no admitidos.

Nuevo en la versión 3.6: `%G`, `%u` y `%V` fueron añadidos.

Detalle técnico

En términos generales, `d.strftime (fmt)` actúa como el módulo *time* `time.strftime(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 `strftime` 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.

[4] Si se pasa `datetime.strptime ('29 de febrero', '%b %d')` fallará ya que `1900` no es un año bisiesto.

Distinto en la versión 3.2: En versiones anteriores, el método `strftime()` estaba restringido a años >= 1900.

Distinto en la versión 3.3: En la versión 3.2, el método `strftime()` 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 En `strftime()`, `%Z` se reemplaza por una cadena de caracteres vacía si `tzname()` retorna `None`; de lo contrario, `%Z` se reemplaza por el valor retornado, que debe ser una cadena de caracteres.

`strptime()` solo acepta ciertos valores para `%Z`:

1. cualquier valor en `time.tzname` para la configuración regional de su máquina
2. los valores codificados de forma rígida `UTC` y `GMT`

Entonces, alguien que viva en Japón puede tener `JST`, `UTC` y `GMT` como valores válidos, pero probablemente no `EST`. Lanzará `ValueError` para valores no válidos.

Distinto en la versión 3.2: Cuando la directiva `%z` se proporciona al método `strptime()`, se lanzará 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) When used with the `strptime()` method, the leading zero is optional for formats `%d`, `%m`, `%H`, `%I`, `%M`, `%S`, `%j`, `%U`, `%W`, and `%V`. Format `%y` does require a leading zero.

Pie de notas

8.2 zoneinfo — Soporte de zona horaria IANA

Nuevo en la versión 3.9.

Source code: Lib/zoneinfo

El módulo *zoneinfo* proporciona una implementación concreta de zonas horarias para soportar la base de datos de zonas horarias de IANA tal y como se especificó originalmente en **PEP 615**. Por defecto, *zoneinfo* utiliza los datos de zona horaria del sistema si están disponibles; si no hay datos de zona horaria del sistema, la biblioteca volverá a utilizar el paquete de primera parte tzdata disponible en PyPI.

Ver también:

Modulo: *datetime* Proporciona los tipos *time* y *datetime* con los que está diseñada la clase *ZoneInfo*.

Paquete tzdata Paquete de origen mantenido por los desarrolladores del núcleo de CPython para suministrar datos de zonas horarias a través de PyPI.

8.2.1 Usando ZoneInfo

ZoneInfo es una implementación concreta de la clase base abstracta *datetime.tzinfo*, y está pensada para ser adjuntada a `tzinfo`, bien a través del constructor, del método *datetime.replace* o de *datetime.astimezone*:

```
>>> from zoneinfo import ZoneInfo
>>> from datetime import datetime, timedelta

>>> dt = datetime(2020, 10, 31, 12, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-10-31 12:00:00-07:00

>>> dt.tzname()
'PDT'
```

Las fechas construidas de esta manera son compatibles con la aritmética de fechas y manejan las transiciones del horario de verano sin ninguna otra intervención:

```
>>> dt_add = dt + timedelta(days=1)

>>> print(dt_add)
2020-11-01 12:00:00-08:00

>>> dt_add.tzname()
'PST'
```

Estas zonas horarias también soportan el atributo *fold* introducido en **PEP 495**. Durante las transiciones de desfase que inducen tiempos ambiguos (como una transición de horario de verano a horario estándar), se utiliza el desfase de *antes* de la transición cuando `fold=0`, y el desfase *después* de la transición cuando `fold=1`, por ejemplo:

```
>>> dt = datetime(2020, 11, 1, 1, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-11-01 01:00:00-07:00

>>> print(dt.replace(fold=1))
2020-11-01 01:00:00-08:00
```

Al convertir desde otra zona horaria, el pliegue se ajustará al valor correcto:

```
>>> from datetime import timezone
>>> LOS_ANGELES = ZoneInfo("America/Los_Angeles")
>>> dt_utc = datetime(2020, 11, 1, 8, tzinfo=timezone.utc)

>>> # Before the PDT -> PST transition
>>> print(dt_utc.astimezone(LOS_ANGELES))
2020-11-01 01:00:00-07:00

>>> # After the PDT -> PST transition
>>> print((dt_utc + timedelta(hours=1)).astimezone(LOS_ANGELES))
2020-11-01 01:00:00-08:00
```

8.2.2 Fuentes de datos

El módulo `zoneinfo` no proporciona directamente los datos de la zona horaria, y en su lugar extrae la información de la zona horaria de la base de datos de la zona horaria del sistema o del paquete de PyPI tzdata, si está disponible. Algunos sistemas, incluyendo notablemente los sistemas Windows, no tienen una base de datos IANA disponible, por lo que para los proyectos que tienen como objetivo la compatibilidad entre plataformas que requieren datos de zona horaria, se recomienda declarar una dependencia de tzdata. Si no están disponibles ni los datos del sistema ni los de tzdata, todas las llamadas a *ZoneInfo* lanzarán un *ZoneInfoNotFoundError*.

Configurando los orígenes de datos

Cuando se llama a `ZoneInfo(key)`, el constructor busca primero en los directorios especificados en *TZPATH* un archivo que coincida con `key`, y en caso de fallo busca una coincidencia en el paquete tzdata. Este comportamiento puede configurarse de tres maneras:

1. El *TZPATH* por defecto cuando no se especifica otra cosa puede configurarse en *tiempo de compilación*.
2. *TZPATH* puede configurarse utilizando *una variable de entorno*.
3. En *runtime*, la ruta de búsqueda puede ser manipulada usando la función *reset_tzpath()*.

Configuración en tiempo de compilación

El *TZPATH* predeterminado incluye varias ubicaciones de implementación comunes para la base de datos de la zona horaria (excepto en Windows, donde no hay ubicaciones «conocidas» para los datos de la zona horaria). En los sistemas POSIX, los distribuidores posteriores y aquellos que compilan Python desde la fuente que saben dónde se implementan los datos de la zona horaria del sistema pueden cambiar la ruta de la zona horaria predeterminada especificando la opción de tiempo de compilación `TZPATH` (o, más probablemente, el `configure flag --with-tzpath`), que debería ser una cadena delimitada por *os.pathsep*.

En todas las plataformas, el valor configurado está disponible como la clave `TZPATH` en *sysconfig.get_config_var()*.

Configuración del entorno

Cuando se inicializa *TZPATH* (ya sea en el momento de la importación o cuando se llama a *reset_tzpath()* sin argumentos), el módulo `zoneinfo` utilizará la variable de entorno `PYTHONTZPATH`, si existe, para establecer la ruta de búsqueda.

PYTHONTZPATH

Se trata de una cadena separada por *os.pathsep* que contiene la ruta de búsqueda de la zona horaria a utilizar. Debe consistir sólo en rutas absolutas y no relativas. Los componentes relativos especificados en `PYTHONTZPATH` no se utilizarán, pero por lo demás el comportamiento cuando se especifica una ruta relativa es definido por la implementación; CPython lanzará *InvalidTZPathWarning*, pero otras implementaciones son libres de ignorar silenciosamente el componente erróneo o lanzar una excepción.

Para que el sistema ignore los datos del sistema y utilice el paquete tzdata en su lugar, establezca `PYTHONTZPATH=""`.

Configuración de tiempo de ejecución

La ruta de búsqueda de TZ también puede configurarse en tiempo de ejecución mediante la función *reset_tzpath()*. Por lo general, esta operación no es aconsejable, aunque es razonable utilizarla en funciones de prueba que requieran el uso de una ruta de zona horaria específica (o que requieran deshabilitar el acceso a las zonas horarias del sistema).

8.2.3 La clase ZoneInfo

class zoneinfo.**ZoneInfo**(*key*)

Una subclase concreta de *datetime.tzinfo* que representa una zona horaria IANA especificada por la cadena `key`. Las llamadas al constructor primario siempre devolverán objetos que se comparan de forma idéntica; dicho de otro modo, salvo la invalidación de la caché mediante *ZoneInfo.clear_cache()*, para todos los valores de `key`, la siguiente afirmación siempre será verdadera:

```
a = ZoneInfo(key)
b = ZoneInfo(key)
assert a is b
```

La `key` debe tener la forma de una ruta POSIX relativa y normalizada, sin referencias de nivel superior. El constructor lanzará *ValueError* si se pasa una clave no conforme.

Si no se encuentra ningún archivo que coincida con la `clave`, el constructor lanzará *ZoneInfoNotFoundError*.

La clase `ZoneInfo` tiene dos constructores alternativos:

classmethod ZoneInfo.**from_file**(*fobj*, */*, *key=None*)

Construye un objeto `ZoneInfo` a partir de un objeto tipo archivo que retorna bytes (por ejemplo, un archivo abierto en modo binario o un objeto *io.BytesIO*). A diferencia del constructor primario, éste siempre construye un nuevo objeto.

El parámetro `key` establece el nombre de la zona a efectos de `__str__()` y `__repr__()`.

Los objetos creados a través de este constructor no pueden ser serializados (ver *pickling*).

classmethod ZoneInfo.**no_cache**(*key*)

Un constructor alternativo que omite la caché del constructor. Es idéntico al constructor principal, pero retorna un nuevo objeto en cada llamada. Es muy probable que esto sea útil para propósitos de prueba o demostración, pero también se puede utilizar para crear un sistema con una estrategia de invalidación de caché diferente.

Los objetos creados a través de este constructor también pasarán por encima de la caché de un proceso de deserialización cuando sean deserializados.

> **Prudencia:** El uso de este constructor puede cambiar la semántica de tus datetimes de manera sorprendente, sólo úsalo si sabes que lo necesitas.

También están disponibles los siguientes métodos de clase:

classmethod ZoneInfo.**clear_cache**(*, *only_keys=None*)

Un método para invalidar la caché de la clase `ZoneInfo`. Si no se pasan argumentos, se invalidan todas las cachés y la siguiente llamada al constructor primario de cada clave devolverá una nueva instancia.

Si se pasa un iterable de nombres de claves al parámetro `only_keys`, sólo se eliminarán de la caché las claves especificadas. Las claves pasadas a `only_keys` pero que no se encuentran en la caché se ignoran.

> **Advertencia:** La invocación de esta función puede cambiar la semántica de las fechas utilizando `ZoneInfo` de forma sorprendente; esto modifica el estado global del proceso y por lo tanto puede tener efectos de gran alcance. Utilícela sólo si sabe que lo necesita.

La clase tiene un atributo:

`ZoneInfo.key`
: Se trata de un *attribute* de sólo lectura que retorna el valor de `key` pasado al constructor, que debe ser una clave de búsqueda en la base de datos de zonas horarias de la IANA (por ejemplo, `America/New_York`, `Europe/Paris` o `Asia/Tokyo`.

  Para las zonas construidas a partir de un archivo sin especificar un parámetro `key`, se establecerá como `None`.

Nota: Aunque es una práctica algo común exponerlos a los usuarios finales, estos valores están diseñados para ser claves primarias para representar las zonas relevantes y no necesariamente elementos orientados al usuario. Se pueden utilizar proyectos como CLDR (Unicode Common Locale Data Repository) para obtener cadenas más fáciles de usar a partir de estas claves.

Representaciones de cadenas

La representación de cadena que se retorna al llamar a *`str`* sobre un objeto *`ZoneInfo`* utiliza por defecto el atributo *`ZoneInfo.key`* (ver la nota de uso en la documentación del atributo):

```
>>> zone = ZoneInfo("Pacific/Kwajalein")
>>> str(zone)
'Pacific/Kwajalein'

>>> dt = datetime(2020, 4, 1, 3, 15, tzinfo=zone)
>>> f"{dt.isoformat()} [{dt.tzinfo}]"
'2020-04-01T03:15:00+12:00 [Pacific/Kwajalein]'
```

Para los objetos construidos a partir de un fichero sin especificar un parámetro `key`, `str` vuelve a llamar a `repr`()`. El parámetro `repr` de `ZoneInfo` está definido por la implementación y no es necesariamente estable entre versiones, pero se garantiza que no es una clave válida de `ZoneInfo`.Pickled.

Serialización de Pickle

En lugar de serializar todos los datos de transición, los objetos `ZoneInfo` se serializan por clave, y los objetos `ZoneInfo` construidos a partir de archivos (incluso los que tienen un valor por `key` especifico) no pueden ser serializados.

El comportamiento de un archivo `ZoneInfo` depende de cómo se haya construido:

1. `ZoneInfo(key)`: Cuando se construye con el constructor primario, un objeto `ZoneInfo` se serializa por la clave, y cuando se deserializa, el proceso de deserialización utiliza el primario y por lo tanto se espera que estos sean el mismo objeto que otras referencias a la misma zona horaria. Por ejemplo, si `europe_berlin_pkl` es una cadena que contiene un pickle construido a partir de `ZoneInfo("Europe/Berlin")`, se esperaría el siguiente comportamiento:

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl)
   >>> a is b
   True
   ```

2. `ZoneInfo.no_cache(key)`: Cuando se construye a partir del constructor que evita la caché, el objeto `ZoneInfo` también se serializa por clave, pero cuando se deserializa, el proceso de deserialización utiliza el constructor que evita la caché. Si `europe_berlin_pkl_nc` es una cadena que contiene un pickle

construido a partir de `ZoneInfo.no_cache("Europe/Berlin")`, cabría esperar el siguiente comportamiento:

```
>>> a = ZoneInfo("Europe/Berlin")
>>> b = pickle.loads(europe_berlin_pkl_nc)
>>> a is b
False
```

3. `ZoneInfo.from_file(fobj, /, key=None)`: Cuando se construye a partir de un fichero, el objeto `ZoneInfo` lanza una excepción al ser recogido. Si un usuario final quiere recoger un `ZoneInfo` construido a partir de un archivo, se recomienda que utilice un tipo de envoltura o una función de serialización personalizada: ya sea serializando por clave o almacenando el contenido del objeto archivo y serializándolo.

Este método de serialización requiere que los datos de la zona horaria para la clave requerida estén disponibles tanto en el lado de serialización como en el de deserialización, de forma similar a como se espera que las referencias a las clases y funciones existan tanto en el entorno de serialización como en el de deserialización. También significa que no se garantiza la consistencia de los resultados cuando se retira un `ZoneInfo` recogido en un entorno con una versión diferente de los datos de la zona horaria.

8.2.4 Funciones

`zoneinfo.`**`available_timezones`**`()`

Obtiene un conjunto que contiene todas las claves válidas para las zonas horarias de la IANA disponibles en cualquier lugar de la ruta de zonas horarias. Se recalcula en cada llamada a la función.

Esta función sólo incluye los nombres de zona canónicos y no incluye las zonas «especiales» como las que se encuentran bajo los directorios `posix/` y `right/`, o la zona `posixrules`.

> **Prudencia:** Esta función puede abrir un gran número de archivos, ya que la mejor manera de determinar si un archivo en la ruta de la zona horaria es una zona horaria válida es leer la «magic string» (cadena mágica) al principio.

Nota: Estos valores no están diseñados para ser expuestos a los usuarios finales; para los elementos de cara al usuario, las aplicaciones deberían utilizar algo como CLDR (el Unicode Common Locale Data Repository) para obtener cadenas más fáciles de usar. Véase también la nota de advertencia sobre *`ZoneInfo.key`*.

`zoneinfo.`**`reset_tzpath`**`(`*`to=None`*`)`

Establece o restablece la ruta de búsqueda de la zona horaria (*`TZPATH`*) para el módulo. Cuando se llama sin argumentos, *`TZPATH`* se establece en el valor por defecto.

La llamada a `reset_tzpath` no invalidará la caché de *`ZoneInfo`*, por lo que las llamadas al constructor primario de `ZoneInfo` sólo utilizarán el nuevo `TZPATH` en caso de que se pierda la caché.

El parámetro `to` debe ser un *sequence* de cadenas o *`os.PathLike`* y no una cadena, todos los cuales deben ser rutas absolutas. *`ValueError`* se lanzará si se pasa algo que no sea una ruta absoluta.

8.2.5 Globales

zoneinfo.**TZPATH**
Una secuencia de sólo lectura que representa la ruta de búsqueda de zonas horarias – cuando se construye un ZoneInfo a partir de una clave, la clave se une a cada entrada del TZPATH, y se utiliza el primer archivo encontrado.

TZPATH sólo puede contener rutas absolutas, nunca relativas, independientemente de cómo esté configurado.

El objeto al que apunta zoneinfo.TZPATH puede cambiar en respuesta a una llamada a *reset_tzpath()*, por lo que se recomienda utilizar zoneinfo.TZPATH en lugar de importar TZPATH desde zoneinfo o asignar una variable de larga duración a zoneinfo.TZPATH.

Para más información sobre la configuración de la ruta de búsqueda de zonas horarias, consulte *Configurando los orígenes de datos*.

8.2.6 Excepciones y advertencias

exception zoneinfo.**ZoneInfoNotFoundError**
Se lanza cuando la construcción de un objeto *ZoneInfo* falla porque la clave especificada no puede encontrarse en el sistema. Es una subclase de *KeyError*.

exception zoneinfo.**InvalidTZPathWarning**
Se lanza cuando *PYTHONTZPATH* contiene un componente no válido que será filtrado, como una ruta relativa.

8.3 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 proléptico» 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*)
Creates a *Calendar* object. *firstweekday* is an integer specifying the first day of the week. *MONDAY* is 0 (the default), *SUNDAY* is 6.

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*)
Retorna una matriz que representa el calendario de un mes. Cada fila representa una semana; los días fuera del mes se representan con ceros. Cada semana comienza con el lunes a menos que lo establezca *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_abbr**
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_abbr[0] es la cadena vacía.

calendar.**MONDAY**
calendar.**TUESDAY**
calendar.**WEDNESDAY**
calendar.**THURSDAY**
calendar.**FRIDAY**
calendar.**SATURDAY**
calendar.**SUNDAY**
Aliases for day numbers, where MONDAY is 0 and SUNDAY is 6.

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.

8.4 `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`.

`namedtuple()`	función *factory* para crear subclases de *tuplas* con campos con nombre
`deque`	contenedor similar a una lista con *appends* y *pops* rápidos en ambos extremos
`ChainMap`	clase similar a *dict* para crear una vista única de múltiples *mapeados*
`Counter`	dict subclass for counting *hashable* objects
`OrderedDict`	subclase de *dict* que recuerda las entradas de la orden que se agregaron
`defaultdict`	subclase de *dict* que llama a una función de *factory* para suministrar valores faltantes
`UserDict`	envoltura alrededor de los objetos de diccionario para facilitar subclasificaciones *dict*
`UserList`	envoltura alrededor de los objetos de lista para facilitar la subclasificación de un *list*
`UserString`	envoltura alrededor de objetos de cadena para facilitar la subclasificación de *string*

8.4.1 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*, ***kwargs*)

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 diccionario vacío, de modo que una llamada a `d.new_child()` es equivalente a: `ChainMap({}, *d.maps)`. Si se especifica algún argumento de palabra clave, actualiza el mapa pasado o el nuevo diccionario vacío. 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` .

Distinto en la versión 3.10: Se añadió soporte para argumentos por palabras clave.

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']
```

Distinto en la versión 3.9: Se agregó soporte para los operadores `|` y `|=`, especificados en **PEP 584**.

Ver también:

- La clase MultiContext en el paquete de Enthought llamado CodeTools tiene opciones para admitir la escritura en cualquier mapeo de la cadena.
- La clase de Django Context para crear plantillas es una cadena de mapeo de solo lectura. También presenta características de pushing y popping de contextos similar al método *new_child()* y a la propiedad *parents* .
- 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}
```

(continué en la próxima página)

(proviene de la página anterior)

```
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'})
```

8.4.2 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*])

A *Counter* is a *dict* subclass for counting *hashable* objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The *Counter* class is similar to bags or multisets in other languages.

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: As a *dict* subclass, *Counter* inherited the capability to remember insertion order. Math operations on *Counter* objects also preserve order. Results are ordered according to when an element is first encountered in the left operand and then by the order encountered in the right operand.

Counter objects support additional methods beyond those available for all dictionaries:

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.

total()
Computa la suma de las cuentas.

```
>>> c = Counter(a=10, b=5, c=0)
>>> c.total()
15
```

Nuevo en la versión 3.10.

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)`.

Los contadores admiten operadores de comparación ricos para las relaciones de igualdad, subconjunto, y superconjunto: ==, !=, <, <=, >, >=. Todas esas pruebas tratan los elementos faltantes como si tuvieran cero recuentos, por lo que `Counter(a=1) == Counter(a=1, b=0)` retorne verdadero.

Nuevo en la versión 3.10: Rich comparison operations were added.

Distinto en la versión 3.10: En pruebas de igualdad, los elementos faltantes son tratados como si tuvieran cero recuentos. Anteriormente, `Counter(a=3)` y `Counter(a=3, b=0)` se consideraban distintos.

Patrones comunes para trabajar con objetos *Counter*:

```
c.total()                       # 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
```

Several mathematical operations are provided for combining *Counter* objects to produce multisets (counters that have counts greater than zero). Addition and subtraction combine counters by adding or subtracting the counts of corresponding elements. Intersection and union return the minimum and maximum of corresponding counts. Equality and inclusion compare corresponding counts. Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                       # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                       # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                       # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                       # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
>>> c == d                      # equality:  c[x] == d[x]
False
>>> c <= d                      # inclusion:  c[x] <= d[x]
False
```

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
```

8.4.3 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 incurren 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 deques 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 deques 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*)

Cuente 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*)

Ingrese *x* en el dique 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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 appending 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
    # https://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])
```

(continué en la próxima página)

(proviene de la página anterior)

```
                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`.

8.4.4 Objetos `defaultdict`

class collections.**defaultdict**(*default_factory=None*, */*[, ...])

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.

Distinto en la versión 3.9: Se agregaron los operadores unir (`|`) y actualizar (`|=`), especificados en **PEP 584**.

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})]
```

8.4.5 `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 autodocumentado. 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.

Para admitir el serializado (*pickling*), la clase tupla con nombre debe asignarse a una variable que coincida con *typename*.

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 él 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 tut-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):
...         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
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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.

8.4.6 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.
- The *OrderedDict* algorithm can handle frequent reordering operations better than *dict*. As shown in the recipes below, this makes it suitable for implementing various kinds of LRU caches.
- La operación de igualdad para *OrderedDict* comprueba el orden coincidente.

  A regular *dict* can emulate the order sensitive equality test with `p == q and all(k1 == k2 for k1, k2 in zip(p, q))`.
- El método `popitem()` de *OrderedDict* tiene una firma diferente. Acepta un argumento opcional para especificar qué elemento es *popped*.

  A regular *dict* can emulate OrderedDict's `od.popitem(last=True)` with `d.popitem()` which is guaranteed to pop the rightmost (last) item.

  A regular *dict* can emulate OrderedDict's `od.popitem(last=False)` with `(k := next(iter(d)), d.pop(k))` which will return and remove the leftmost (first) item if it exists.
- *OrderedDict* tiene un método `move_to_end()` para reposiciones eficientemente un elemento a un punto final.

  A regular *dict* can emulate OrderedDict's `od.move_to_end(k, last=True)` with `d[k] = d.pop(k)` which will move the key and its associated value to the rightmost (last) position.

  A regular *dict* does not have an efficient equivalent for OrderedDict's `od.move_to_end(k, last=False)` which moves the key and its associated value to the leftmost (first) position.
- 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*)

Move an existing *key* to either end of an ordered dictionary. The item is moved to the right end if *last* is true (the default) or to the beginning if *last* is false. Raises *KeyError* if the *key* does not exist:

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d)
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d)
'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()`.

Distinto en la versión 3.9: Se agregaron los operadores unir (`|`) y actualizar (`|=`), especificados en **PEP 584**.

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)
```

Un *OrderedDict* también sería útil para implementar variantes de *functools.lru_cache()*:

```
from time import time

class TimeBoundedLRU:
    "LRU Cache that invalidates and refreshes old entries."

    def __init__(self, func, maxsize=128, maxage=30):
        self.cache = OrderedDict()      # { args : (timestamp, result)}
        self.func = func
        self.maxsize = maxsize
        self.maxage = maxage

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            timestamp, result = self.cache[args]
            if time() - timestamp <= self.maxage:
                return result
        result = self.func(*args)
        self.cache[args] = time(), result
        if len(self.cache) > self.maxsize:
```

(continué en la próxima página)

(proviene de la página anterior)

```
            self.cache.popitem(0)
        return result
```

```
class MultiHitLRUCache:
    """ LRU cache that defers caching a result until
        it has been requested multiple times.

        To avoid flushing the LRU cache with one-time requests,
        we don't cache until a request has been made more than once.

    """

    def __init__(self, func, maxsize=128, maxrequests=4096, cache_after=1):
        self.requests = OrderedDict()   # { uncached_key : request_count }
        self.cache = OrderedDict()      # { cached_key : function_result }
        self.func = func
        self.maxrequests = maxrequests  # max number of uncached requests
        self.maxsize = maxsize          # max number of stored return values
        self.cache_after = cache_after

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            return self.cache[args]
        result = self.func(*args)
        self.requests[args] = self.requests.get(args, 0) + 1
        if self.requests[args] <= self.cache_after:
            self.requests.move_to_end(args)
            if len(self.requests) > self.maxrequests:
                self.requests.popitem(0)
        else:
            self.requests.pop(args, None)
            self.cache[args] = result
            if len(self.cache) > self.maxsize:
                self.cache.popitem(0)
        return result
```

8.4.7 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 de *`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`* .

8.4.8 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` .

Requisitos de subclases: Se espera que las subclases de `UserList` ofrezcan un constructor al que se pueda llamar sin argumentos o con un solo argumento. Las operaciones de lista que retornan una nueva secuencia intentan crear una instancia de la clase de implementación real. Para hacerlo, asume que se puede llamar al constructor con un solo parámetro, que es un objeto de secuencia utilizado como fuente de datos.

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.

8.4.9 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`.

8.5 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

This module provides *abstract base classes* that can be used to test whether a class provides a particular interface; for example, whether it is *hashable* or whether it is a mapping.

Una prueba *issubclass()* o *isinstance()* para una interfaz funciona de tres formas.

1) A newly written class can inherit directly from one of the abstract base classes. The class must supply the required abstract methods. The remaining mixin methods come from inheritance and can be overridden if desired. Other methods may be added as needed:

```
class C(Sequence):                          # Direct inheritance
    def __init__(self): ...                 # Extra method not required by the ABC
    def __getitem__(self, index):  ...      # Required abstract method
    def __len__(self):  ...                 # Required abstract method
    def count(self, value): ...             # Optionally override a mixin method
```

```
>>> issubclass(C, Sequence)
True
>>> isinstance(C(), Sequence)
True
```

2) Existing classes and built-in classes can be registered as «virtual subclasses» of the ABCs. Those classes should define the full API including all of the abstract methods and all of the mixin methods. This lets users rely on *issubclass()* or *isinstance()* tests to determine whether the full interface is supported. The exception to this rule is for methods that are automatically inferred from the rest of the API:

```
class D:                                    # No inheritance
    def __init__(self): ...                 # Extra method not required by the ABC
    def __getitem__(self, index):  ...      # Abstract method
    def __len__(self):  ...                 # Abstract method
    def count(self, value): ...             # Mixin method
    def index(self, value): ...             # Mixin method

Sequence.register(D)                        # Register instead of inherit
```

```
>>> issubclass(D, Sequence)
True
>>> isinstance(D(), Sequence)
True
```

En este ejemplo, la clase `D` no necesita definir `__contains__`, `__iter__` y `__reversed__` porque el operador in, la lógica de *iteración* y la función *reversed()* recurren automáticamente al uso de `__getitem__` y `__len__`.

3) Some simple interfaces are directly recognizable by the presence of the required methods (unless those methods have been set to *None*):

```
class E:
    def __iter__(self): ...
    def __next__(next): ...
```

```
>>> issubclass(E, Iterable)
True
>>> isinstance(E(), Iterable)
True
```

Las interfaces complejas no admiten esta última técnica porque una interfaz es más que la simple presencia de nombres de métodos. Las interfaces especifican la semántica y las relaciones entre métodos que no se pueden inferir únicamente de la presencia de nombres de métodos específicos. Por ejemplo, saber que una clase proporciona `__getitem__`, `__len__` y `__iter__` no es suficiente para distinguir un *Sequence* de un *Mapping*.

Nuevo en la versión 3.9: These abstract classes now support `[]`. See *Tipo Alias Genérico* and **PEP 585**.

8.5.1 Colecciones Clases Base Abstractas

El módulo de colecciones ofrece lo siguiente *ABCs*:

ABC	Hereda de	Métodos Abstractos	Métodos Mixin
Container[1]		`__contains__`	
Hashable[1]		`__hash__`	
Iterable[1,2]		`__iter__`	
Iterator[1]	*Iterable*	`__next__`	`__iter__`
Reversible[1]	*Iterable*	`__reversed__`	
Generator[1]	*Iterator*	`send`, `throw`	`close`, `__iter__`, `__next__`
Sized[1]		`__len__`	
Callable[1]		`__call__`	
Collection[1]	*Sized*, *Iterable*, *Container*	`__contains__`, `__iter__`, `__len__`	
Sequence	*Reversible*, *Collection*	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
MutableSequence	*Sequence*	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Métodos heredados *Sequence* y `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
ByteString	*Sequence*	`__getitem__`, `__len__`	Métodos heredados *Sequence*
Set	*Collection*	`__contains__`, `__iter__`, `__len__`	`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__`, `__sub__`, `__xor__`, and `isdisjoint`
MutableSet	*Set*	`__contains__`, `__iter__`, `__len__`, `add`, `discard`	Métodos heredados *Set* y `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__`, and `__isub__`
Mapping	*Collection*	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, and `__ne__`
MutableMapping	*Mapping*	`__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`	Métodos heredados *Mapping* y `pop`, `popitem`, `clear`, `update`, and `setdefault`
MappingView	*Sized*		`__len__`
ItemsView	*MappingView*, *Set*		`__contains__`, `__iter__`
KeysView	*MappingView*, *Set*		`__contains__`, `__iter__`
ValuesView	*MappingView*, *Collection*		`__contains__`, `__iter__`
Awaitable[1]		`__await__`	
Coroutine[1]	*Awaitable*	`send`, `throw`	`close`
AsyncIterable[1]		`__aiter__`	
AsyncIterator[1]	*AsyncIterable*	`__anext__`	`__aiter__`
AsyncGenerator[1]	*AsyncIterator*	`asend`, `athrow`	`aclose`, `__aiter__`, `__anext__`

[1] Estos ABC anulan `object.__subclasshook__()` para admitir la prueba de una interfaz verificando que los métodos requeridos estén presentes y no se hayan configurado en *None*. Esto solo funciona para interfaces simples. Las interfaces más complejas requieren registro o herencia directa.

[2] La verificación de `isinstance(obj, Iterable)` detecta 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)`.

Notas al pie

8.5.2 Colecciones Clases base abstractas - Descripciones detalladas

class collections.abc.**Container**
ABC para clases que proporcionan el método `__contains__()`.

class collections.abc.**Hashable**
ABC para clases que proporcionan el método `__hash__()`.

class collections.abc.**Sized**
ABC para clases que proporcionan el método `__len__()`.

class collections.abc.**Callable**
ABC para clases que proporcionan el método `__call__()`.

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 objetos e instancias de la clase `Coroutine` ABC son todas las instancias de este 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.

8.5.3 Ejemplos y Recetas

Los ABC nos permiten preguntar a las clases o instancias si brindan 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')
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 a partir de un iterable. Se supone que el constructor de la clase tiene una firma con el formato `ClassName(iterable)`. Esa suposición se factoriza en un método de clase interno llamado `_from_iterable()` que llama a `cls(iterable)` para producir un nuevo conjunto. Si el mixin *Set* se está usando en una clase con una firma de constructor diferente, necesitarás anular `_from_iterable()` con un método de clase o método regular 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) The *Set* mixin provides a `_hash()` method to compute a hash value for the set; however, `__hash__()` is not defined because not all sets are *hashable* or immutable. To add set hashability using mixins, inherit from both *Set()* and *Hashable()*, then define `__hash__ = Set._hash`.

Ver también:

- OrderedSet receta para un ejemplo basado en *MutableSet*.
- Para obtener más información sobre ABCs, ver el módulo *abc* y **PEP 3119**.

8.6 `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 *iem* 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 apile el nuevo *item*. El tamaño del montículo no cambia. Si el montículo está vacío, *IndexError* se lanza.

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.

8.6.1 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')
```

8.6.2 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')
```

8.6.3 Teoría

Los montículos son conjuntos para los cuales“a[k] <= a[2*k+1]“ y «a[k] <= a[2*k+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]“:

```
                               0

              1                                 2

      3               4                 5               6

  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

En el árbol de arriba, cada celda *k* está coronada por `2*k+1` y `2*k+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 organizada[1]. 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

8.7 `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

[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! :-)

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)*, *, *key=None*)
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* divide el arreglo *a* en dos mitades, de modo 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 lado derecho.

key specifies a *key function* of one argument that is used to extract a comparison key from each element in the array. To support searching complex records, the key function is not applied to the *x* value.

If *key* is `None`, the elements are compared directly with no intervening function call.

Distinto en la versión 3.10: Se agregó el parámetro *key*.

bisect.**bisect_right**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)
bisect.**bisect**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)
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* divide el arreglo *a* en dos mitades, de modo 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 lado derecho.

key specifies a *key function* of one argument that is used to extract a comparison key from each element in the array. To support searching complex records, the key function is not applied to the *x* value.

If *key* is `None`, the elements are compared directly with no intervening function call.

Distinto en la versión 3.10: Se agregó el parámetro *key*.

bisect.**insort_left**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)
Inserte *x* en *a* en orden ordenado.

Esta función primero ejecuta *bisect_left()* para localizar un punto de inserción. A continuación, ejecuta el método `insert()` en *a* para insertar *x* en la posición adecuada para mantener el orden de clasificación.

To support inserting records in a table, the *key* function (if any) is applied to *x* for the search step but not for the insertion step.

Tenga en cuenta que la búsqueda `O(log n)` está dominada por el lento paso de inserción O (n).

Distinto en la versión 3.10: Se agregó el parámetro *key*.

bisect.**insort_right**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)
bisect.**insort**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)
Similar a *insort_left()*, pero inserta *x* en *a* después de cualquier entrada *x* existente.

Esta función primero ejecuta *bisect_right()* para localizar un punto de inserción. A continuación, ejecuta el método `insert()` en *a* para insertar *x* en la posición adecuada para mantener el orden de clasificación.

To support inserting records in a table, the *key* function (if any) is applied to *x* for the search step but not for the insertion step.

Tenga en cuenta que la búsqueda `O(log n)` está dominada por el lento paso de inserción O (n).

Distinto en la versión 3.10: Se agregó el parámetro *key*.

8.7.1 Notas de rendimiento

Al escribir código sensible al tiempo usando *bisect()* y *insort()*, tenga en cuenta estos pensamientos:

- La bisección es eficaz para buscar rangos de valores. Para localizar valores específicos, los diccionarios son más eficaces.
- Las funciones *insort()* son `O(n)` porque el paso de búsqueda logarítmica está dominado por el paso de inserción de tiempo lineal.
- Las funciones de búsqueda no tienen estado y descartan los resultados de las funciones clave después de su uso. En consecuencia, si las funciones de búsqueda se utilizan en un bucle, la función clave se puede llamar una y otra vez en los mismos elementos del arreglo. Si la función clave no es rápida, considere envolverla con *`functools.cache()`* para evitar cálculos duplicados. Alternativamente, considere buscar un arreglo de claves precalculadas para ubicar el punto de inserción (como se muestra en la sección de ejemplos a continuación).

Ver también:

- Sorted Collections is a high performance module that uses *bisect* to managed sorted collections of data.
- El SortedCollection recipe usa bisect para construir una clase de colección con todas las funciones con métodos de búsqueda sencillos y soporte para una función clave. Las teclas se calculan previamente para ahorrar llamadas innecesarias a la función de la tecla durante las búsquedas.

8.7.2 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    if i != len(a):
        return a[i]
    raise ValueError
```

8.7.3 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']
```

The *bisect()* and *insort()* functions also work with lists of tuples. The *key* argument can serve to extract the field used for ordering records in a table:

```
>>> from collections import namedtuple
>>> from operator import attrgetter
>>> from bisect import bisect, insort
>>> from pprint import pprint

>>> Movie = namedtuple('Movie', ('name', 'released', 'director'))

>>> movies = [
...     Movie('Jaws', 1975, 'Speilberg'),
...     Movie('Titanic', 1997, 'Cameron'),
...     Movie('The Birds', 1963, 'Hitchcock'),
...     Movie('Aliens', 1986, 'Scott')
... ]

>>> # Find the first movie released after 1960
>>> by_year = attrgetter('released')
>>> movies.sort(key=by_year)
>>> movies[bisect(movies, 1960, key=by_year)]
Movie(name='The Birds', released=1963, director='Hitchcock')

>>> # Insert a movie while maintaining sort order
>>> romance = Movie('Love Story', 1970, 'Hiller')
>>> insort(movies, romance, key=by_year)
>>> pprint(movies)
[Movie(name='The Birds', released=1963, director='Hitchcock'),
 Movie(name='Love Story', released=1970, director='Hiller'),
 Movie(name='Jaws', released=1975, director='Speilberg'),
 Movie(name='Aliens', released=1986, director='Scott'),
 Movie(name='Titanic', released=1997, director='Cameron')]
```

If the key function is expensive, it is possible to avoid repeated function calls by searching a list of precomputed keys to find the index of a record:

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])       # Or use operator.itemgetter(1).
>>> keys = [r[1] for r in data]         # Precompute a list of keys.
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
```

(continué en la próxima página)

(proviene de la página anterior)

```
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.8 `array` — Arreglos eficientes de valores numéricos

Este modulo 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'`	wchar_t	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) Puede ser de 16 bits o 32 bits según la plataforma.

Distinto en la versión 3.9: `array('u')` now uses `wchar_t` as C type instead of deprecated `Py_UNICODE`. This change doesn't affect its behavior because `Py_UNICODE` is alias of `wchar_t` since Python 3.3.

Obsoleto desde la versión 3.3, se eliminará en la versión 4.0.

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the *`array.itemsize`* attribute.

The module defines the following item:

`array.`**`typecodes`**
Una cadena de caracteres con todos los códigos de tipos disponible.

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()`*.

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.

Lanza un *evento de auditoría* `array.__new__` con argumentos `typecode`, `initializer`.

typecode
El carácter typecode utilizado para crear el arreglo.

itemsize
La longitud en bytes de un elemento del arreglo en su representación interna.

append(*x*)
Añade un nuevo elemento con valor *x* al final del arreglo.

buffer_info()
Return a tuple `(address, length)` giving the current memory address and the length in elements of the buffer used to hold array's contents. The size of the memory buffer in bytes can be computed as `array.buffer_info()[1] * array.itemsize`. This is occasionally useful when working with low-level (and inherently unsafe) I/O interfaces that require memory addresses, such as certain `ioctl()` operations. The returned numbers are valid as long as the array exists and no length-changing operations are applied to it.

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.

---

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.

count(*x*)
Retorna el número de ocurrencias de *x* en el arreglo.

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.

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()` is renamed to *frombytes()* for clarity.

fromfile(*f*, *n*)
Lee *n* elementos (como valores de máquina) del *file object* *f* y los añade al final del arreglo. Si hay menos de *n* elementos disponibles, se lanza *EOFError*, pero los elementos que estaban disponibles todavía se insertan en el arreglo.

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.

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.

index(*x*[, *start*[, *stop*]])
Retorna el *i* más pequeño de modo que *i* es el índice de la primera aparición de *x* en el arreglo. Los argumentos opcionales *start* y *stop* pueden ser especificados para buscar *x* dentro de una subsección del arreglo. Lanza *ValueError* si no se encuentra *x*.

Distinto en la versión 3.10: Se agregaron parámetros opcionales *start* y *stop*.

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.

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.

remove(*x*)
Elimina la primera ocurrencia de *x* del arreglo.

reverse()
Invierte el orden de los elementos en el arreglo.

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()` is renamed to *tobytes()* for clarity.

tofile(*f*)
Escribe todos los elementos (incluido elementos máquina) a el *file object* *f*.

tolist()
Convierte el arreglo a una lista ordinaria con los mismos elementos.

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.

NumPy El paquete NumPy define otro tipo de arreglo.

8.9 weakref — Referencias débiles

Código Fuente: Lib/weakref.py

El módulo *weakref* le permite al programador de Python crear *referencias débiles*<*weak references*> a objetos.

En lo consecutivo, el término *referente* aludirá 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, frozensets, algunos *objetos de archivo*, *generadores*, objetos de tipos, sockets, arreglos, deques, 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
```

Detalles de implementación de CPython: 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.

Cuando `__slots__``es definido para un tipo específico, el soporte para referencia débil es deshabilitado a menos que una cadena ``'__weakref__'` también esté presente en la secuencia de cadenas en la declaración `__slots__`. Véase __slots__ documentation para más detalles.

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, la retrollamada (*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 la retrollamada 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 *object* es mapeable. Ellos mantendrán su valor del hash incluso cuando el *objet* 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()`.

Accessing an attribute of the proxy object after the referent is garbage collected raises `ReferenceError`.

Distinto en la versión 3.8: Se extendió el soporto 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.

Note that when a key with equal value to an existing key (but not equal identity) is inserted into the dictionary, it replaces the value but does not replace the existing key. Due to this, when the reference to the original key is deleted, it also deletes the entry in the dictionary:

```
>>> class T(str): pass
...
>>> k1, k2 = T(), T()
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1   # d = {k1: 1}
>>> d[k2] = 2   # d = {k1: 2}
>>> del k1      # d = {}
```

A workaround would be to remove the key prior to reassignment:

```
>>> class T(str): pass
...
>>> k1, k2 = T(), T()
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1   # d = {k1: 1}
>>> del d[k1]
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> d[k2] = 2    # d = {k2: 2}
>>> del k1       # d = {k2: 2}
```

Distinto en la versión 3.9: Se agregó soporte para los operadores `|` y `|=`, especificados en **PEP 584**.

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.

Distinto en la versión 3.9: Se agregó soporte para los operadores `|` y `|=`, como se especifica en **PEP 584**.

Los objetos *WeakValueDictionary* tienen un método adicional que tiene los mismos problemas que el método `keyrefs()` de los objetos `WeakyKeyDictionary`.

`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()()
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.

8.9.1 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 «vivacidad» 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():
            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
```

8.9.2 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]
```

8.9.3 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
>>> 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
```

8.9.4 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 descargado:

```
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 daemonic *atexit.register()*, `try: ... finally: ...` y `with: ...` no garantizan que la limpieza ocurra tampoco.

8.10 `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.

8.10.1 Creación dinámica de tipos

`types.`**`new_class`**(*name*, *bases=()*, *kwds=None*, *exec_body=None*)

Crea un objeto de clase dinámicamente utilizando la metaclase 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`).

El argumento *exec_body* es una retrollamada que se usa para rellenar el espacio de nombres de clase recién creado. Debe aceptar el espacio de nombre de clase como su único argumento y actualizar el espacio de nombres directamente con el contenido de la clase. Si no se proporciona ninguna retrollamada, tiene el mismo efecto que pasar `lambda ns: None`.

Nuevo en la versión 3.3.

`types.`**`prepare_class`**(*name*, *bases=()*, *kwds=None*)

Calcula la metaclase 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 metaclase 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 metaclase no tiene un método `__prepare__`.

Ver también:

metaclasses Detalles completos del proceso de creación de clases soportado por estas funciones

PEP 3115 - Metaclases 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 desempaquetado 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

8.10.2 Tipos de Intérpretes 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.`**`NoneType`**
El tipo de *None*.

Nuevo en la versión 3.10.

`types.`**`FunctionType`**
`types.`**`LambdaType`**
El tipo de funciones definidas por el usuario y funciones creadas por expresiones `lambda`.

Lanza un *auditing event* `function.__new__` con el argumento `code`.

El evento auditor solo ocurre para la instanciación directa de objetos de código y no se genera para la compilación normal.

`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`.

Tenga en cuenta que los argumentos auditados pueden no coincidir con los nombres o posiciones requeridos por el inicializador. El evento auditor solo ocurre para la instanciación directa de objetos de código y no se genera para la compilación normal.

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.**NotImplementedType**
El tipo de *NotImplemented*.

Nuevo en la versión 3.10.

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*)
El tipo de *módulos*. El constructor toma el nombre del módulo que se va a crear y de forma opcional su *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 `__spec__` object.

Nota: Una futura versión de Python puede dejar de establecer este atributo por defecto. Para cuidarse de este cambio potencial, lea preferiblemente del atributo `__spec__` en su lugar o use `getattr(module, "__loader__", None)` si explícitamente necesita usar este atributo.

Distinto en la versión 3.4: El valor predeterminado es `None`. Anteriormente el atributo era opcional.

__name__
El nombre del módulo. Se espera que coincida con *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 `__spec__` object.

Nota: Una futura versión de Python puede dejar de establecer este atributo por defecto. Para cuidarse de este cambio potencial, lea preferiblemente del atributo `__spec__` en su lugar o use `getattr(module, "__package__", None)` si explícitamente necesita usar este atributo.

Distinto en la versión 3.4: El valor predeterminado es `None`. Anteriormente el atributo era opcional.

__spec__
Un registro del estado relacionado con el sistema de importación del módulo. Se espera que sea una instancia de *importlib.machinery.ModuleSpec*.

Nuevo en la versión 3.4.

`types.`**EllipsisType**
El tipo de *Ellipsis*.

Nuevo en la versión 3.10.

class `types.`**GenericAlias**(*t_origin*, *t_args*)
El tipo de *parameterized generics* como `list[int]`.

`t_origin` debería ser una clase genérica no parametrizada, como `list`, `tuple` o `dict`. `t_args` debe ser una *tuple* (posiblemente de longitud 1) de tipos que parametriza `t_origin`:

```
>>> from types import GenericAlias

>>> list[int] == GenericAlias(list, (int,))
True
>>> dict[str, int] == GenericAlias(dict, (str, int))
True
```

Nuevo en la versión 3.9.

Distinto en la versión 3.9.2: Este tipo ahora puede heredarse.

class `types.`**UnionType**
El tipo de *union type expressions*.

Nuevo en la versión 3.10.

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.days`. 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.

Detalles de implementación de CPython: 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.

Distinto en la versión 3.9: Actualizado para soportar el nuevo operador de unión (`|`) de **PEP 584**, que simplemente delega al mapeo subyacente.

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.

reversed(proxy)
Retorna un iterador inverso sobre las claves de la asignación subyacente.

Nuevo en la versión 3.9.

8.10.3 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):
        items = (f"{k}={v!r}" for k, v in self.__dict__.items())
        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.

Distinto en la versión 3.9: El orden de los atributos en el *repr* cambió de alfabético a orden de inserción (como dict).

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 (véase *enum.Enum* para obtener un ejemplo).

Nuevo en la versión 3.4.

8.10.4 Funciones de utilidad de corutina

types.**coroutine**(*gen_func*)

This function transforms a *generator* function into a *coroutine function* which returns a generator-based coroutine. The generator-based coroutine is still a *generator iterator*, but is also considered to be a *coroutine* object and is *awaitable*. However, it may not necessarily implement the `__await__()` method.

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.

8.11 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.

This module does not copy types like module, method, stack trace, stack frame, file, socket, window, or any similar types. It does «copy» functions and classes (shallow and deeply), by returning the original object unchanged; this is compatible with the way these are treated by the *pickle* module.

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 le pasa un argumento, el diccionario `memo`. Si la implementación de `__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. El diccionario de notas debe tratarse como un objeto opaco.

Ver también:

Módulo ***pickle*** Discusión de los métodos especiales utilizados para apoyar la recuperación y restauración del estado del objeto.

8.12 `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.

Distinto en la versión 3.9: Soporte añadido para imprimir de forma bonita *types.SimpleNamespace*.

Distinto en la versión 3.10: Soporte añadido para imprimir de forma bonita *dataclasses.dataclass*.

El módulo *pprint* solo define una clase:

class pprint.**PrettyPrinter**(*indent=1*, *width=80*, *depth=None*, *stream=None*, ***, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

Constructor de la instancia *PrettyPrinter*. Este constructor interpreta varios parámetros.

stream (por defecto `sys.stdout`) es un *file-like object* el cual la salida va a ser escrita usando el método `write()`.

Otros valores configuran la manera en que el anidamiento de estructuras datos complejos son visualizados.

indent (por defecto 1) especifica la cantidad de sangría agregada para cada nivel de anidamiento.

depth controla el número de niveles de anidamientos que podría ser impreso; si la estructura de datos a imprimir es muy profunda, el siguiente nivel es reemplazado por `...`. Por defecto, no hay ninguna restricción en la profundidad de los objetos que se formatean.

width (por defecto 80) especifica el número máximo deseado de caracteres por línea en la salida. Si no se puede formatear una estructura dentro de la restricción de ancho, se hará el mejor esfuerzo.

compact impacta la forma en que secuencias largas (listas, tuplas, conjuntos, etc) son formateadas. Si *compact* es falso (el valor predeterminado) entonces cada elemento de una secuencia será formateada en una línea separada. Si *compact* es verdadero, todos los elementos que encajen en el *width* de cada línea de salida, lo harán.

Si *sort_dicts* es verdadero (el valor predeterminado), los diccionarios se formatearán con sus claves ordenadas; de lo contrario, se mostrarán en orden de inserción.

Si *underscore_numbers* es verdadero, los enteros se formatearán con el carácter _ para un separador de miles; de lo contrario, no se mostrarán los guiones bajos (el valor predeterminado).

Distinto en la versión 3.4: Añadido el argumento *compact*.

Distinto en la versión 3.8: Añadido el argumento *sort_dicts*.

Distinto en la versión 3.10: Se agregó el parámetro *underscore_numbers*.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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*, *underscore_numbers=False*)

Retorna la representación formateada de *object* como una cadena. *indent*, *width*, *depth*, *compact*, *sort_dicts* y *underscore_numbers* se pasarán al constructor *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*.

Distinto en la versión 3.10: Se agregó el parámetro *underscore_numbers*.

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*, *underscore_numbers=False*)

Imprime la representación formateada de *object* en *stream*, seguida de una nueva línea. Si *stream* es `None`, se utiliza `sys.stdout`. Esto se puede usar en el intérprete interactivo en lugar de la función *print()* para inspeccionar valores (incluso puede reasignar `print = pprint.pprint` para usarlo dentro de un alcance). *indent*, *width*, *depth*, *compact*, *sort_dicts* y *underscore_numbers* se pasarán al constructor *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*.

Distinto en la versión 3.10: Se agregó el parámetro *underscore_numbers*.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

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
↪']"
```

8.12.1 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.

8.12.2 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,
 '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/',
```

(continué en la próxima página)

(proviene de la página anterior)

```
 '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'
```

(continué en la próxima página)

(proviene de la página anterior)

```
                '\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'}
```

8.13 `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.

8.13.1 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)`.

8.13.2 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>'
```

8.14 `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: Caso de miembros de Enum

Debido a que las enumeraciones se usan para representar constantes, recomendamos usar nombres UPPER_CASE para los miembros de enumeración, y usaremos ese estilo en nuestros ejemplos.

8.14.1 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`

8.14.2 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 hasheables, 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
```

8.14.3 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'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> member.value
1
```

8.14.4 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.

8.14.5 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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

8.14.6 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: El objetivo del método predeterminado `_generate_next_value_()` es proporcionar el siguiente *int* en secuencia con el último *int* proporcionado, pero la forma en que lo hace es un detalle de implementación y puede cambiar.

Nota: El método `_generate_next_value_()` debe definirse antes que cualquier miembro.

8.14.7 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>)
```

(continué en la próxima página)

(proviene de la página anterior)

```
('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']
```

8.14.8 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
```

8.14.9 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
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     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__()` o `__init__()`, los valores que se hayan dado al miembro enum se pasarán a esos métodos. Ver *Planet* para un ejemplo.

8.14.10 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: MoreColor: cannot extend enumeration 'Color'
```

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.)

8.14.11 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.

8.14.12 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 *fuente* de los nombres de los miembros de la enumeración. Puede se 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 parámetros `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 numero 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 sí solo se pasan nombres.

Distinto en la versión 3.5: Se agregó el parámetro *start*.

8.14.13 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     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):
...     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, por ejemplo, *int* de arriba. Esta restricción no se aplica a las mezclas que solo agregan métodos y no especifican otro tipo.
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. Cadenas de caracteres literales formateadas, *str.format()*, y *format()* usará el tipo mixto `__format__()` a menos que `__str__()` o `__format__()` se sobreescriba en la subclase, en cuyo caso se utilizarán los métodos anulados o *Enum*. Use los códigos de formato !s y !r para forzar el uso de los métodos *Enum* de las clases `__str__()` y `__repr__()`.

8.14.14 Cuándo usar `__new__()` contra `__init__()`

`__new__()` debe usarse siempre que desee personalizar el valor del miembro real `Emum`. 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.
...     """
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     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
```

8.14.15 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>
>>> Color.GREEN.value
2
```

Para hacer un `AutoNumber` de propósito más general, agregue `*args` a la firma:

```
>>> 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
...
```

Luego, cuando hereda de `AutoNumber`, puede escribir su propio `__init__` para manejar cualquier argumento adicional:

```
>>> 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>
>>> 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
...     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)
...
>>> 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)>]
```

8.14.16 ¿Cómo son diferentes las Enums?

Los Enums tienen una metaclase 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 otro 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 lanzará 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.

_Private__names

Private names will be normal attributes in Python 3.11 instead of either an error or a member (depending on if the name ends with an underscore). Using these names in 3.10 will issue a *DeprecationWarning*.

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
```

Nota: This behavior is deprecated and will be removed in 3.11.

Distinto en la versión 3.5.

Valor booleano de las clases y miembros `Enum`

Lo 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__', 'mass', 'name', 'radius', '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>
```

Nota: In 3.11 unnamed combinations of flags will only produce the canonical flag members (aka single-value flags). So `Color(7)` will produce something like `<Color.BLUE|GREEN|RED: 7>`.

8.15 graphlib —Funcionalidad para operar con estructuras de tipo-grafo

Código fuente: Lib/graphlib.py

---

class graphlib.**TopologicalSorter**(*graph=None*)

Provides functionality to topologically sort a graph of *hashable* nodes.

Un ordenamiento topológico es un ordenamiento lineal de los vértices en un grafo de modo que para cada arista dirigida u -> v desde el vértice u al vértice v, el vértice u viene antes del vértice v en el ordenamiento. Por ejemplo, los vértices del grafo pueden representar tareas a realizar y las aristas pueden representar restricciones de que una tarea debe realizarse antes que otra; en este ejemplo, un ordenamiento topológico es solo una secuencia válida para las tareas. Es posible un ordenamiento topológico completo si y solo si el grafo no tiene ciclos dirigidos, es decir, si es un grafo acíclico dirigido.

Si se proporciona el argumento opcional *graph*, este debe ser un diccionario que represente un grafo acíclico dirigido donde las claves son nodos y los valores son iterables de todos los predecesores de ese nodo en el grafo (los nodos que tienen las aristas que apuntan al valor clave). Se pueden agregar nodos adicionales al grafo utilizando el *add()* method.

En el caso general, los pasos necesarios para realizar el ordenamiento de un grafo son los siguientes:

- Cree una instancia *TopologicalSorter* con un grafo inicial opcional.
- Agregue nodos adicionales al grafo.
- Llame a *prepare()* en el grafo.
- Mientras *is_active()* is `True`, itere sobre los nodos retornados por *get_ready()* y procéselos . Llame a *done()* en cada nodo a medida que finaliza el procesamiento.

En caso de que sólo se requiera una ordenación inmediata de los nodos del grafo y no haya paralelismo, se puede utilizar directamente el método de conveniencia *TopologicalSorter.static_order()*

```
>>> graph = {"D": {"B", "C"}, "C": {"A"}, "B": {"A"}}
>>> ts = TopologicalSorter(graph)
>>> tuple(ts.static_order())
('A', 'C', 'B', 'D')
```

La clase está diseñada para soportar fácilmente el procesamiento en paralelo de los nodos a medida que estén listos. Para la instancia:

```
topological_sorter = TopologicalSorter()

# Add nodes to 'topological_sorter'...

topological_sorter.prepare()
while topological_sorter.is_active():
    for node in topological_sorter.get_ready():
        # Worker threads or processes take nodes to work on off the
        # 'task_queue' queue.
        task_queue.put(node)

    # When the work for a node is done, workers put the node in
    # 'finalized_tasks_queue' so we can get more nodes to work on.
    # The definition of 'is_active()' guarantees that, at this point, at
    # least one node has been placed on 'task_queue' that hasn't yet
    # been passed to 'done()', so this blocking 'get()' must (eventually)
    # succeed.  After calling 'done()', we loop back to call 'get_ready()'
    # again, so put newly freed nodes on 'task_queue' as soon as
    # logically possible.
```

(continué en la próxima página)

(proviene de la página anterior)

```
    node = finalized_tasks_queue.get()
    topological_sorter.done(node)
```

add(*node*, **predecessors*)
Add a new node and its predecessors to the graph. Both the *node* and all elements in *predecessors* must be *hashable*.

Si se llama varias veces con el mismo argumento del nodo, el conjunto de dependencias será la unión de todas las dependencias pasadas.

Es posible añadir un nodo sin dependencias (no se proporciona *predecessors*) o proporcionar una dependencia dos veces. Si un nodo que no se ha proporcionado antes se incluye entre los *predecessors*, se añadirá automáticamente al grafo sin predecesores propios.

Provoca *ValueError* si se llama después de *prepare()*.

prepare()
Marca el grafo como terminado y comprueba si existen ciclos en el grafo. Si se detecta algún ciclo, se lanzará *CycleError*, pero se puede seguir utilizando *get_ready()* para obtener tantos nodos como sea posible hasta que los ciclos bloqueen más el progreso. Después de una llamada a esta función, el grafo no puede ser modificado, y por lo tanto no se pueden añadir más nodos utilizando *add()*.

is_active()
Retorna `True` si se puede avanzar más y `False` en caso contrario. Se puede avanzar si los ciclos no bloquean la resolución y, o bien todavía existen nodos listos que aún no han sido retornados por *TopologicalSorter.get_ready()* o el número de nodos marcados con *TopologicalSorter.done()* es menor que el número que han sido retornados por *TopologicalSorter.get_ready()*.

El método `__bool__()` de esta clase defiere a esta función, por lo que en lugar de:

```
if ts.is_active():
    ...
```

es posible hacer simplemente:

```
if ts:
    ...
```

Lanzar *ValueError* si se llama sin llamar previamente a *prepare()*.

done(**nodes*)
Marca un conjunto de nodos retornados por *TopologicalSorter.get_ready()* como procesados, desbloqueando cualquier sucesor de cada nodo en *nodes* para ser retornado en el futuro por una llamada a *TopologicalSorter.get_ready()*.

Lanza *ValueError* si algún nodo de *nodes* ya ha sido marcado como procesado por una llamada anterior a este método o si un nodo no fue añadido al grafo usando *TopologicalSorter.add()*, si se llama sin llamar a *prepare()* o si el nodo aún no ha sido retornado por *get_ready()*.

get_ready()
Retorna una `tupla` con todos los nodos que están listos. Inicialmente retorna todos los nodos sin predecesores, y una vez que éstos se marcan como procesados llamando a *TopologicalSorter.done()*, las llamadas posteriores devolverán todos los nuevos nodos que tengan todos sus predecesores ya procesados. Una vez que no se puede avanzar más, se retornan tuplas vacías.

Lanzar *ValueError* si se llama sin llamar previamente a *prepare()*.

static_order()
Retorna un iterable de nodos en un ordenamiento topológico. El uso de este método no requiere llamar a *TopologicalSorter.prepare()* o *TopologicalSorter.done()*. Este método es equivalente a:

```
def static_order(self):
    self.prepare()
    while self.is_active():
        node_group = self.get_ready()
        yield from node_group
        self.done(*node_group)
```

El orden concreto que se retorna puede depender del orden específico en que se insertaron los elementos en el grafo. Por ejemplo:

```
>>> ts = TopologicalSorter()
>>> ts.add(3, 2, 1)
>>> ts.add(1, 0)
>>> print([*ts.static_order()])
[2, 0, 1, 3]

>>> ts2 = TopologicalSorter()
>>> ts2.add(1, 0)
>>> ts2.add(3, 2, 1)
>>> print([*ts2.static_order()])
[0, 2, 1, 3]
```

Esto se debe a que «0» y «2» están en el mismo nivel en el gráfico (habrían sido retornados en la misma llamada a *get_ready()*) y el orden entre ellos está determinado por el orden de inserción.

Si se detecta algún ciclo, se lanzará *CycleError*.

Nuevo en la versión 3.9.

8.15.1 Excepciones

El módulo *graphlib* define las siguientes clases de excepción:

exception graphlib.**CycleError**

Subclase de *ValueError* planteada por *TopologicalSorter.prepare()* si existen ciclos en el grafo de trabajo. Si existen múltiples ciclos, sólo se informará de una elección indefinida entre ellos y se incluirá en la excepción.

Se puede acceder al ciclo detectado a través del segundo elemento del atributo `args` de la instancia de la excepción y consiste en una lista de nodos, tal que cada nodo este, en el grafo, un predecesor inmediato del siguiente nodo en la lista. En la lista reportada, el primer y el último nodo serán el mismo, para dejar claro que es cíclico.

CAPÍTULO 9

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

El módulo *numbers* (**PEP 3141**) define una jerarquía de *abstract base classes* numérico que define progresivamente más operaciones. Ninguno de los tipos definidos en este módulo está destinado a ser instanciado.

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 La torre numérica

class numbers.**Complex**
Las subclases de este tipo describen números complejos e incluyen las operaciones que funcionan en el tipo *complex* integrado. Estos son: conversiones a *complex* y *bool*, *real*, *imag*, +, −, *, /, **, *abs()*, *conjugate()*, == y !=. Todos excepto − y != 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. It also provides a default for *float()*.

The *numerator* and *denominator* values should be instances of *Integral* and should be in lowest terms with *denominator* positive.

numerator
Abstracto.

denominator
Abstracto.

class numbers.**Integral**
Subtipos *Rational* y agrega una conversión a *int*. Proporciona valores predeterminados para *float()*, *numerator* y *denominator*. Agrega métodos abstractos para *pow()* con operaciones de módulo y cadena de bits: <<, >>, &, ^, |, ~.

9.1.2 Notas para implementadores de tipos

Los implementadores deben tener cuidado de igualar números iguales y aplicar un hash a los mismos valores. Esto puede ser sutil si hay dos extensiones diferentes de los números reales. Por ejemplo, *fractions.Fraction* implementa *hash()* de la siguiente manera:

```
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 esta 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 probara `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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
        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 porqué 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*)
Return the ceiling of *x*, the smallest integer greater than or equal to *x*. If *x* is not a float, delegates to `x.__ceil__`, which should return an *Integral* value.

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 <= 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.

Obsoleto desde la versión 3.9: Aceptar flotantes con valores integrales (como `5.0`) está obsoleto.

math.**floor**(*x*)
Return the floor of *x*, the largest integer less than or equal to *x*. If *x* is not a float, delegates to `x.__floor__`, which should return an *Integral* value.

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 <= 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**(**integers*)
Retorna el máximo común divisor de los argumentos enteros. Si cualquiera de los argumentos no es cero, entonces el valor retornado es el entero positivo más grande que divide a todos los argumentos. Si todos los argumentos son cero, entonces el valor retornado es `0`. `gcd()` sin argumentos retorna `0`.

Nuevo en la versión 3.5.

Distinto en la versión 3.9: Agregado soporte para un número arbitrario de argumentos. Anteriormente sólo se soportaba dos argumentos.

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.**lcm**(**integers*)
Retorna el mínimo común múltiplo de los argumentos enteros. Si todos los argumentos no son cero, entonces el valor retornado es el entero positivo más pequeño que es un múltiplo de todos los argumentos. Si cualquiera de los argumentos es cero, entonces el valor retornado es `0`. `lcm()` sin argumentos retorna `1`.

Nuevo en la versión 3.9.

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.**nextafter**(*x*, *y*)
Retorna el siguiente valor flotante después de *x* en la dirección de *y*.

Si *x* es igual a *y*, retorna *y*.

Ejemplos:

- `math.nextafter(x, math.inf)` va hacia el infinito positivo.
- `math.nextafter(x, -math.inf)` va hacia el infinito negativo.
- `math.nextafter(x, 0.0)` va hacia cero.
- `math.nextafter(x, math.copysign(math.inf, x))` se aleja de cero.

Ver también *math.ulp()*.

Nuevo en la versión 3.9.

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 <= 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 = remainder(x, y)` siempre satisface `abs(r) <= 0.5 * 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*)
Return *x* with the fractional part removed, leaving the integer part. This rounds toward 0: `trunc()` is equivalent to *floor()* for positive *x*, and equivalent to *ceil()* for negative *x*. If *x* is not a float, delegates to `x.__trunc__`, which should return an *Integral* value.

math.**ulp**(*x*)
Retorna el valor del bit menos significativo del flotante *x*:

- Si *x* es un NaN (*not a number*), retorna *x*.
- Si *x* es negativo, retorna `ulp(-x)`.
- Si *x* es un infinito positivo, retorna *x*.
- Si *x* es igual a cero, retorna el flotante representable *desnormalizado* positivo más pequeño (menor que el flotante *normalizado* positivo mínimo, *sys.float_info.min*).
- Si *x* es igual al flotante representable positivo más pequeño, retorna el bit menos significativo de *x*, de tal manera que el primer flotante menor que *x* es `x - ulp(x)`.
- De lo contrario (*x* es un número finito positivo), retorna el valor del bit menos significativo de *x* , de tal forma que el primer flotante mayor a *x* es `x + ulp(x)`.

ULP significa *Unit in the Last Place* (unidad en el último lugar).

Ver también *math.nextafter()* y *sys.float_info.epsilon*.

Nuevo en la versión 3.9.

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 `abs(x) >= 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… 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(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. El resultado está entre `0` y `pi`.

math.**asin**(*x*)
Retorna el arcoseno de *x*, en radianes. El resultado está entre `-pi/2` y `pi/2`.

math.**atan**(*x*)
Retorna la arcotangente de *x*, en radianes. El resultado está entre `-pi/2` y `pi/2`.

math.**atan2**(*y*, *x*)
Retorna `atan(y / x)`, en radianes. El resultado está entre `-pi` y `pi`. 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.

Distinto en la versión 3.10: Se mejoró la precisión del algoritmo para que el error máximo sea inferior a 1 ulp (unidad en último lugar). Más típicamente, el resultado casi siempre se redondea correctamente dentro de 1/2 ulp.

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*.

The *erf()* function can be used to compute traditional statistical functions such as the cumulative standard normal distribution:

```
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 - erf(x)`. Se usa para valores grandes de *x* donde una resta de 1 causaría una pérdida de presición.

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 $\pi = 3.141592\ldots$, hasta la precisión disponible.

math.**e**
La constante matemática $e = 2.718281\ldots$, hasta la precisión disponible.

math.**tau**
La constante matemática $\tau = 6.283185\ldots$, 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**
A floating-point «not a number» (NaN) value. Equivalent to the output of `float('nan')`. Due to the requirements of the IEEE-754 standard, `math.nan` and `float('nan')` are not considered to equal to any other numeric value, including themselves. To check whether a number is a NaN, use the *isnan()* function to test for NaNs instead of `is` or ==. Example:

```
>>> import math
>>> math.nan == math.nan
False
>>> float('nan') == float('nan')
False
>>> math.isnan(math.nan)
True
>>> math.isnan(float('nan'))
True
```

Nuevo en la versión 3.5.

Detalles de implementación de CPython: 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 modulo 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: For functions involving branch cuts, we have the problem of deciding how to define those functions on the cut itself. Following Kahan's «Branch cuts for complex elementary functions» paper, as well as Annex G of C99 and later C standards, we use the sign of zero to distinguish one side of the branch cut from the other: for a branch cut along (a portion of) the real axis we look at the sign of the imaginary part, while for a branch cut along the imaginary axis we look at the sign of the real part.

For example, the *cmath.sqrt()* function has a branch cut along the negative real axis. An argument of `complex(-2.0, -0.0)` is treated as though it lies *below* the branch cut, and so gives a result on the negative imaginary axis:

```
>>> cmath.sqrt(complex(-2.0, -0.0))
-1.4142135623730951j
```

But an argument of `complex(-2.0, 0.0)` is treated as though it lies above the branch cut:

```
>>> cmath.sqrt(complex(-2.0, 0.0))
1.4142135623730951j
```

9.3.1 Conversión a y desde coordenadas polares

Un numero 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 linea 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)`

Return the phase of *x* (also known as the *argument* of *x*), as a float. `phase(x)` is equivalent to `math.atan2(x.imag, x.real)`. The result lies in the range [-π, π], and the branch cut for this operation lies along the negative real axis. The sign of the result is the same as the sign of `x.imag`, even when `x.imag` is zero:

```
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```

Nota: El módulo (valor absoluto) de un número complejo *x* puede ser calculado usado 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*])
Returns the logarithm of *x* to the given *base*. If the *base* is not specified, returns the natural logarithm of *x*. There is one branch cut, from 0 along the negative real axis to -∞.

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*)
Return the arc cosine of *x*. There are two branch cuts: One extends right from 1 along the real axis to ∞. The other extends left from -1 along the real axis to -∞.

cmath.**asin**(*x*)
Retorna el arcoseno de *x*. Este tiene los mismos tramos que *acos()*.

cmath.**atan**(*x*)
Return the arc tangent of *x*. There are two branch cuts: One extends from `1j` along the imaginary axis to ∞`j`. The other extends from `-1j` along the imaginary axis to −∞`j`.

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*)
Return the inverse hyperbolic cosine of *x*. There is one branch cut, extending left from 1 along the real axis to -∞.

cmath.**asinh**(*x*)
Return the inverse hyperbolic sine of *x*. There are two branch cuts: One extends from `1j` along the imaginary axis to ∞`j`. The other extends from `-1j` along the imaginary axis to −∞`j`.

cmath.**atanh**(*x*)
Return the inverse hyperbolic tangent of *x*. There are two branch cuts: One extends from `1` along the real axis to ∞. The other extends from `-1` along the real axis to −∞.

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 `Falso` 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 `Falso` 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` — Aritmética decimal de coma fija y coma flotante

Código fuente: 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 «se basa en un modelo de coma flotante que se diseñó pensando en las personas, y necesariamente tiene un principio rector supremo: las computadoras deben proporcionar una aritmética que funcione de la misma manera que la aritmética que las personas aprenden en la escuela.» – extracto (traducido) de la especificación de la aritmética decimal.
- 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`.
- A diferencia del punto flotante binario basado en hardware, el módulo decimal tiene una precisión modificable por el usuario (por defecto es de 28 dígitos decimales) que puede ser tan grande como sea necesario para un problema dado:

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- Tanto la representación en coma flotante binaria como la decimal se implementan de acuerdo a estándares publicados. Mientras que el tipo float expone solo una pequeña parte de sus capacidades, el módulo decimal expone todos los componentes requeridos del estándar. Cuando es necesario, el desarrollador tiene control total sobre el redondeo y la gestión de las señales. Esto incluye la capacidad de forzar la aritmética exacta, utilizando excepciones para bloquear cualquier operación inexacta.
- El módulo decimal fue diseñado para admitir «indiscriminadamente, tanto aritmética decimal exacta sin redondeo (a veces llamada aritmética de coma fija) como la aritmética de coma flotante con redondeo.» – extracto (traducido) de la especificación de la aritmética decimal.

El módulo está diseñado en torno a tres conceptos: el número decimal, el contexto aritmético y las señales.

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`.

El contexto aritmético es un entorno que permite especificar una precisión, reglas de redondeo, límites en los exponentes, flags que indican el resultado de las operaciones y habilitadores de trampas que especifican si las señales (reportadas durante operaciones ilegales) son tratadas o no como excepciones de Python. Las opciones de redondeo incluyen *ROUND_CEILING*, *ROUND_DOWN*, *ROUND_FLOOR*, *ROUND_HALF_DOWN*, *ROUND_HALF_EVEN*, *ROUND_HALF_UP*, *ROUND_UP* y *ROUND_05UP*.

Las señales son grupos de condiciones excepcionales que ocurren durante el cálculo. Dependiendo de las necesidades de la aplicación, las señales pueden ignorarse, tratarse como información o tratarse como excepciones. Las señales existentes en el módulo decimal son *Clamped*, *InvalidOperation*, *DivisionByZero*, *Inexact*, *Rounded*, *Subnormal*, *Overflow*, *Underflow* y *FloatOperation*.

Por cada señal hay un flag y un habilitador de trampa. Cuando ocurre una operación ilegal, su flag se establece en uno, luego, si su habilitador de trampa está establecido en uno, se lanza una excepción. La configuración de los flags es persistente, por lo que el usuario debe restablecerlos antes de comenzar un cálculo que desee monitorear.

Ver también:

- IBM's General Decimal Arithmetic Specification, The General Decimal Arithmetic Specification.

9.4.1 Tutorial de inicio rápido

El punto de partida habitual para usar decimales es importar el módulo, ver el contexto actual con *getcontext()* y, si es necesario, establecer nuevos valores para la precisión, el redondeo o trampas de señales habilitadas:

```
>>> 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('1.4142135623730950488016887242')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

Si la señal *FloatOperation* es atrapada, la mezcla accidental de decimales y flotantes en constructores o comparaciones de orden lanzará una excepción:

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

Nuevo en la versión 3.3.

La significación de un nuevo objeto Decimal es determinada únicamente por el número de dígitos ingresados. La precisión y el redondeo establecidos en el contexto solo entran en juego durante las operaciones aritméticas.

```
>>> 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')
```

Se lanza una excepción *InvalidOperation* si durante la construcción de un objeto Decimal se exceden los límites internos de la versión de C:

```
>>> Decimal("1e9999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

Distinto en la versión 3.3.

Los objetos Decimal interactúan bien con gran parte del resto de Python. Aquí hay un pequeño circo volador de punto flotante decimal:

```
>>> 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)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

Y algunas funciones matemáticas también están disponibles para 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')
```

Como se muestra arriba, la función *getcontext()* accede al contexto actual y permite cambiar la configuración. Este enfoque satisface las necesidades de la mayoría de las aplicaciones.

Para trabajos más avanzados, puede resultar útil crear contextos alternativos utilizando el constructor Context(). Para activar un contexto alternativo, usa la función *setcontext()*.

De acuerdo con el estándar, el módulo *decimal* proporciona dos contextos estándar listos para usar, *BasicContext* y *ExtendedContext*. El primero es particularmente útil para la depuración, ya que muchas de las trampas de señales están habilitadas por defecto:

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 pi was rounded (digits beyond the context precision were thrown away) and that the result is inexact (some of the discarded digits were non-zero).

Individual traps are set using the dictionary in the `traps` attribute 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
```

La mayoría de los programas ajustan el contexto actual una sola vez, al comienzo del programa. Y, en muchas aplicaciones, los datos se convierten a *Decimal* mediante una única conversión dentro de un bucle. Con el contexto establecido y los decimales creados, la mayor parte del programa manipula los datos de la misma forma que con otros tipos numéricos de Python.

9.4.2 Objetos Decimal

class decimal.**Decimal**(*value='0'*, *context=None*)

Construye un nuevo objeto *Decimal* basado en *value*.

value puede ser un entero, una cadena de caracteres, una tupla, un *float* u otro objeto *Decimal*. Si no se proporciona *value*, retorna `Decimal('0')`. Si *value* es una cadena, debe ajustarse a la sintaxis de cadena numérica decimal después de que los espacios en blanco iniciales y finales, así como los guiones bajos, sean eliminados:

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

(continué en la próxima página)

(proviene de la página anterior)

```
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
```

También se permiten otros dígitos decimales Unicode en aquellos lugares en los que arriba aparece `digit`. Estos incluyen dígitos decimales de otros alfabetos (por ejemplo, dígitos del alfabeto árabe-índico y devanāgarī) junto con los dígitos de ancho completo desde `'\uff10'` a `'\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')`.

Si *value* es un *float*, el valor binario de coma flotante se convierte sin pérdidas a su equivalente decimal exacto. Esta conversión a menudo puede requerir 53 o más dígitos de precisión. Por ejemplo, `Decimal(float('1.1'))` se convierte en `Decimal('1.1000000000000000888178419700125232338905334472656 25')`.

La precisión de *context* no afecta a la cantidad de dígitos almacenados. Eso está determinado exclusivamente por el número de dígitos en *value*. Por ejemplo, `Decimal('3.00000')` registra los cinco ceros incluso si la precisión del contexto es solo tres.

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`.

Una vez construidos, los objetos *Decimal* son inmutables.

Distinto en la versión 3.2: Ahora se permite que el argumento del constructor sea una instancia *float*.

Distinto en la versión 3.3: Los argumentos *float* ahora generan una excepción si se establece la trampa *FloatOperation*. Por defecto, la trampa está desactivada.

Distinto en la versión 3.6: Se permiten guiones bajos para la agrupación, como ocurre en el código con los literales enteros y de punto flotante.

Los objetos de coma flotante decimal comparten muchas propiedades con los otros tipos numéricos integrados, como *float* e *int*. Se aplican todas las operaciones matemáticas habituales y los métodos especiales. Asimismo, los objetos decimales se pueden copiar, serializar con pickle, imprimir, usar como claves de un diccionario o como elementos de un conjunto, comparar, ordenar y convertir a otros tipos (como *float* o *int*).

Hay algunas pequeñas diferencias entre la aritmética en objetos decimales y la aritmética en enteros y flotantes. Cuando el operador de resto `%` se aplica a objetos Decimal, el signo del resultado es el signo del *dividendo* en lugar del signo del divisor:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

El operador de división entera `//` se comporta de manera análoga, retornando la parte entera del cociente verdadero (truncando hacia cero) en lugar del resultado de aplicarle la función suelo. Esto se hace con la finalidad de preservar la identidad habitual `x == (x // y) * y + x % y`:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

Los operadores `%` y `//` implementan las operaciones `remainder` y `divide-integer` (respectivamente) como se describe en la especificación.

Los objetos de la clase Decimal generalmente no se pueden combinar con flotantes o instancias de *`fractions.Fraction`* en operaciones aritméticas: un intento de agregar un objeto *`Decimal`* a un *`float`*, por ejemplo, lanzará una excepción *`TypeError`*. Sin embargo, es posible usar los operadores de comparación de Python para comparar una instancia de *`Decimal`* `x` con otro número `y`. Esto evita resultados confusos al hacer comparaciones de igualdad entre números de diferentes tipos.

Distinto en la versión 3.2: Las comparaciones de tipo mixto entre instancias de *`Decimal`* y otros tipos numéricos ahora son totalmente compatibles.

Además de las propiedades numéricas estándar, los objetos de coma flotante decimal también tienen varios métodos especializados:

`adjusted``()`
Retorna el exponente ajustado después de desplazar los dígitos del extremo derecho del coeficiente hasta que solo quede el dígito principal: `Decimal('321e+5').adjusted()` retorna siete. Se utiliza para determinar la posición del dígito más significativo con respecto al punto decimal.

`as_integer_ratio``()`
Retorna un par de enteros `(n, d)` que representan la instancia de *`Decimal`* proporcionada como una fracción irreducible y con un denominador positivo:

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

La conversión es exacta. Lanza una excepción OverflowError si se proporcionan valores infinitos y ValueError con valores NaN.

Nuevo en la versión 3.6.

`as_tuple``()`
Retorna una representación en forma de *named tuple* del número: `DecimalTuple(sign, digits, exponent)`.

`canonical``()`
Retorna la codificación canónica del argumento. Actualmente, la codificación de una instancia de *`Decimal`* es siempre canónica, por lo que esta operación retorna su argumento sin cambios.

`compare`(*other*, *context=None*)
Compara los valores de dos instancias de Decimal. El método *`compare()`* retorna una instancia de Decimal, y si alguno de los operandos es un NaN, el resultado es un 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*)
Esta operación es idéntica al método *`compare()`*, excepto que todos los valores NaN generan una señal. Es decir, si ninguno de los operandos es un NaN señalizador, cualquier operando de NaN silencioso se trata como si fuera un NaN señalizador.

`compare_total`(*other*, *context=None*)
Compara dos operandos utilizando su representación abstracta en lugar de su valor numérico. Similar al método *`compare()`*, pero el resultado proporciona un ordenamiento total en las instancias de *`Decimal`*. Dos instancias de *`Decimal`* con el mismo valor numérico, pero diferentes representaciones, se comparan como desiguales usando este orden:

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

Los NaN silenciosos y señalizadores también se incluyen en el ordenamiento total. El resultado de esta función es `Decimal('0')` si ambos operandos tienen la misma representación, `Decimal('-1')` si el primer operando es menor en el orden total que el segundo y `Decimal('1')` si el primer operando es mayor en el orden total que el segundo operando. Consulta las especificaciones para obtener detalles sobre el ordenamiento total.

Esta operación no se ve afectada por el contexto y es silenciosa: no se cambian los flags y no se realiza ningún redondeo. Como excepción, la versión de C puede lanzar InvalidOperation si el segundo operando no se puede convertir exactamente.

compare_total_mag(*other*, *context=None*)

Compara dos operandos usando su representación abstracta en lugar de su valor, como en *compare_total()*, pero ignorando el signo de cada operando. `x.compare_total_mag(y)` es equivalente a `x.copy_abs().compare_total(y.copy_abs())`.

Esta operación no se ve afectada por el contexto y es silenciosa: no se cambian los flags y no se realiza ningún redondeo. Como excepción, la versión de C puede lanzar InvalidOperation si el segundo operando no se puede convertir exactamente.

conjugate()

Simplemente retorna self (el propio objeto al que pertenece el método invocado). Este método existe solo para cumplir con la Especificación decimal.

copy_abs()

Retorna el valor absoluto del argumento. Esta operación no se ve afectada por el contexto y es silenciosa: no se modifican los flags y no se realiza ningún redondeo.

copy_negate()

Retorna la negación del argumento. Esta operación no se ve afectada por el contexto y es silenciosa: no se cambian los flags y no se realiza ningún redondeo.

copy_sign(*other*, *context=None*)

Retorna una copia del primer operando pero con el signo establecido para que sea el mismo que el del segundo operando. Por ejemplo:

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

Esta operación no se ve afectada por el contexto y es silenciosa: no se cambian los flags y no se realiza ningún redondeo. Como excepción, la versión de C puede lanzar InvalidOperation si el segundo operando no se puede convertir exactamente.

exp(*context=None*)

Retorna el valor de la función exponencial (natural) `e**x` en el número dado. El resultado es correctamente redondeado utilizando el modo de redondeo *ROUND_HALF_EVEN*.

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

classmethod from_float(*f*)

Alternative constructor that only accepts instances of *float* or *int*.

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: Desde Python 3.2 en adelante, una instancia de *Decimal* también se puede construir directamente desde una instancia de *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*)
Fusión de la multiplicación y la suma. Retorna self*other+third sin redondeo del producto intermedio self*other.

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical()
Retorna *True* si el argumento es canónico y *False* en caso contrario. Actualmente, una instancia de *Decimal* es siempre canónica, por lo que esta operación siempre retorna *True*.

is_finite()
Retorna *True* si el argumento es un número finito y *False* si el argumento es un valor infinito o un NaN.

is_infinite()
Retorna *True* si el argumento es un valor infinito positivo o negativo y *False* en caso contrario.

is_nan()
Retorna *True* si el argumento es un NaN (silencioso o señalizador) y *False* en caso contrario.

is_normal(*context=None*)
Retorna *True* si el argumento es un número finito *normal*. Retorna *False* si el argumento es cero, subnormal, infinito o un NaN.

is_qnan()
Retorna *True* si el argumento es un NaN silencioso y *False* en caso contrario.

is_signed()
Retorna *True* si el argumento tiene signo negativo y *False* en caso contrario. Ten en cuenta que tanto los ceros como los NaN pueden tener signo.

is_snan()
Retorna *True* si el argumento es un NaN señalizador y *False* en caso contrario.

is_subnormal(*context=None*)
Retorna *True* si el argumento es subnormal y *False* en caso contrario.

is_zero()
Retorna *True* si el argumento es un cero (positivo o negativo) y *False* en caso contrario.

ln(*context=None*)
Retorna el logaritmo natural (base e) del operando. El resultado es correctamente redondeado utilizando el modo de redondeo *ROUND_HALF_EVEN*.

log10(*context=None*)
Retorna el logaritmo en base diez del operando. El resultado es correctamente redondeado utilizando el modo de redondeo *ROUND_HALF_EVEN*.

logb(*context=None*)
Para un número distinto de cero, retorna el exponente ajustado de su operando como una instancia de *Decimal*. Si el operando es cero, se retorna `Decimal('-Infinity')` y se activa el flag *DivisionByZero*. Si el operando es infinito, se retorna `Decimal('Infinity')`.

logical_and(*other*, *context=None*)
`logic_and()` es una operación lógica que toma dos *operandos lógicos* (consultar *Operandos lógicos*). El resultado es el `and` dígito por dígito de los dos operandos.

logical_invert(*context=None*)
`logic_invert()` es una operación lógica. El resultado es la inversión dígito a dígito del operando.

logical_or(*other*, *context=None*)
`logical_or()` es una operación lógica que toma dos *operandos lógicos* (consultar *Operandos lógicos*). El resultado es un `or` dígito a dígito de los dos operandos.

logical_xor(*other*, *context=None*)
`logic_xor()` es una operación lógica que toma dos *operandos lógicos* (consultar *Operandos lógicos*). El resultado es la disyunción exclusiva («exclusive or») dígito a dígito de ambos operandos.

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 al método *`max()`*, pero la comparación se realiza utilizando los valores absolutos de los operandos.

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 al método *`min()`*, pero la comparación se realiza utilizando los valores absolutos de los operandos.

next_minus(*context=None*)
Retorna el número más grande representable en el contexto proporcionado (o en el contexto del hilo actual si no se proporciona un contexto) que sea más pequeño que el operando proporcionado.

next_plus(*context=None*)
Retorna el número más pequeño representable en el contexto proporcionado (o en el contexto del hilo actual si no se proporciona ningún contexto) que sea más grande que el operando proporcionado.

next_toward(*other*, *context=None*)
Si los dos operandos no son iguales, retorna el número más cercano al primer operando en la dirección del segundo operando. Si ambos operandos son numéricamente iguales, retorna una copia del primer operando con el signo establecido para que sea el mismo que el signo del segundo operando.

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*)
Retorna una cadena de caracteres que describe la *class* del operando. El valor retornado es una de las siguientes diez cadenas de caracteres.

- `"-Infinity"`, que indica que el operando es un infinito negativo.
- `"-Normal"`, que indica que el operando es un número normal negativo.
- `"-Subnormal"`, que indica que el operando es negativo y subnormal.
- `"-Zero"`, que indica que el operando es un cero negativo.
- `"+Zero"`, que indica que el operando es un cero positivo.
- `"+Subnormal"`,que indica que el operando es positivo y subnormal.

- `"+Normal"`, que indica que el operando es un número normal positivo.
- `"+Infinity"`, que indica que el operando es un infinito positivo.
- `"NaN"`, que indica que el operando es un NaN (no es un número) silencioso.
- `"sNaN"`, que indica que el operando es un NaN (no es un número) señalizador.

quantize(*exp*, *rounding=None*, *context=None*)

Retorna un valor igual al primer operando después de ser redondeado y de asignarle el exponente del segundo operando.

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

A diferencia de otras operaciones, se genera una señal *InvalidOperation* si la longitud del coeficiente después de la operación quantize es mayor que la precisión. Esto garantiza que, a menos que exista una condición de error, el exponente cuantificado sea siempre igual al del operando de la derecha.

Además, a diferencia de otras operaciones, quantize nunca genera una señal Underflow, incluso si el resultado es subnormal e inexacto.

Si el exponente del segundo operando es mayor que el del primero, puede ser necesario redondear. En este caso, el modo de redondeo está determinado por el argumento `rounding`, si se proporciona, o por el argumento `context` en caso contrario. Si no se proporciona ninguno de estos dos argumentos, se utiliza el modo de redondeo establecido en el contexto del hilo actual.

An error is returned whenever the resulting exponent is greater than `Emax` or less than *Etiny()*.

radix()

Retorna `Decimal(10)`, que es la raíz (base) en la que la clase *Decimal* hace toda su aritmética. Este método está incluido solo por compatibilidad con la especificación.

remainder_near(*other*, *context=None*)

Retorna el resto de dividir *self* entre *other*. Esto difiere de la operación `self%other`, en la que el signo del resto se elige para minimizar su valor absoluto. Más precisamente, el valor de retorno es `self - n * other`, donde `n` es el número entero más cercano al valor exacto de `self / other`. Si dos enteros están igualmente cerca, entonces el valor par es el elegido.

Si el resultado es cero, entonces su signo será el signo de *self*.

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*, *context=None*)

Retorna el resultado de rotar los dígitos del primer operando en una cantidad especificada por el segundo operando. El segundo operando debe ser un número entero en el rango comprendido desde -precisión hasta precisión. El valor absoluto del segundo operando da el número de lugares a rotar. Si el segundo operando es positivo, la rotación es hacia la izquierda; de lo contrario, la rotación es hacia la derecha. El coeficiente del primer operando se rellena con ceros a la izquierda para satisfacer la precisión de longitud si es necesario. El signo y el exponente del primer operando no se modifican.

same_quantum(*other*, *context=None*)

Test whether self and other have the same exponent or whether both are `NaN`.

Esta operación no se ve afectada por el contexto y es silenciosa: no se cambian los flags y no se realiza ningún redondeo. Como excepción, la versión de C puede lanzar InvalidOperation si el segundo operando no se puede convertir exactamente.

scaleb(*other*, *context=None*)

Retorna el primer operando con su exponente ajustado por el segundo. De manera equivalente, retorna el primer operando multiplicado por `10**other`. El segundo operando debe ser un número entero.

shift(*other*, *context=None*)
Retorna el resultado de cambiar los dígitos del primer operando en una cantidad especificada por el segundo operando. El segundo operando debe ser un número entero en el rango comprendido desde -precisión hasta precisión. El valor absoluto del segundo operando da el número de lugares a desplazar. Si el segundo operando es positivo, el desplazamiento es hacia la izquierda; de lo contrario, el desplazamiento es hacia la derecha. Los dígitos desplazados en el coeficiente son ceros. El signo y el exponente del primer operando no se modifican.

sqrt(*context=None*)
Retorna la raíz cuadrada del argumento con precisión total.

to_eng_string(*context=None*)
Convierte a una cadena de caracteres, usando notación de ingeniería si se necesita un exponente.

La notación de ingeniería tiene como exponente un múltiplo de 3. Esto puede dejar hasta 3 dígitos a la izquierda del punto decimal y puede requerir la adición de uno o dos ceros finales.

Por ejemplo, este método convierte `Decimal('123E+1')` en `Decimal('1.23E+3')`.

to_integral(*rounding=None*, *context=None*)
Idéntico al método *to_integral_value()*. El nombre `to_integral` se ha mantenido por compatibilidad con versiones anteriores.

to_integral_exact(*rounding=None*, *context=None*)
Redondea al entero más cercano, generando la señal *Inexact* o *Rounded*, según corresponda, si se produce un redondeo. El modo de redondeo está determinado por el parámetro `rounding` si es proporcionado, o por el establecido en el `context` proporcionado en caso contrario. Si no se proporciona ninguno de estos dos parámetros, se utiliza el modo de redondeo establecido en el contexto actual.

to_integral_value(*rounding=None*, *context=None*)
Redondea al entero más cercano sin generar la señal *Inexact* o *Rounded*. Si se proporciona, se aplica el método de redondeo especificado por *rounding*; en caso contrario, se utiliza el método de redondeo del *context* proporcionado o el del contexto actual.

Operandos lógicos

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 Objetos Context

Los contextos son entornos para operaciones aritméticas. Gobiernan la precisión, establecen reglas para el redondeo, determinan qué señales se tratan como excepciones y limitan el rango para los exponentes.

Cada hilo tiene su propio contexto actual, al que se accede o se reemplaza usando las funciones *getcontext()* y *setcontext()* respectivamente:

`decimal`.**getcontext**()
Retorna el contexto actual del hilo activo.

`decimal`.**setcontext**(*c*)
Establece *c* como contexto actual para el hilo activo.

También puedes usar la declaración `with` y la función *localcontext()* para cambiar temporalmente el contexto activo.

`decimal`.**localcontext**(*ctx=None*)
Retorna un gestor de contexto que establecerá el contexto actual para el hilo activo en una copia de *ctx* al ingresar en la declaración with y restaurará el contexto anterior al salir de la misma. Si no se especifica ningún contexto, se utiliza una copia del contexto actual.

Por ejemplo, el siguiente código establece la precisión decimal actual en 42 lugares, realiza un cálculo y luego restaura automáticamente el contexto anterior:

```
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
```

También se pueden crear nuevos contextos utilizando el constructor de la clase *Context* que se describe a continuación. Además, el módulo proporciona tres contextos prediseñados:

class decimal.**BasicContext**

Este es un contexto estándar definido por la Especificación general de la aritmética decimal. La precisión se establece en nueve. El redondeo se establece en *ROUND_HALF_UP*. Se restablecen todos los flags. Todas las trampas están habilitadas (las señales son tratadas como excepciones) excepto *Inexact*, *Rounded* y *Subnormal*.

Debido a que la mayoría de las trampas están habilitadas, este contexto es especialmente útil para la depuración.

class decimal.**ExtendedContext**

Este es un contexto estándar definido por la Especificación general de la aritmética decimal. La precisión se establece en nueve. El redondeo se establece en *ROUND_HALF_EVEN*. Se restablecen todos los flags. No se habilitan trampas (para que no se generen excepciones durante los cálculos).

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**

Este contexto es utilizado por el constructor de la clase *Context* como un prototipo para nuevos contextos. Cambiar un campo (como la precisión) tiene el efecto de cambiar el valor predeterminado para los nuevos contextos creados por el constructor de *Context*.

Este contexto es más útil en entornos con múltiples hilos. Cambiar uno de los campos antes de que se inicien los hilos tiene el efecto de establecer valores predeterminados en todo el sistema. No se recomienda cambiar los campos después de que se hayan iniciado los hilos, ya que requeriría el uso de mecanismos de sincronización para evitar condiciones de carrera entre los hilos.

En entornos de un solo hilo, es preferible no utilizar este contexto en absoluto. En su lugar, simplemente crea contextos explícitamente como se describe a continuación.

The default values are `Context.prec=28`, `Context.rounding=`*ROUND_HALF_EVEN*, and enabled traps for *Overflow*, *InvalidOperation*, and *DivisionByZero*.

Además de los tres contextos proporcionados, se pueden crear nuevos contextos mediante el constructor de la clase *Context*.

class decimal.**Context**(*prec=None*, *rounding=None*, *Emin=None*, *Emax=None*, *capitals=None*, *clamp=None*, *flags=None*, *traps=None*)

Crea un nuevo contexto. Si no se especifica un campo, o es *None*, los valores predeterminados se copian de *DefaultContext*. Si el campo *flags* no está especificado, o es *None*, se restablecen todas los flags.

prec is an integer in the range [`1`, *MAX_PREC*] that sets the precision for arithmetic operations in the context.

La opción *rounding* es una de las constantes enumeradas en la sección *Rounding Modes*.

Los campos *traps* y *flags* enumeran las señales que se deben establecer. Generalmente, los nuevos contextos solo deben establecer trampas y dejar los flags sin establecer.

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 `Emin - prec + 1 <= e <= Emax - 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()
Retorna un duplicado del contexto.

copy_decimal(*num*)
Retorna una copia de la instancia de Decimal num.

create_decimal(*num*)
Crea una nueva instancia de Decimal a partir de *num* pero usando *self* como contexto. A diferencia del constructor de *Decimal*, la precisión del contexto, el método de redondeo, los flags y las trampas se aplican a la conversión.

Esto es útil porque las constantes a menudo se proporcionan con una precisión mayor que la que necesita la aplicación. Otro beneficio es que el redondeo elimina inmediatamente los efectos no deseados de los dígitos más allá de la precisión actual. En el siguiente ejemplo, usar entradas no redondeadas significa que agregar cero a una suma puede cambiar el resultado:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

Este método implementa la operación to-number de la especificación de IBM. Si el argumento es una cadena de caracteres, no se permiten espacios en blanco ni guiones bajos, ni al principio ni al final.

create_decimal_from_float(*f*)
Crea una nueva instancia de Decimal a partir de un flotante *f*, pero redondeando usando *self* como contexto. A diferencia del método de clase *Decimal.from_float()*, la precisión del contexto, el método de redondeo, los flags y las trampas se aplican a la conversión.

```
>>> 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
Traceback (most recent call last):
    ...
decimal.Inexact: None
```

Nuevo en la versión 3.1.

Etiny()
Retorna un valor igual a `Emin - prec + 1` que es el valor mínimo del exponente para resultados subnormales. Cuando ocurre un desbordamiento numérico negativo («underflow»), el exponente se establece en *Etiny*.

Etop()
Retorna un valor igual a `Emax - prec + 1`.

El enfoque habitual para trabajar con decimales es crear instancias de la clase *Decimal* y luego aplicar operaciones aritméticas que tienen lugar dentro del contexto actual para el hilo activo. Un enfoque alternativo es utilizar métodos de contexto para calcular dentro de un contexto específico. Los métodos son similares a los de la clase *Decimal* y aquí solo se relatan brevemente.

abs(*x*)
Retorna el valor absoluto de *x*.

add(*x*, *y*)
Retorna la suma de *x* e *y*.

canonical(*x*)
Retorna el mismo objeto Decimal *x*.

compare(*x*, *y*)
Compara *x* e *y* numéricamente.

compare_signal(*x*, *y*)
Compara los valores de los dos operandos numéricamente.

compare_total(*x*, *y*)
Compara los dos operandos utilizando su representación abstracta.

compare_total_mag(*x*, *y*)
Compara los dos operandos utilizando su representación abstracta, ignorando el signo.

copy_abs(*x*)
Retorna una copia de *x* con el signo establecido en 0.

copy_negate(*x*)
Retorna una copia de *x* con el signo invertido.

copy_sign(*x*, *y*)
Copia el signo de *y* en *x*.

divide(*x*, *y*)
Retorna *x* dividido entre *y*.

divide_int(*x*, *y*)
Retorna *x* dividido entre *y*, truncando el resultado a un número entero.

divmod(*x*, *y*)
Divide dos números y retorna la parte entera del resultado.

exp(*x*)
Returns `e ** x`.

fma(*x*, *y*, *z*)
Retorna *x* multiplicado por *y*, más *z*.

is_canonical(*x*)
Retorna `True` si *x* está en forma canónica, en caso contrario retorna `False`.

is_finite(*x*)
Retorna `True` si *x* es un valor finito, en caso contrario retorna `False`.

is_infinite(*x*)
Retorna `True` si *x* es un valor infinito, en caso contrario retorna `False`.

is_nan(*x*)
Retorna `True` si *x* es un valor qNaN o sNaN , en caso contrario retorna `False`.

is_normal(*x*)
Retorna `True` si *x* es un número normal, en caso contrario retorna `False`.

is_qnan(*x*)
Retorna `True` si *x* es un NaN silencioso, en caso contrario retorna `False`.

is_signed(*x*)
Retorna `True` si *x* es un valor negativo, en caso contrario retorna `False`.

is_snan(*x*)
Retorna `True` si *x* es un NaN señalizador, en caso contrario retorna `False`.

is_subnormal(*x*)
Retorna `True` si *x* es un número subnormal, en caso contrario retorna `False`.

is_zero(*x*)
Retorna `True` si *x* es un cero, en caso contrario retorna `False`.

ln(*x*)
Retorna el logaritmo natural (base e) de *x*.

log10(*x*)
Retorna el logaritmo en base 10 de *x*.

logb(*x*)
Retorna el exponente de la magnitud del MSD («dígito más significativo») del operando.

logical_and(*x*, *y*)
Aplica la operación lógica *and* entre los dígitos de cada operando.

logical_invert(*x*)
Invierte todos los dígitos en *x*.

logical_or(*x*, *y*)
Aplica la operación lógica *or* entre los dígitos de cada operando.

logical_xor(*x*, *y*)
Aplica la operación lógica *xor* entre los dígitos de cada operando.

max(*x*, *y*)
Compara dos valores numéricamente y retorna el mayor de ellos.

max_mag(*x*, *y*)
Compara los valores numéricamente ignorando sus signos.

min(*x*, *y*)
Compara dos valores numéricamente y retorna el menor de ellos.

min_mag(*x*, *y*)
Compara los valores numéricamente ignorando sus signos.

minus(*x*)
Se corresponde con el operador unario de resta (prefijo) de Python.

multiply(*x*, *y*)
Retorna el producto de *x* por *y*.

next_minus(*x*)
Retorna el número más grande representable menor que *x*.

next_plus(*x*)
Retorna el número más pequeño representable mayor que *x*.

next_toward(*x*, *y*)
Retorna el número más cercano a *x*, en la dirección de *y*.

normalize(*x*)
Reduce *x* a su forma más simple.

number_class(*x*)
Retorna una cadena de caracteres indicando la clase de *x*.

plus(*x*)
Se corresponde con el operador unario de suma (prefijo) de Python. Esta operación aplica la precisión y el redondeo establecidos en el contexto, por lo que *no* es una operación de identidad.

power(*x*, *y*, *modulo=None*)
Retorna `x` elevado a la potencia `y`, reconduciendo al módulo `modulo` si se proporciona.

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)` da como resultado `InvalidOperation`, y si `InvalidOperation` no es atrapada, entonces da como resultado `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».

Con tres argumentos, calcula `(x**y) % modulo`. Para la forma de tres argumentos, se mantienen las siguientes restricciones sobre los argumentos:

- los tres argumentos deben ser enteros
- `y` debe ser un valor no negativo
- al menos uno, `x` o `y` , no debe ser cero
- `modulo` no debe ser cero y tener como mínimo los dígitos de la "precisión"

El valor resultante de `Context.power(x, y, modulo)` es igual al valor que se obtendría calculando `(x**y) % modulo` con precisión ilimitada, la diferencia es que se calcula de manera más eficiente . El exponente del resultado es cero, independientemente de los exponentes de `x`, `y` y `modulo`. El resultado siempre es exacto.

quantize(*x*, *y*)
Retorna un valor igual a *x* (redondeado), pero que tiene el exponente de *y*.

radix()
Simplemente retorna 10, ya que es Decimal, :)

remainder(*x*, *y*)
Retorna el resto de la división entera.

El signo del resultado, si no es cero, es el mismo que el del dividendo original.

remainder_near(*x*, *y*)
Retorna `x - y * n`, donde *n* es el número entero más cercano al valor exacto de `x / y` (si el resultado es 0, entonces su signo será el signo de *x*).

rotate(*x*, *y*)
Retorna una copia de *x* rotada *y* veces.

same_quantum(*x*, *y*)
Retorna `True` si los dos operandos tienen el mismo exponente.

scaleb(*x*, *y*)
Retorna el primer operando después de agregar el segundo valor a su exponente.

shift (*x*, *y*)
Retorna una copia de *x* desplazada *y* veces.

sqrt (*x*)
Retorna la raíz cuadrada de un número no negativo para la precisión del contexto.

subtract (*x*, *y*)
Retorna la diferencia entre *x* e *y*.

to_eng_string (*x*)
Convierte a una cadena de caracteres, usando notación de ingeniería si se necesita un exponente.

La notación de ingeniería tiene como exponente un múltiplo de 3. Esto puede dejar hasta 3 dígitos a la izquierda del punto decimal y puede requerir la adición de uno o dos ceros finales.

to_integral_exact (*x*)
Redondea a un entero.

to_sci_string (*x*)
Convierte un número en una cadena de caracteres usando notación científica.

9.4.4 Constantes

Las constantes detalladas en esta sección solo son relevantes para el módulo de C. Se incluyen también en la versión pura de Python por compatibilidad.

	32-bit	64-bit
`decimal.`**`MAX_PREC`**	`425000000`	`999999999999999999`
`decimal.`**`MAX_EMAX`**	`425000000`	`999999999999999999`
`decimal.`**`MIN_EMIN`**	`-425000000`	`-999999999999999999`
`decimal.`**`MIN_ETINY`**	`-849999999`	`-1999999999999999997`

`decimal.`**`HAVE_THREADS`**
El valor es `True`. Está obsoleta, debido ha que Python ahora siempre tiene soporte para hilos.

Obsoleto desde la versión 3.9.

`decimal.`**`HAVE_CONTEXTVAR`**
El valor predeterminado es `True`. Si Python se `configura usando --without-decimal-contextvar`, la versión de C usa un contexto de hilos-locales en lugar de un contexto de corrutinas-locales y el valor de la constante es `False`. Esto es algo más rápido en algunos escenarios de contexto anidado.

Nuevo en la versión 3.9: retro-portado a las versiones 3.7 y 3.8.

9.4.5 Modos de redondeo

decimal.**ROUND_CEILING**
Round towards Infinity.

decimal.**ROUND_DOWN**
Redondear hacia cero.

decimal.**ROUND_FLOOR**
Round towards -Infinity.

decimal.**ROUND_HALF_DOWN**
Redondear al valor contiguo más cercano, con empates hacia cero.

decimal.**ROUND_HALF_EVEN**
Redondear al valor contiguo más cercano, con empates al entero par contiguo.

decimal.**ROUND_HALF_UP**
Redondear al valor contiguo más cercano, con empates alejándose de cero.

decimal.**ROUND_UP**
Redondear alejándose de cero.

decimal.**ROUND_05UP**
Si el último dígito después de redondear hacia cero es 0 ó 5, redondear alejándose de cero, en caso contrario, redondear hacia cero.

9.4.6 Señales

Las señales representan condiciones que surgen durante el cálculo. Cada una se corresponde con un solo flag de contexto y un habilitador de trampas de contexto.

El flag de contexto se establece siempre que se encuentra la condición. Después del cálculo, los flags pueden comprobarse con fines informativos (por ejemplo, para determinar si un cálculo fue exacto). Después de verificar los flags, asegúrate de borrarlos antes de comenzar con el siguiente cálculo.

Si el habilitador de trampas del contexto está configurado para la señal, entonces la condición hace que se lance una excepción de Python. Por ejemplo, si se establece la trampa *DivisionByZero*, se genera una excepción *DivisionByZero* al encontrar la condición.

class decimal.**Clamped**
Cambia un exponente para ajustar las restricciones de representación.

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**
Clase base para otras señales. Es una subclase de *ArithmeticError*.

class decimal.**DivisionByZero**
Señala la división de un número no infinito entre cero.

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**
Indica que se produjo un redondeo y el resultado no es exacto.

Señala que se descartaron dígitos distintos de cero durante el redondeo. Se retorna el resultado redondeado. El flag o la trampa de señal se utiliza para detectar cuando los resultados son inexactos.

class decimal.**InvalidOperation**
Señala que se realizó una operación no válida.

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**
Desbordamiento numérico.

Indicates the exponent is larger than `Context.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**
Se produjo un redondeo, aunque posiblemente no hubo pérdida de información.

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.

Ocurre cuando el resultado de una operación es subnormal (el exponente es demasiado pequeño). Si no está atrapada, se retorna el resultado sin cambios.

class decimal.**Underflow**
Desbordamiento numérico negativo con resultado redondeado a cero.

Ocurre cuando un resultado subnormal se lleva a cero mediante redondeo. *Inexact* y *Subnormal* también se señalan.

class decimal.**FloatOperation**
Habilita una semántica más estricta para mezclar flotantes y objetos Decimal.

Si la señal no está atrapada (predeterminado), se permite mezclar flotantes y objetos Decimal en el constructor de *Decimal*, en el método *create_decimal()* y en todos los operadores de comparación. Tanto la conversión como las comparaciones son exactas. Cualquier ocurrencia de una operación mixta se registra silenciosamente estableciendo *FloatOperation* a los flags del contexto. Las conversiones explícitas usando *from_float()* o *create_decimal_from_float()* no establecen el flag.

En caso contrario (la señal está atrapada), solo las comparaciones de igualdad y las conversiones explícitas permanecen silenciadas. Todas las demás operaciones mixtas lanzan una excepción *FloatOperation*.

La siguiente tabla resume la jerarquía de señales:

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

9.4.7 Notas sobre la representación en coma flotante

Mitigación del error de redondeo usando mayor precisión

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.

Los efectos del error de redondeo pueden amplificarse mediante la suma o resta de cantidades casi compensadas, lo que da como resultado una pérdida de significación. Knuth proporciona dos ejemplos instructivos en los que la aritmética de coma flotante redondeada con precisión insuficiente provoca la ruptura de las propiedades asociativas y distributivas de la suma:

```
# 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')
```

El módulo *decimal* permite restaurar las identidades ampliando la precisión lo suficiente para evitar la pérdida de significación:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

Valores especiales

The number system for the *decimal* module provides special values including `NaN`, `sNaN`, `-Infinity`, `Infinity`, and two zeros, `+0` and `-0`.

Los infinitos se pueden construir directamente con `Decimal('Infinity')`. Además, pueden surgir al dividir entre cero cuando la señal *DivisionByZero* no es interceptada. Asimismo, cuando la señal *Overflow* no es interceptada, un infinito puede resultar del redondeo más allá de los límites del mayor número representable.

Los infinitos tienen signo (afín) y se pueden usar en operaciones aritméticas donde se tratan como números muy grandes e indeterminados. Por ejemplo, adicionar una constante a infinito resulta en otro infinito.

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.

Los ceros con signo pueden resultar de cálculos que desbordan la precisión establecida. Mantienen el signo que habría resultado si el cálculo se hubiera realizado con mayor precisión. Dado que su magnitud es cero, los ceros positivos y negativos se tratan como iguales y su signo es solo informativo.

Además de los dos ceros con signo, que son distintos pero iguales, hay varias representaciones del cero con diferente precisión pero equivalentes en valor. Esto requiere de algo de tiempo para acostumbrarse. Para un ojo habituado a las representaciones normalizadas de coma flotante, no es inmediatamente obvio que el siguiente cálculo retorne un valor igual a cero:

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

9.4.8 Trabajando con hilos

La función *`getcontext()`* accede a un objeto *`Context`* diferente para cada hilo. Tener contextos de hilo separados significa que los hilos pueden realizar cambios (como `getcontext().prec=10`) sin interferir con otros hilos.

Asimismo, la función *`setcontext()`* asigna automáticamente su objetivo al hilo actual.

Si *`setcontext()`* no ha sido invocada antes de *`getcontext()`*, entonces *`getcontext()`* creará automáticamente un nuevo contexto para usar en el hilo actual.

El nuevo contexto es copiado a partir de un contexto prototipo llamado *DefaultContext*. Modifica directamente el objeto *DefaultContext* para controlar los valores predeterminados, de modo que cada hilo utilice los mismos valores en toda la aplicación. Esto debe hacerse *antes* de que se inicien los hilos, para evitar que tenga lugar una condición de carrera entre los mismos al invocar a *`getcontext()`*. Por ejemplo:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.9 Casos prácticos

A continuación hay algunos casos prácticos que sirven como funciones de utilidad y que muestran formas de trabajar con la clase *Decimal*:

```
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)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print(pi())
    3.141592653589793238462643383

    """
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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
    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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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')))
    0.47942553860420300027328793521
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.10 Preguntas frecuentes sobre Decimal

P. Es engorroso escribir `decimal.Decimal('1234.5')`. ¿Hay alguna forma de minimizar la escritura cuando se usa el intérprete interactivo?

R. Algunos usuarios abrevian el constructor a una sola letra:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

P. En una aplicación de coma fija con dos decimales, algunas entradas tienen muchos dígitos decimales y deben redondearse. En cambio, otras no tienen dígitos en exceso y deben ser validadas. ¿Qué métodos deben utilizarse?

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
```

P. Si tengo entradas validadas con dos dígitos decimales, ¿cómo mantengo eso invariante en una aplicación?

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')]
```

P. Algunos valores decimales siempre se imprimen usando notación exponencial. ¿Hay alguna forma de obtener una representación no exponencial?

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.

Si una aplicación no necesita preocuparse por el seguimiento de significación, es fácil eliminar el exponente y los ceros finales, perdiendo significación, pero manteniendo el valor sin cambios:

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

```
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

P. ¿Hay alguna forma de convertir un flotante regular en un *Decimal*?

R. Sí, cualquier número de coma flotante binario se puede expresar exactamente mediante un Decimal, aunque una conversión exacta puede requerir más precisión de la que sugiere la intuición:

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

P. Dentro de un cálculo complejo, cómo puedo asegurarme de que no he obtenido un resultado adulterado debido a una precisión insuficiente o anomalías de redondeo.

R. El módulo decimal facilita la comprobación de resultados. Una buena práctica es volver a ejecutar los cálculos con mayor precisión y con varios modos de redondeo. La obtención de resultados muy dispares indica una precisión insuficiente, problemas relacionados con el modo de redondeo, entradas mal acondicionadas o un algoritmo numéricamente inestable.

P. Noté que la precisión del contexto se aplica a los resultados de las operaciones pero no a las entradas. ¿Hay algo a tener en cuenta al mezclar valores con distintas precisiones?

R. Sí. El principio es que todos los valores se consideran exactos y también lo es la aritmética de esos valores. Solo se redondean los resultados. La ventaja para las entradas es que «lo que escribes es lo que obtienes». Una desventaja es que los resultados pueden parecer extraños si olvidas que las entradas no se han redondeado:

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

La solución es aumentar la precisión o forzar el redondeo de las entradas utilizando la operación unaria más:

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

Alternativamente, las entradas se pueden redondear en el momento que se crean usando el método *Context.create_decimal()*:

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

P. ¿La implementación de CPython es rápida para números grandes?

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[1]. `libmpdec` uses Karatsuba multiplication for medium-sized numbers and the Number Theoretic Transform for very large numbers.

The context must be adapted for exact arbitrary precision arithmetic. `Emin` and `Emax` should always be set to the maximum values, `clamp` should always be 0 (the default). Setting `prec` requires some care.

The easiest approach for trying out bignum arithmetic is to use the maximum value for `prec` as well[2]:

```
>>> setcontext(Context(prec=MAX_PREC, Emax=MAX_EMAX, Emin=MIN_EMIN))
>>> x = Decimal(2) ** 256
>>> x / 128
Decimal(
↪'904625697166532776746648320380374280103671755200316906558262375061821325312')
```

Para resultados inexactos, *MAX_PREC* es demasiado grande en plataformas de 64 bits y la memoria disponible será insuficiente:

```
>>> Decimal(1) / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

[1] Nuevo en la versión 3.3.

[2] Distinto en la versión 3.9: Este enfoque ahora funciona para todos los resultados exactos excepto para las potencias no enteras. También retroportado a 3.7 y 3.8.

On systems with overallocation (e.g. Linux), a more sophisticated approach is to adjust `prec` to the amount of available RAM. Suppose that you have 8GB of RAM and expect 10 simultaneous operands using a maximum of 500MB each:

```
>>> import sys
>>>
>>> # Maximum number of digits for a single operand using 500MB in 8-byte words
>>> # with 19 digits per word (4-byte and 9 digits for the 32-bit build):
>>> maxdigits = 19 * ((500 * 1024**2) // 8)
>>>
>>> # Check that this works:
>>> c = Context(prec=maxdigits, Emax=MAX_EMAX, Emin=MIN_EMIN)
>>> c.traps[Inexact] = True
>>> setcontext(c)
>>>
>>> # Fill the available precision with nines:
>>> x = Decimal(0).logical_invert() * 9
>>> sys.getsizeof(x)
524288112
>>> x + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  decimal.Inexact: [<class 'decimal.Inexact'>]
```

En general (y especialmente en sistemas sin sobreasignación), se recomienda estimar límites aún más estrictos y establecer la trampa *Inexact* si se espera que todos los cálculos sean exactos.

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)
```

The *Fraction* class inherits from the abstract base class *numbers.Rational*, and implements all of the methods and operations from that class. *Fraction* instances are *hashable*, and should be treated as immutable. In addition, *Fraction* has the following properties and methods:

Distinto en la versión 3.2: El constructor de *Fraction* ahora acepta instancias de *float* y *decimal*.

Distinto en la versión 3.9: La función *math.gcd()* ahora se usa para normalizar el *numerator* y *denominator*. *math.gcd()* siempre retorna un tipo *int*. Anteriormente, el tipo de GCD dependía de *numerator* y *denominator*.

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.

classmethod from_float(*flt*)
Alternative constructor which only accepts instances of *float* or *numbers.Integral*. Beware that `Fraction.from_float(0.3)` is not the same value as `Fraction(3, 10)`.

Nota: Desde Python 3.2 en adelante, puedes construir una instancia *Fraction* directamente desde *float*.

---

classmethod from_decimal(*dec*)
Alternative constructor which only accepts instances of *decimal.Decimal* or *numbers.Integral*.

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* <= `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* >= `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()*.

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 y 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 para otro generador de números aleatorios con un período largo compatible y con operaciones de actualización comparativamente simples.

9.6.1 Funciones de contabilidad

`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.

Obsoleto desde la versión 3.9: En el futuro, la *semilla* debe ser de uno de los siguientes tipos: *NoneType*, *int*, *float*, *str*, *bytes*, o *bytearray*.

`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 tenia cuando se llamó a la función *getstate()*.

9.6.2 Funciones para los bytes

random.**randbytes**(*n*)
Genera *n* bytes aleatorios.

Este método no debe utilizarse para generar tokens de seguridad. Utilice *secrets.token_bytes()* en su lugar.

Nuevo en la versión 3.9.

9.6.3 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.

Obsoleto desde la versión 3.10: La conversión automática de tipos que no son enteros a enteros equivalentes está obsoleta. Actualmente `randrange(10.0)` se convierte sin pérdidas a `randrange(10)`. En el futuro, esto lanzará un *TypeError*.

Obsoleto desde la versión 3.10: La excepción que se lanza de valores no enteros como `randrange(10.5)` o `randrange('10')` se cambiará de *ValueError* a *TypeError*.

random.**randint**(*a*, *b*)
Retorna un entero aleatorio *N* tal que `a <= N <= b`. Alias de `randrange(a, b+1)`.

random.**getrandbits**(*k*)
Retorna un entero de Python no negativo con *k* bits aleatorios. Este método se provee con el generador MersenneTwister y algunos otros generadores pueden también proveerlo como una parte opcional de la API. Cuando está disponible, *getrandbits()* permite a *randrange()* manejar rangos arbitrariamente grandes.

Distinto en la versión 3.9: Este método ahora acepta cero para *k*.

9.6.4 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*.

Los *weights* o los *cum_weights* pueden utilizar cualquier tipo numérico que interactúe con los valores `float` retornados por `random()` (eso incluye enteros, flotantes y fracciones pero excluye decimales). Se asume que los pesos son no negativos y finitos. Se lanza un `ValueError` si todos los pesos son cero.

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.

Distinto en la versión 3.9: Genera un `ValueError` si todos los pesos son cero.

random.**shuffle**(*x*[, *random*])
Mezcla la secuencia *x* in-situ.

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 mezclar una secuencia inmutable y retornar una nueva lista mezclada, utilice `muestra(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.

Obsoleto desde la versión 3.9, se eliminará en la versión 3.11: El parámetro opcional *random*.

random.**sample**(*population*, *k*, *, *counts=None*)
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.

Los elementos repetidos pueden especificarse de uno en uno o con el parámetro opcional *counts*, que es una palabra clave. Por ejemplo, `sample(['red', 'blue'], counts=[4, 2], k=5)` es equivalente a `sample(['red', 'red', 'red', 'red', 'blue', 'blue'], k=5)`.

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`.

Distinto en la versión 3.9: Se añadió el parámetro *counts*.

Obsoleto desde la versión 3.9: En el futuro, la *población* debe ser una secuencia. Las instancias de `set` ya no se admiten. El conjunto debe convertirse primero en una `list` o `tuple`, preferiblemente en un orden determinista para que la muestra sea reproducible.

9.6.5 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 <= N <= b` para `a <= b` y `b <= N <= 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) * random()`.

`random.`**`triangular`**`(`*low*, *high*, *mode*`)`
Retorna un número de coma flotante *N* tal que `low <= N <= 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:

```
            x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

`random.`**`gauss`**`(`*mu*, *sigma*`)`
Distribución normal, también llamada la distribución Gaussiana. *mu* es la media y *sigma* es la desviación estándar. Es un poco más rápida que la función *`normalvariate()`* definida debajo.

Nota sobre el multithreading: Cuando dos hilos llaman a esta función simultáneamente, es posible que reciban el mismo valor de retorno. Esto se puede evitar de tres maneras. 1) Hacer que cada hilo utilice una instancia diferente del generador de números aleatorios. 2) Poner bloqueos alrededor de todas las llamadas. 3) Utilizar la función *`normalvariate()`*, más lenta pero segura para los hilos, en su lugar.

`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 radiantes entre 0 y 2**pi*, 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 2**pi*.

`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.

9.6.6 Generador alternativo

class random.**Random**([*seed*])
Esta clase implementa el generador de números pseudoaleatorios predeterminado que usa el módulo *random*.

Obsoleto desde la versión 3.9: En el futuro, la *semilla* debe ser de uno de los siguientes tipos: NoneType, *int*, *float*, *str*, *bytes*, o *bytearray*.

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.

9.6.7 Notas sobre la Reproducibilidad

A veces es útil poder reproducir las secuencias dadas por un generador de números pseudoaleatorios. Al reutilizar un valor de semilla, la misma secuencia debería ser reproducible de una ejecución a otra siempre que no se estén ejecutando múltiples hilos.

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.

9.6.8 Ejemplos

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']
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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:  ten, jack, queen, or king.
>>> dealt = sample(['tens', 'low cards'], counts=[16, 36], k=20)
>>> dealt.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
```

Ejemplo de statistical bootstrapping utilizando el remuestreo con reemplazo para estimar un intervalo de confianza para la media de una muestra:

```
# https://www.thoughtco.com/example-of-bootstrapping-3126155
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):])
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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.')
```

Simulación de tiempos de llegada y entrega de servicios para una cola de múltiples servidores:

```
from heapq import heapify, heapreplace
from random import expovariate, gauss
from statistics import mean, quantiles

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
heapify(servers)
for i in range(1_000_000):
    arrival_time += expovariate(1.0 / average_arrival_interval)
    next_server_available = servers[0]
    wait = max(0.0, next_server_available - arrival_time)
    waits.append(wait)
    service_duration = max(0.0, gauss(average_service_time, stdev_service_time))
    service_completed = arrival_time + wait + service_duration
    heapreplace(servers, service_completed)

print(f'Mean wait: {mean(waits):.1f}   Max wait: {max(waits):.1f}')
print('Quartiles:', [round(q, 1) for q in quantiles(waits)])
```

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 a simulation of a marketplace by Peter Norvig that shows effective use of many of the tools and distributions provided by this module (gauss, uniform, sample, betavariate, choice, triangular, and randrange).

A Concrete Introduction to Probability (using Python) a tutorial by Peter Norvig covering the basics of probability theory, how to write simulations, and how to perform data analysis using Python.

9.6.9 Recetas

La función `random()` por defecto devuelve múltiplos de 2^{-53} en el rango $0.0 \le x < 1.0$. Todos estos números están espaciados uniformemente y son representables exactamente como flotantes de Python. Sin embargo, muchos otros flotadores representables en ese intervalo no son selecciones posibles. Por ejemplo, `0.0595486140802560 9` no es un múltiplo entero de 2^{-53}.

La siguiente receta adopta un enfoque diferente. Todos los flotantes en el intervalo son selecciones posibles. La mantisa proviene de una distribución uniforme de enteros en el rango $2^{52} \le mantisa < 2^{53}$. El exponente proviene de una distribución geométrica en la que los exponentes menores de *-53* ocurren con la mitad de frecuencia que el siguiente exponente mayor.

```
from random import Random
from math import ldexp
```

(continué en la próxima página)

(proviene de la página anterior)

```
class FullRandom(Random):

    def random(self):
        mantissa = 0x10_0000_0000_0000 | self.getrandbits(52)
        exponent = -53
        x = 0
        while not x:
            x = self.getrandbits(32)
            exponent += x.bit_length() - 32
        return ldexp(mantissa, exponent)
```

Todas las *distribuciones de valor real* de la clase utilizarán el nuevo método:

```
>>> fr = FullRandom()
>>> fr.random()
0.05954861408025609
>>> fr.expovariate(0.25)
8.87925541791544
```

La receta es conceptualmente equivalente a un algoritmo que elige entre todos los múltiplos de 2^{-1074} en el rango *0,0 ≤ x < 1,0*. Todos esos números son uniformes, pero la mayoría tienen que ser redondeados al flotante de Python representable más cercano. (El valor 2^{-1074} es el menor flotante positivo no normalizado y es igual a `math.ulp(0.0)`.)

Ver también:

Generating Pseudo-random Floating-Point Values un artículo de Allen B. Downey en el que se describen formas de generar flotantes más refinados que los generados normalmente por *`random()`*.

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)`.

Some datasets use `NaN` (not a number) values to represent missing data. Since NaNs have unusual comparison semantics, they cause surprising or undefined behaviors in the statistics functions that sort data or that count occurrences. The functions affected are `median()`, `median_low()`, `median_high()`, `median_grouped()`, `mode()`, `multimode()`, and `quantiles()`. The `NaN` values should be stripped before calling these functions:

```
>>> from statistics import median
>>> from math import isnan
>>> from itertools import filterfalse

>>> data = [20.7, float('NaN'),19.2, 18.3, float('NaN'), 14.4]
>>> sorted(data)  # This has surprising behavior
```

(continué en la próxima página)

(proviene de la página anterior)

```
[20.7, nan, 14.4, 18.3, 19.2, nan]
>>> median(data)  # This result is unexpected
16.35

>>> sum(map(isnan, data))     # Number of missing values
2
>>> clean = list(filterfalse(isnan, data))  # Strip NaN values
>>> clean
[20.7, 19.2, 18.3, 14.4]
>>> sorted(clean)  # Sorting now works as expected
[14.4, 18.3, 19.2, 20.7]
>>> median(clean)       # This result is now well defined
18.75
```

9.7.1 Promedios y medidas de tendencia central

Estas funciones calculan el promedio o el valor típico de una población o muestra.

mean()	Media aritmética («promedio») de los datos.
fmean()	Media aritmética usando coma flotante, más rápida.
geometric_mean()	Media geométrica de los datos.
harmonic_mean()	Media armónica de los datos.
median()	Mediana (valor central) de los datos.
median_low()	Mediana baja de los datos.
median_high()	Mediana alta de los datos.
median_grouped()	Mediana, o percentil 50, de los datos agrupados.
mode()	Moda única (valor más común) de datos discretos o nominales.
multimode()	Lista de modas (valores más comunes) de datos discretos o nominales.
quantiles()	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.

pstdev()	Desviación típica poblacional de los datos.
pvariance()	Varianza poblacional de los datos.
stdev()	Desviación típica muestral de los datos.
variance()	Varianza muestral de los datos.

9.7.3 Estadísticas para relaciones entre dos entradas

Estas funciones calculan estadísticas sobre las relaciones entre dos entradas.

covariance()	Covarianza muestral de dos variables.
correlation()	Coeficiente de correlación de Pearson para dos variables.
linear_regression()	Pendiente e intersección para regresión lineal simple.

9.7.4 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: The mean is strongly affected by outliers and is not necessarily a typical example of the data points. For a more robust, although less efficient, measure of central tendency, see *`median()`*.

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*, *weights=None*)

Retorna la media armónica de *data*, que debe ser una secuencia o un iterable de números que reales. Si *weights* se omite o es *None*, se asume igual ponderación.

La media armónica es recíproco de la *mean()* aritmética de los recíprocos de los datos. Por ejemplo, la media armónica de tres valores *a*, *b* and *c* es equivalente a `3/(1/a + 1/b + 1/c)`. Si alguno de los valores es cero, el resultado va a ser 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 un automóvil viaja 5 km a 40 km/h, y cuando el tráfico se despeja, acelera a 60 km/h durante los 30 km restantes del viaje. ¿Cuál es su velocidad media?

```
>>> harmonic_mean([40, 60], weights=[5, 30])
56.0
```

Una excepción *StatisticsError* es lanzada si *data* está vacío, algún elemento es menor que cero, o si la suma ponderada no es positiva.

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.

Distinto en la versión 3.10: Soporte añadido a *weights*.

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*.

Detalles de implementación de CPython: 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('aabbbbccddddeeffffgg')
['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')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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.

statistics.**covariance**(*x*, *y*, */*)

Retorna la covarianza de muestra de dos entradas *x* y *y*. La covarianza es una medida de la variabilidad conjunta de dos entradas.

Ambas entradas deben ser del mismo largo (no menor a dos), de lo contrario se lanza *StatisticsError*.

Ejemplos:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> covariance(x, y)
0.75
>>> z = [9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> covariance(x, z)
-7.5
>>> covariance(z, x)
-7.5
```

Nuevo en la versión 3.10.

statistics.**correlation**(*x*, *y*, */*)

Retorna el coeficiente de correlación de Pearson para dos entradas. El coeficiente de correlación de Pearson *r* toma valores entre -1 y +1. Mide la fuerza y dirección de la relación lineal, donde +1 significa una relación muy fuerte, positiva y lineal, -1 una relación muy fuerte, negativa y lineal, y 0 una relación no lineal.

Ambas entradas deben ser del mismo largo (no menor a dos), y no necesitan ser constantes, de lo contrario se lanza *StatisticsError*.

Ejemplos:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y = [9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> correlation(x, x)
1.0
>>> correlation(x, y)
-1.0
```

Nuevo en la versión 3.10.

statistics.**linear_regression**(*x*, *y*, */*)

Retorna la pendiente e intersección de los parámetros de una regresión lineal simple estimados usando mínimos cuadrados ordinarios. La regresión lineal simple describe la relación entre la variable independiente *x* y la variable dependiente *y* en términos de esta función lineal:

*y = slope * x + intercept + noise*

donde `slope` e `intercept` son los parámetros de regresión que son estimados, y `noise` representa la variabilidad de los datos que no fue explicado por la regresión lineal (es igual a la diferencia entre los valores predichos y reales de la variable dependiente).

Ambas entradas deben ser del mismo largo (no menor a dos), y la variable independiente *x* no puede ser constante, de lo contrario se lanza *StatisticsError*.

Por ejemplo, podemos usar las fechas de lanzamiento de las películas de Monty Python para predecir el número acumulativo de películas de Monty Python que se habría producido en 2019 asumiendo que hubiesen mantenido el ritmo:

```
>>> year = [1971, 1975, 1979, 1982, 1983]
>>> films_total = [1, 2, 3, 4, 5]
>>> slope, intercept = linear_regression(year, films_total)
>>> round(slope * 2019 + intercept)
16
```

Nuevo en la versión 3.10.

9.7.5 Excepciones

Se define una sola excepción:

exception statistics.**StatisticsError**
Subclase de *ValueError* para excepciones relacionadas con la estadística.

9.7.6 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 <= X < x+dx) / dx` cuando *dx* tiende a cero.

The relative likelihood is computed as the probability of a sample occurring in a narrow range divided by the width of the range (hence the word «density»). Since the likelihood is relative to other points, its value can be greater than `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 <= x)`.

inv_cdf(*p*)
Compute the inverse cumulative distribution function, also known as the quantile function or the percent-point function. Mathematically, it is written `x : P(X <= 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).

zscore(*x*)
Computa el Standard Score describiendo *x* en términos de los números de desviaciones estándar sobre o bajo la media de una distribución normal: `(x - mean) / stdev`.

Nuevo en la versión 3.9.

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]
```

Normal distributions can be used to approximate Binomial distributions when the sample size is large and when the probability of a successful trial is near 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 has a nice example of a Naive Bayesian Classifier. The challenge is to predict a person's gender from measurements of normally distributed features including height, weight, and foot size.

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'
```

CAPÍTULO 10

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 (*callables*).

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
`count()`	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … indefinidamente o hasta n veces	`repeat(10, 3) --> 10 10 10`

Iteradores que terminan en la secuencia de entrada más corta:

Iterador	Argumentos	Resultados	Ejemplo
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, …	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
chain()	p, q, …	p0, p1, … plast, q0, q1, …	`chain('ABC', 'DEF') --> A B C D E F`
chain.from_iterable()	iterable	p0, p1, … plast, q0, q1, …	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
dropwhile()	pred, seq	seq[n], seq[n+1], comenzando cuando pred falla	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
filterfalse()	pred, seq	elementos de seq donde pred(elem) es falso	`filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`
groupby()	iterable[, key]	sub-iteradores agrupados según el valor de key(v)	
islice()	seq, [start,] stop [, step]	elementos de seq[start:stop:step]	`islice('ABCDEFG', 2, None) --> C D E F G`
pairwise()	iterable	(p[0], p[1]), (p[1], p[2])	`pairwise('ABCDEFG') --> AB BC CD DE EF FG`
starmap()	func, seq	func(*seq[0]), func(*seq[1]), …	`starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000`
takewhile()	pred, seq	seq[0], seq[1], hasta que pred falle	`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`
tee()	it, n	it1, it2, … itn divide un iterador en n	
zip_longest()	p, q, …	(p[0], q[0]), (p[1], q[1]), …	`zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-`

Iteradores combinatorios:

Iterador	Argumentos	Resultados
product()	p, q, … [repeat=1]	producto cartesiano, equivalente a un bucle *for* anidado
permutations()	p[, r]	tuplas de longitud r, en todas los órdenes posibles, sin elementos repetidos
combinations()	p, r	tuplas de longitud r, ordenadas, sin elementos repetidos
combinations_with_replacement()	p, r	tuplas de longitud r, ordenadas, con elementos repetidos

Ejemplos	Resultados
`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`	`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`	`AA AB AC AD BB BC BD CC CD DD`

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 sumable, 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',
 '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 unas 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.

Las tuplas de combinación se emiten en orden lexicográfico según el orden de la entrada *iterable*. Entonces, si la entrada *iterable* está ordenada, las tuplas de combinación se producirán en una secuencia ordenada.

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 <= r <= 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.

Las tuplas de combinación se emiten en orden lexicográfico según el orden de la entrada *iterable*. Entonces, si la entrada *iterable* está ordenada, las tuplas de combinación se producirán en una secuencia ordenada.

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 consecutivamente 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 multilínea puede contener un nombre de campo cada tres líneas). Aproximadamente equivalente a:

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) --> A B
    # islice('ABCDEFG', 2, 4) --> C D
    # islice('ABCDEFG', 2, None) --> C D E F G
    # islice('ABCDEFG', 0, None, 2) --> A C E G
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop or sys.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.`**`pairwise`**(*iterable*)

Retorna los sucesivos pares superpuestos tomados de la entrada *iterable*.

El número de tuplas de 2 elementos en el iterador de salida será uno menos que el número de entradas. Estará vacío si la entrada iterable tiene menos de dos valores.

Aproximadamente equivalente a:

```
def pairwise(iterable):
    # pairwise('ABCDEFG') --> AB BC CD DE EF FG
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)
```

Nuevo en la versión 3.10.

`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.

Las tuplas de permutación se emiten en orden lexicográfico según el orden de la entrada *iterable*. Entonces, si la entrada *iterable* está ordenada, las tuplas de combinación se producirán en una secuencia ordenada.

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:
```

(continué en la próxima página)

(proviene de la página anterior)

```
        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]
                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 <= r <= 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 cuentapasos 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)
```

Antes de que *product()* se ejecute, consume completamente los iterables de entrada, manteniendo grupos de valores en la memoria para generar los productos. En consecuencia, solo es útil con entradas finitas.

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:
        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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
            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 *generator*s que puedan incurrir en costos extra.

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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 an iterator over the last n items"
    # tail(3, 'ABCDEFG') --> 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 pad_none(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 convolve(signal, kernel):
    # See:  https://betterexplained.com/articles/intuitive-convolution/
    # convolve(data, [0.25, 0.25, 0.25, 0.25]) --> Moving average (blur)
    # convolve(data, [1, -1]) --> 1st finite difference (1st derivative)
    # convolve(data, [1, -2, 1]) --> 2nd finite difference (2nd derivative)
    kernel = tuple(kernel)[::-1]
    n = len(kernel)
    window = collections.deque([0], maxlen=n) * n
```

(continué en la próxima página)

(proviene de la página anterior)

```
    for x in chain(signal, repeat(0, n-1)):
        window.append(x)
        yield sum(map(operator.mul, kernel, window))

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 grouper(iterable, n, *, incomplete='fill', fillvalue=None):
    "Collect data into non-overlapping fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, fillvalue='x') --> ABC DEF Gxx
    # grouper('ABCDEFG', 3, incomplete='strict') --> ABC DEF ValueError
    # grouper('ABCDEFG', 3, incomplete='ignore') --> ABC DEF
    args = [iter(iterable)] * n
    if incomplete == 'fill':
        return zip_longest(*args, fillvalue=fillvalue)
    if incomplete == 'strict':
        return zip(*args, strict=True)
    if incomplete == 'ignore':
        return zip(*args)
    else:
        raise ValueError('Expected fill, strict, or ignore')

def triplewise(iterable):
    "Return overlapping triplets from an iterable"
    # triplewise('ABCDEFG') -> ABC BCD CDE DEF EFG
    for (a, _), (b, c) in pairwise(pairwise(iterable)):
        yield a, b, c

def sliding_window(iterable, n):
    # sliding_window('ABCDEFG', 4) -> ABCD BCDE CDEF DEFG
    it = iter(iterable)
    window = collections.deque(islice(it, n), maxlen=n)
    if len(window) == n:
        yield tuple(window)
    for x in it:
        window.append(x)
        yield tuple(window)

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))
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 before_and_after(predicate, it):
    """ Variant of takewhile() that allows complete
        access to the remainder of the iterator.

        >>> it = iter('ABCdEfGhI')
        >>> all_upper, remainder = before_and_after(str.isupper, it)
        >>> ''.join(all_upper)
        'ABC'
        >>> ''.join(remainder)     # takewhile() would lose the 'd'
        'dEfGhI'

        Note that the first iterator must be fully
        consumed before the second iterator can
        generate valid results.
    """
    it = iter(it)
    transition = []
    def true_iterator():
        for elem in it:
            if predicate(elem):
                yield elem
            else:
                transition.append(elem)
                return
    def remainder_iterator():
        yield from transition
        yield from it
    return true_iterator(), remainder_iterator()

def subslices(seq):
    "Return all contiguous non-empty subslices of a sequence"
    # subslices('ABCD') --> A AB ABC ABCD B BC BCD C CD D
    slices = starmap(slice, combinations(range(len(seq) + 1), 2))
    return map(operator.getitem, repeat(seq), slices)

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:
```

(continué en la próxima página)

(proviene de la página anterior)

```
                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:
        iter_except(functools.partial(heappop, 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, **kwds)"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(map(random.choice, 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))
```

(continué en la próxima página)

(proviene de la página anterior)

```
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.choices(range(n), k=r))
    return tuple(pool[i] for i in indices)

def nth_combination(iterable, r, index):
    "Equivalent to list(combinations(iterable, r))[index]"
    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.cache`(*user_function*)

Caché de funciones ilimitado, ligero y simple. Aveces llamado «memoización».

Retorna lo mismo que `lru_cache(maxsize=None)`, creando una envoltura delgada alrededor de una búsqueda de diccionario para los argumentos de la función. Debido a que nunca necesita desalojar los valores antiguos, esto es más pequeño y más rápido que *`lru_cache()`* con un límite de tamaño.

Por ejemplo:

```
@cache
def factorial(n):
    return n * factorial(n-1) if n else 1

>>> factorial(10)      # no previously cached result, makes 11 recursive calls
3628800
>>> factorial(5)       # just looks up cached value result
120
>>> factorial(12)      # makes two new recursive calls, the other 10 are cached
479001600
```

Nuevo en la versión 3.9.

`@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 = tuple(sequence_of_numbers)

    @cached_property
    def stdev(self):
        return statistics.stdev(self._data)
```

La mecánica de *cached_property()* es algo diferente de *property()*. Un atributo de bloques de propiedad normal escribe a menos que se defina un establecedor. Por el contrario, *cached_property* permite escrituras.

El decorador *cached_property* solo se ejecuta en búsquedas y solo cuando no existe un atributo con el mismo nombre. Cuando se ejecuta, *cached_property* escribe en el atributo con el mismo nombre. Las lecturas y escrituras de atributos posteriores tienen prioridad sobre el método *cached_property* y funciona como un atributo normal.

El valor en caché se puede borrar eliminando el atributo. Esto permite que el método *cached_property* se ejecute nuevamente.

Tenga en cuenta que este decorador interfiere con el funcionamiento de diccionarios de intercambio de claves **PEP 412**. Esto significa que los diccionarios de instancias pueden ocupar más espacio de lo habitual.

Además, este decorador requiere que el atributo `__dict__` en cada instancia sea un mapeo mutable. Esto significa que no funcionará con algunos tipos, como las metaclases (ya que los atributos `__dict__` en las instancias de tipos son proxies de solo lectura para el espacio de nombres de la clase) y aquellos que especifican `__slots__` sin incluir `__dict__` como una de las ranuras definidas (ya que tales clases no proporcionan un atributo `__dict__` en absoluto).

Si un mapeo mutable no está disponible o si se desea compartir claves con espacio eficiente, se puede lograr un efecto similar a *cached_property()* apilando *property()* encima de *cache()*:

```
class DataSet:
    def __init__(self, sequence_of_numbers):
        self._data = sequence_of_numbers

    @property
    @cache
    def stdev(self):
        return statistics.stdev(self._data)
```

Nuevo en la versión 3.8.

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.

A comparison function is any callable that accepts two arguments, compares them, and returns a negative number for less-than, zero for equality, or a positive number for greater-than. A key function is a callable that accepts one argument and returns another value to be used as the sort key.

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.

Since a dictionary is used to cache results, the positional and keyword arguments to the function must be *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):
    return sum(sentence.count(vowel) for vowel in 'AEIOUaeiou')
```

Si *maxsize* está configurado como `None`, la función LRU está desactivada y la caché puede crecer sin límites.

If *typed* is set to true, function arguments of different types will be cached separately. If *typed* is false, the implementation will usually regard them as equivalent calls and only cache a single result. (Some types such as *str* and *int* may be cached separately even when *typed* is false.)

Note, type specificity applies only to the function's immediate arguments rather than their contents. The scalar arguments, `Decimal(42)` and `Fraction(42)` are be treated as distinct calls with distinct results. In contrast, the tuple arguments `('answer', Decimal(42))` and `('answer', Fraction(42))` are treated as equivalent.

La función envuelta está instrumentada con una función `cache_parameters()` que retorna un nuevo `dict` que muestra los valores para *maxsize* y *typed*. Esto es solo para fines informativos. La mutación de los valores no tiene ningún efecto.

Para ayudar a medir la efectividad de la caché 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*.

El decorador también proporciona una función `cache_clear()` para limpiar o invalidar la 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.

La caché mantiene referencias de los argumentos y retorna los valores hasta que se caduquen de la caché o hasta que se borre la caché.

Un caché LRU (menos usado recientemente) funciona mejor cuando las llamadas más recientes son los mejores predictores de las próximas llamadas (por ejemplo, el los artículos más populares en un servidor de noticias tienden a cambiar cada día). El límite de tamaño de la caché asegura que la caché no crezca sin límites en procesos de larga ejecución como 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 = 'https://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, currsize=8)
```

Ejemplo de computar eficientemente los «números de Fibonacci» <https://en.wikipedia.org/wiki/Fibonacci_number>‘_ usando una memoria caché 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, currsize=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*.

Nuevo en la versión 3.9: Añadida la función `cache_parameters()`

`@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):
            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.

Nota: Este decorador no intenta anular métodos que han sido declarados en la clase *sus superclases*. Lo que significa que si una superclase define un operador de comparación, *total_ordering* no lo implementará de nuevo, incluso si el método original es abstracto.

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:
...     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*.

To define a generic function, decorate it with the `@singledispatch` decorator. When defining a function using `@singledispatch`, note that the dispatch happens on the type of the first argument:

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
```

To add overloaded implementations to the function, use the `register()` attribute of the generic function, which can be used as a decorator. For functions annotated with types, the decorator will infer the type of the first argument automatically:

```
>>> @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)
...
```

To enable registering *lambdas* and pre-existing functions, the `register()` attribute can also be used in a functional form:

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

The `register()` attribute returns the undecorated function. This enables decorator stacking, *pickling*, and the creation of unit tests for each variant independently:

```
>>> @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.
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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
```

Where there is no registered implementation for a specific type, its method resolution order is used to find a more generic implementation. The original function decorated with `@singledispatch` is registered for the base *`object`* type, which means it is used if no better implementation is found.

If an implementation is registered to an *abstract base class*, virtual subclasses of the base class will be dispatched to that implementation:

```
>>> from collections.abc import Mapping
>>> @fun.register
... def _(arg: Mapping, verbose=False):
...     if verbose:
...         print("Keys & Values")
...     for key, value in arg.items():
...         print(key, "=>", value)
...
>>> fun({"a": "b"})
a => b
```

To check which implementation the generic function will choose for a given type, use the `dispatch()` attribute:

```
>>> 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: The `register()` attribute now supports using type annotations.

class `functools.`**singledispatchmethod**(*func*)

Transformar un método en un *single-dispatch generic function*.

To define a generic method, decorate it with the `@singledispatchmethod` decorator. When defining a function using `@singledispatchmethod`, note that the dispatch happens on the type of the first non-*self* or non-*cls* argument:

```
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
```

`@singledispatchmethod` supports nesting with other decorators such as *`@classmethod`*. Note that to allow for `dispatcher.register`, `singledispatchmethod` must be the *outer most* decorator. Here is the `Negator` class with the `neg` methods bound to the class, rather than an instance of the class:

```
class Negator:
    @singledispatchmethod
    @classmethod
    def neg(cls, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    @classmethod
    def _(cls, arg: int):
        return -arg

    @neg.register
    @classmethod
    def _(cls, arg: bool):
        return not arg
```

The same pattern can be used for other similar decorators: *`@staticmethod`*, *`@abstractmethod`*, and others.

Nuevo en la versión 3.8.

`functools.`**`update_wrapper`**(*wrapper*, *wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

Actualiza una función *wrapper* para que se parezca a la función *wrapped*. Los argumentos opcionales son tuplas para especificar qué atributos de la función original se asignan directamente a los atributos coincidentes en la función contenedora y qué atributos de la función contenedora se actualizan con los atributos correspondientes de la función original. Los valores predeterminados para estos argumentos son las constantes de nivel de módulo `WRAPPER_ASSIGNMENTS` (que se asigna a la función contenedora `__module__`, `__name__`, `__qualname__`, `__annotations__` y `__doc__`, la cadena de caracteres de documentación) y `WRAPPER_UPDATES` (que actualiza el `__dict__` de la función contenedora, es decir, el diccionario de instancia).

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()
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__()`.

Distinto en la versión 3.10: El resultado siempre tiene el tipo exacto *int*. Anteriormente, el resultado podría haber sido una instancia de una subclase de `int`.

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 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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
            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
```

The items can be any type accepted by the operand's *__getitem__()* method. Dictionaries accept any *hashable* value. Lists, tuples, and strings accept an index or a slice:

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1, 3, 5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2, None))('ABCDEFG')
'CDEFG'
>>> 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	`a + b`	`add(a, b)`
Concatenación	`seq1 + seq2`	`concat(seq1, seq2)`
Chequeo de pertenencia	`obj in seq`	`contains(seq, obj)`
División	`a / b`	`truediv(a, b)`
División	`a // b`	`floordiv(a, b)`
Conjunción lógica bit a bit (*bitwise and*)	`a & b`	`and_(a, b)`
Disyunción lógica bit a bit (*bitwise exclusive or*)	`a ^ b`	`xor(a, b)`
Inversión bit a bit (*bitwise inversion*)	`~ a`	`invert(a)`
Disyunción lógica bit a bit (*bitwise or*)	`a | b`	`or_(a, b)`
Exponenciación	`a ** b`	`pow(a, b)`
Identidad	`a is b`	`is_(a, b)`
Identidad	`a is not b`	`is_not(a, b)`
Asignación indexada	`obj[k] = v`	`setitem(obj, k, v)`
Eliminación indexada	`del obj[k]`	`delitem(obj, k)`
Indexado	`obj[k]`	`getitem(obj, k)`
Desplazamiento a izquierda (*left shift*)	`a << b`	`lshift(a, b)`
Módulo	`a % b`	`mod(a, b)`
Multiplicación	`a * b`	`mul(a, b)`
Multiplicación de matrices	`a @ b`	`matmul(a, b)`
Negación (aritmética)	`- a`	`neg(a)`
Negación (lógica)	`not a`	`not_(a)`
Positivo	`+ a`	`pos(a)`
Desplazamiento a derecha (*right shift*)	`a >> b`	`rshift(a, b)`
Asignación por segmento	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
Eliminación por segmento	`del seq[i:j]`	`delitem(seq, slice(i, j))`
Segmentación	`seq[i:j]`	`getitem(seq, slice(i, j))`
Formateo de cadenas	`s % obj`	`mod(s, obj)`
Sustracción	`a - b`	`sub(a, b)`
Chequeo de verdad	`obj`	`truth(obj)`
Ordenado	`a < b`	`lt(a, b)`
Ordenado	`a <= b`	`le(a, b)`
Igualdad	`a == b`	`eq(a, b)`
Diferencia	`a != b`	`ne(a, b)`
Ordenado	`a >= b`	`ge(a, b)`
Ordenado	`a > b`	`gt(a, b)`

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 ejemplo, 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 = itruediv(a, b)` es equivalente a `a /= b`.

operator.**ixor**(*a*, *b*)
operator.**__ixor__**(*a*, *b*)
    `a = ixor(a, b)` es equivalente a `a ^= b`.

CAPÍTULO 11

Acceso a archivos y directorios

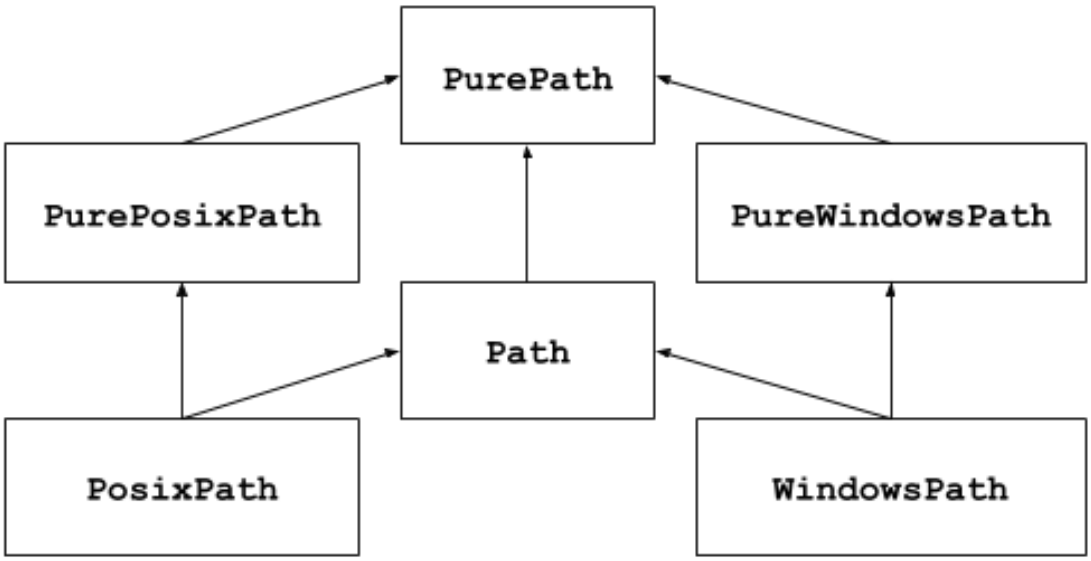
Los módulos descritos en este capítulo tratan de archivos de disco y directorios. Por ejemplo, hay módulos para leer las propiedades de los archivos, manipular rutas de acceso de forma portátil y crear archivos temporales. La lista completa de módulos en este capítulo es:

11.1 `pathlib` — Rutas de sistemas orientada a objetos

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')]
```

Listado de archivos fuente de Python en este árbol de directorios:

```
>>> 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')
```

Cada elemento de *pathsegments* puede ser una cadena que representa un segmento de ruta, un objeto que implemente la interfaz *os.PathLike* la cual retorna una cadena o bien otro objeto ruta:

```
>>> 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('.')
```

If a segment is an absolute path, all previous segments are ignored (like *os.path.join()*):

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

On Windows, the drive is not reset when a rooted relative path segment (e.g., `r'\foo'`) is encountered:

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

Spurious slashes and single dots are collapsed, but double dots (`'..'`) and leading double slashes (`'//'`) are not, since this would change the meaning of a path for various reasons (e.g. symbolic links, UNC paths):

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> 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, including UNC paths:

```
>>> PureWindowsPath('c:/Program Files/')
PureWindowsPath('c:/Program Files')
>>> PureWindowsPath('//server/share/file')
PureWindowsPath('//server/share/file')
```

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

The slash operator helps create child paths, like *os.path.join()*. If the argument is an absolute path, the previous path is ignored. On Windows, the drive is not reset when the argument is a rooted relative path (e.g., `r'\foo'`):

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
>>> p / '/an_absolute_path'
PurePosixPath('/an_absolute_path')
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

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'
```

La representación de una ruta en una cadena de caracteres es la ruta del sistema de archivos sin procesar en sí (en forma nativa, por ejemplo con barras invertidas en Windows), y que puede pasar a cualquier función que tome una ruta como una cadena de caracteres:

```
>>> 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
''
```

Las localizaciones UNC también son consideradas como unidades:

```
>>> 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
'\\'
```

If the path starts with more than two successive slashes, *PurePosixPath* collapses them:

```
>>> PurePosixPath('//etc').root
'//'
>>> PurePosixPath('///etc').root
'/'
>>> PurePosixPath('////etc').root
'/'
```

Nota: This behavior conforms to *The Open Group Base Specifications Issue 6*, paragraph 4.11 Pathname Resolution:

«A pathname that begins with two successive slashes may be interpreted in an implementation-defined manner, although more than two leading slashes shall be treated as a single slash.»

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
'/'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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:/')
```

Distinto en la versión 3.10: La secuencia de padres ahora admite *rebanada* y valores de índice negativos.

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'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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_relative_to**(**other*)
Retorna si esta ruta es relativa o no a la *otra* ruta.

```
>>> p = PurePath('/etc/passwd')
>>> p.is_relative_to('/etc')
True
>>> p.is_relative_to('/usr')
False
```

Nuevo en la versión 3.9.

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*)
Haga coincidir esta ruta con el *pattern* global proporcionado. Retorna `True` si la coincidencia es exitosa, `False` en caso contrario.

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*)

Computa una versión de la ruta en relación a la ruta representada por *other*. Si es imposible, se genera *ValueError*:

```
>>> 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' is not in the subpath of '/usr' OR one path is␣
↪relative and the other absolute.
```

NOTA: Esta función es parte de *PurePath* y trabaja con strings. No revisa ni accede a la estructura de archivos subyacentes.

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_stem**(*stem*)

Retorna una nueva ruta con *stem* cambiado. Si la ruta original no tiene nombre, se lanza *ValueError*:

```
>>> p = PureWindowsPath('c:/Downloads/draft.txt')
>>> p.with_stem('final')
PureWindowsPath('c:/Downloads/final.txt')
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_stem('lib')
PureWindowsPath('c:/Downloads/lib.gz')
>>> p = PureWindowsPath('c:/')
>>> p.with_stem('')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 861, in with_stem
    return self.with_name(stem + self.suffix)
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 851, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

Nuevo en la versión 3.9.

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')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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 ~). Si el directorio de inicio no se puede resolver, se lanza *RuntimeError*.

```
>>> Path.home()
PosixPath('/home/antoine')
```

Nuevo en la versión 3.5.

Path.**stat**(*, *follow_symlinks=True*)
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.

Este método normalmente sigue enlaces simbólicos; para evitar enlaces simbólicos, agregue el argumento `follow_symlinks = False`, o use `lstat()`.

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

Distinto en la versión 3.10: Se agregó el parámetro *follow_symlinks*.

Path.**chmod**(*mode*, *, *follow_symlinks=True*)
Cambia el modo y los permisos de archivo, como *os.chmod()*.

Este método normalmente sigue enlaces simbólicos. Algunas versiones de Unix admiten el cambio de permisos en el enlace simbólico en sí; en estas plataformas puede agregar el argumento `follow_symlinks=False`, o usar *lchmod()*.

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

Distinto en la versión 3.10: Se agregó el parámetro *follow_symlinks*.

Path.**exists**()
Si la ruta apunta a un archivo o directorio existente:

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
```

(continué en la próxima página)

(proviene de la página anterior)

```
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()*. Si el directorio de inicio no se puede resolver, se lanza *RuntimeError*.

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

Nuevo en la versión 3.5.

`Path.`**`glob`**`(`*pattern*`)`
Analiza el patrón comodín relativo a esta ruta para producir todos los archivos coincidentes (de cualquier tipo):

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
```

Los patrones son los mismos que para *fnmatch*, con el agregado de «`**`» que 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.

Levanta un *auditing event* `pathlib.Path.glob` con argumentos `self`, `pattern`.

`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` también se retorna si la ruta no existe o es un enlace simbólico roto; se propagan otros errores (como errores de permiso).

`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` también se retorna si la ruta no existe o es un enlace simbólico roto; se propagan otros errores (como errores de permiso).

Path.**is_mount**()
Retorna `True` si la ruta es un *punto de montaje*; un punto en un sistema de archivos donde otro sistema de archivo se encuentra montado. En POSIX, la función comprueba si el padre de *path* , `path/..`, se encuentra en un dispositivo diferente que *path*, o si `path/..` y *path* apuntan al mismo i-nodo en el mismo dispositivo — esto debería detectar puntos de montajes para todas las variantes Unix y POSIX. No está implementado en 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` también se retorna si la ruta no existe o es un enlace simbólico roto; se propagan otros errores (como errores de permiso).

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` también se retorna si la ruta no existe o es un enlace simbólico roto; se propagan otros errores (como errores de permiso).

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` también se retorna si la ruta no existe o es un enlace simbólico roto; se propagan otros errores (como errores de permiso).

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` también se retorna si la ruta no existe o es un enlace simbólico roto; se propagan otros errores (como errores de permiso).

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 a 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.**readlink**()
Retorna la ruta a la que apunta el vínculo simbólico (como lo retorna *os.readlink()*):

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> p.readlink()
PosixPath('setup.py')
```

Nuevo en la versión 3.9.

`Path.`**`rename`**(*target*)

Rename this file or directory to the given *target*, and return a new Path instance pointing to *target*. On Unix, if *target* exists and is a file, it will be replaced silently if the user has permission. On Windows, if *target* exists, *FileExistsError* will be raised. *target* can be either a string or another path object:

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
PosixPath('bar')
>>> target.open().read()
'some text'
```

La ruta de destino puede ser absoluta o relativa. Las rutas de acceso relativas se interpretan en relación con el directorio de trabajo actual, *no* el directorio del objeto Path.

It is implemented in terms of *os.rename()* and gives the same guarantees.

Distinto en la versión 3.8: Valor de retorno agregado, retorna la nueva instancia de *Path*.

`Path.`**`replace`**(*target*)

Rename this file or directory to the given *target*, and return a new Path instance pointing to *target*. If *target* points to an existing file or empty directory, it will be unconditionally replaced.

La ruta de destino puede ser absoluta o relativa. Las rutas de acceso relativas se interpretan en relación con el directorio de trabajo actual, *no* el directorio del objeto Path.

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: El argumento *strict* (el comportamiento previo a 3.6 es *strict* = `True`).

`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'),
```

(continué en la próxima página)

(proviene de la página anterior)

```
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

Levanta un *auditing event* `pathlib.Path.rglob` con argumentos `self`, `pattern`.

`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
>>> p.lstat().st_size
8
```

Nota: El orden de los argumentos (*link*, *target*) es el reverso de `os.symlink()`"s.

`Path.`**`hardlink_to`**`(`*target*`)`
Hace de esta ruta un enlace fijo que apunta al mismo archivo que *target*.

Nota: El orden de los argumentos (link, target) es el reverso de `os.link()`.

Nuevo en la versión 3.10.

`Path.`**`link_to`**`(`*target*`)`
Hace que *taget* sea un enlace fijo a esta ruta.

Advertencia: Esta función no hace que esta ruta sea un enlace fijo a *target*, a pesar de la implicación de los nombres de la función y los argumentos. El orden de los argumentos (target, link) es el inverso de `Path.symlink_to()` y `Path.hardlink_to()`, pero coincide con el de `os.link()`.

Nuevo en la versión 3.8.

Obsoleto desde la versión 3.10: Este método está deprecado en favor de `Path.hardlink_to()`, ya que el orden de los argumentos de `Path.link_to()` no coincide con el de `Path.symlink_to()`.

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*, *newline=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.

Distinto en la versión 3.10: Se agregó el parámetro *newline*.

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: No todos los pares de funciones / métodos siguientes son equivalentes. Algunos de ellos, a pesar de tener algunos casos de uso superpuestos, tienen semánticas diferentes. Incluyen *os.path.abspath()* y *Path.resolve()*, *os.path.relpath()* y *PurePath.relative_to()*.

---

os y os.path	pathlib
os.path.abspath()	Path.resolve()[1]
os.chmod()	Path.chmod()
os.mkdir()	Path.mkdir()
os.makedirs()	Path.mkdir()
os.rename()	Path.rename()
os.replace()	Path.replace()
os.rmdir()	Path.rmdir()
os.remove(), os.unlink()	Path.unlink()
os.getcwd()	Path.cwd()
os.path.exists()	Path.exists()
os.path.expanduser()	Path.expanduser() y Path.home()
os.listdir()	Path.iterdir()
os.path.isdir()	Path.is_dir()
os.path.isfile()	Path.is_file()
os.path.islink()	Path.is_symlink()
os.link()	Path.hardlink_to()
os.symlink()	Path.symlink_to()
os.readlink()	Path.readlink()
os.path.relpath()	PurePath.relative_to()[2]
os.stat()	Path.stat(), Path.owner(), Path.group()
os.path.isabs()	PurePath.is_absolute()
os.path.join()	PurePath.joinpath()
os.path.basename()	PurePath.name
os.path.dirname()	PurePath.parent
os.path.samefile()	Path.samefile()
os.path.splitext()	PurePath.stem and PurePath.suffix

Notas al pie

11.2 os.path — Manipulaciones comunes de nombre de ruta

Source code: Lib/posixpath.py (for POSIX) and Lib/ntpath.py (for Windows).

This module implements some useful functions on pathnames. To read or write files see *open()*, and for accessing the filesystem see the *os* module. The path parameters can be passed as strings, or bytes, or any object implementing the *os.PathLike* protocol.

Unlike a Unix shell, Python does not do any *automatic* path expansions. Functions such as *expanduser()* and *expandvars()* can be invoked explicitly when an application desires shell-like path expansion. (See also the *glob* module.)

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

[1] *os.path.abspath()* no resuelve enlaces simbólicos mientras que *Path.resolve()* sí.

[2] *PurePath.relative_to()* requires `self` to be the subpath of the argument, but *os.path.relpath()* does not.

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 valida.

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 invalidas porque trabaja un carácter a la vez. Para obtener una ruta valida, 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 se establece, de lo contrario se usará una combinación de `HOMEPATH` y `HOMEDRIVE`. Un `~user` inicial se maneja comprobando que el último componente de directorio del directorio de inicio del usuario actual coincide con `USERNAME`, y reemplazándolo si es así.

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 segments intelligently. The return value is the concatenation of *path* and all members of **paths*, with exactly one directory separator following each non-empty part, except the last. That is, the result will only end in a separator if the last part is either empty or ends in a separator. If a segment is an absolute path (which on Windows requires both a drive and a root), then all previous segments are ignored and joining continues from the absolute path segment.

On Windows, the drive is not reset when a rooted path segment (e.g., `r'\foo'`) is encountered. If a segment is on a different drive or is an absolute path, all previous segments are ignored and the drive is reset. Note that since there is a current directory for each drive, `os.path.join("c:", "foo")` represents a path relative to the current directory on drive `C:` (`c:foo`), not `c:\foo`.

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()*.

Nota:

On POSIX systems, in accordance with IEEE Std 1003.1 2013 Edition; 4.13 Pathname Resolution, if a pathname begins with exactly two slashes, the first component following the leading characters may be interpreted in an implementation-defined manner, although more than two leading characters shall be treated as a single character.

Distinto en la versión 3.6: Acepta un *path-like object*.

---

os.path.**realpath**(*path*, *, *strict=False*)

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).

Si no existe una ruta o se encuentra un bucle de enlace simbólico, y *strict* es `True`, se genera `OSError`. Si *strict* es `False`, la ruta se resuelve en la medida de lo posible y cualquier resto se anexa sin verificar si existe.

Nota: Esta función emula el procedimiento del sistema operativo para hacer que una ruta sea canónica, que difiere ligeramente entre Windows y UNIX con respecto a cómo interactúan los enlaces y los componentes de la ruta posterior.

Las API del sistema operativo hacen que las rutas sean canónicas según sea necesario, por lo que normalmente no es necesario llamar a esta función.

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.

Distinto en la versión 3.10: Se agregó el parámetro *strict*.

os.path.**relpath**(*path*, *start=os.curdir*)

Retorna un nombre de ruta relativo a *path* desde el directorio actual o de un directorio *start* opcional. Este es un cálculo de ruta: No se accede al sistema de archivos para confirmar la existencia o la naturaleza de *path* o *start*. En Windows, se lanza `ValueError` cuando *path* y *start* están en discos diferentes.

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:

```
>>> splitdrive("c:/dir")
("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:

```
>>> splitdrive("//host/computer/dir")
("//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.

Si la ruta no contiene ninguna extensión, *ext* será `''`:

```
>>> splitext('bar')
('bar', '')
```

Si la ruta contiene una extensión, *ext* se establecerá como esta extensión, incluido el punto por delante. Tenga en cuenta que los puntos anteriores se ignorarán:

```
>>> splitext('foo.bar.exe')
('foo.bar', '.exe')
>>> splitext('/foo/bar.exe')
('/foo/bar', '.exe')
```

Leading periods of the last component of the path are considered to be part of the root:

```
>>> splitext('.cshrc')
('.cshrc', '')
>>> splitext('/foo/....jpg')
('/foo/....jpg', '')
```

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(encoding="utf-8"):
    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 en consecuencia retorna un objeto similar a un archivo abierto. Si *encoding* y/o *errors* son especificados, serán pasados al enlace como argumentos de palabras clave adicionales. Este módulo proporciona *hook_compressed()* para admitir archivos comprimidos.

La siguiente función es la interfaz principal de este módulo:

`fileinput.`**`input`**(*files=None*, *inplace=False*, *backup=''*, ***, *mode='r'*, *openhook=None*, *encoding=None*, *errors=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'), encoding="utf-8") 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.

Distinto en la versión 3.10: Los parámetros exclusivos de palabra clave *encoding* y *errors* son añadidos.

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*, *encoding=None*, *errors=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.

Puedes especificar *encoding* y *errors* que son pasados a *open()* o *openhook*.

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.

Distinto en la versión 3.10: Los parámetros exclusivos de palabra clave *encoding* y *errors* son añadidos.

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*, *, *encoding=None*, *errors=None*)
: 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).

  Los valores *encoding* y *errors* se pasan a *io.TextIOWrapper* para archivos comprimidos y para abrir archivos normales.

  Ejemplo de uso: `fi = fileinput.FileInput(openhook=fileinput.hook_compressed, encoding="utf-8")`

  Distinto en la versión 3.10: Los parámetros exclusivos de palabra clave *encoding* y *errors* son añadidos.

`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*.

  Obsoleto desde la versión 3.10: This function is deprecated since *fileinput.input()* and *FileInput* now have *encoding* and *errors* parameters.

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.lstat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

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 (macOS 10.6+).

stat.**UF_HIDDEN**
El archivo no debe ser mostrado en una GUI (macOS 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.

Consulte la página de manual de sistemas * BSD o macOS *chflags(2)* para obtener 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 y las firmas *os.stat()* (tipo de fichero, tamaño, y tiempo de modificación) de los dos ficheros son idénticas, los ficheros se consideran iguales

De lo contrario, los ficheros son tratados como diferentes si sus tamaños o contenidos difieren.

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 que asigna nombres en *common_dirs* a instancias *dircmp* (o instancias MyDirCmp si esa instancias es del tipo MyDirCmp, una subclase de *dircmp*).

Distinto en la versión 3.10: Anteriormente las entradas siempre eran instancias *dircmp*. Ahora son del mismo tipo que *self*, si *self* es una subclase de *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).

On platforms that are neither Posix nor Cygwin, TemporaryFile is an alias for NamedTemporaryFile.

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*)

This function operates exactly as *TemporaryFile()* does, except that the file is guaranteed to have a visible name in the file system (on Unix, the directory entry is not unlinked). That name can be retrieved from the `name` attribute of the returned file-like object. Whether the name can be used to open the file a second time, while the named temporary file is still open, varies across platforms (it can be so used on Unix; it cannot on Windows). If *delete* is true (the default), the file is deleted as soon as it is closed. The returned object is always a file-like object whose `file` attribute is the underlying true file object. This file-like object can be used in a `with` statement, just like a normal file.

Lanza un *evento de auditoria* `tempfile.mkstemp` con argumento `fullpath`.

Distinto en la versión 3.8: Se agregó el parámetro *errors*.

`class` `tempfile.`**`SpooledTemporaryFile`**(*max_size=0*, *mode='w+b'*, *buffering=- 1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

This class operates exactly as *TemporaryFile()* does, except that data is spooled in memory until the file size exceeds *max_size*, or until the file's `fileno()` method is called, at which point the contents are written to disk and operation proceeds as with *TemporaryFile()*.

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*.

`class` `tempfile.`**`TemporaryDirectory`**(*suffix=None*, *prefix=None*, *dir=None*, *ignore_cleanup_errors=False*)

This class securely creates a temporary directory using the same rules as *mkdtemp()*. The resulting object can be used as a context manager (see *Ejemplos*). On completion of the context or destruction of the temporary directory object, the newly created temporary directory and all its contents are removed from the filesystem.

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()`. Si *ignore_cleanup_errors* es verdadero, se ignorará cualquier excepción no controlada durante la limpieza explícita o implícita (como un *PermissionError* al eliminar archivos abiertos en Windows) y los elementos extraíbles restantes se eliminarán según el «mejor esfuerzo». De lo contrario, se lanzarán errores en cualquier contexto que se realice la limpieza (la llamada `cleanup()`, al salir del gestor de contexto, cuando el objeto se recolecta como basura o durante el cierre del intérprete).

Lanza un *evento de auditoria* `tempfile.mkdtemp` con argumento `fullpath`.

Nuevo en la versión 3.2.

Distinto en la versión 3.10: Se agregó el parámetro *ignore_cleanup_errors*.

tempfile.**mkstemp**(*suffix=None*, *prefix=None*, *dir=None*, *text=False*)

Crea un archivo temporal de la manera más segura posible. No hay condiciones de carrera en la creación del archivo, suponiendo que la plataforma implemente correctamente el indicador `os.O_EXCL` para `os.open()`. El archivo se puede leer y escribir solo mediante el ID del usuario que lo crea. Aunque la plataforma utilice bits de permiso para indicar si un archivo es ejecutable, nadie puede ejecutar el archivo. El descriptor de archivo no es heredado por los procesos hijos.

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''`.

Si se especifica *text* y es verdadero, el archivo se abre en modo texto. De lo contrario, (por defecto) el archivo se abre en modo binario.

`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 especifica 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.

Distinto en la versión 3.10: Siempre retorna str. Anteriormente retornaría cualquier valor *tempdir* independientemente del tipo siempre que no fuera `None`.

`tempfile.`**`gettempdirb`**`()`
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 sobrescribir 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. Este es el enfoque recomendado que no toma por sorpresa otro código inesperado al cambiar el comportamiento global de la API.

`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, incluyendo su tipo, bytes o str. No puede ser un *path-like object*.

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()*.

Nota: Tenga en cuenta que si establece `tempdir` en un valor de bytes, hay un efecto secundario desagradable: el tipo de retorno global predeterminado de *mkstemp()* y *mkdtemp()* cambia a bytes cuando no se proporcionan argumentos explícitos `prefix`, `suffix` o `dir` de tipo str. Por favor, no escriba código esperando o dependiendo de esto. Este comportamiento incómodo se mantiene por compatibilidad con la implementación histórica.

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

# 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/tmptjujjt'
>>> 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

The *glob* module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but `*`, `?`, and character ranges expressed with `[]` will be correctly matched. This is done by using the *os.scandir()* and *fnmatch.fnmatch()* functions in concert, and not by actually invoking a subshell.

Note that files beginning with a dot (`.`) can only be matched by patterns that also start with a dot, unlike *fnmatch.fnmatch()* or *pathlib.Path.glob()*. (For tilde and shell variable expansion, use *os.path.expanduser()* and *os.path.expandvars()*.)

Para una coincidencia literal, envuelve los meta-caracteres en corchetes. Por ejemplo, `'[?]'` empareja el caracter `'?'`.

Ver también:

El módulo *pathlib* ofrece objetos ruta de alto nivel.

glob.**glob**(*pathname*, ***, *root_dir=None*, *dir_fd=None*, *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 *root_dir* no es `None`, debería ser un *path-like object* que especifique el directorio raíz para la búsqueda. Tiene el mismo efecto en *glob()* que cambiar el directorio actual antes de llamarlo. Si *pathname* es relativo, el resultado contendrá rutas relativas a *root_dir*.

Esta función puede admitir *rutas relativas a descriptores de directorio* con el parámetro *dir_fd*.

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`.

Lanza un *auditing event* `glob.glob/2` con argumentos `pathname`, `recursive`, `root_dir`, `dir_fd`.

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 «`**`».

Distinto en la versión 3.10: Se agregaron los parámetros *root_dir* y *dir_fd*.

glob.**iglob**(*pathname*, ***, *root_dir=None*, *dir_fd=None*, *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`.

Lanza un *auditing event* `glob.glob/2` con argumentos `pathname`, `recursive`, `root_dir`, `dir_fd`.

Distinto en la versión 3.5: Soporte para globs recursivos usando «`**`».

Distinto en la versión 3.10: Se agregaron los parámetros *root_dir* y *dir_fd*.

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 *seq*
`[!seq]`	coincide con cualquier carácter ausente en *seq*

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*)
Construye una lista a partir de los elementos de los *names* iterables que coinciden con el *pattern*. Es lo mismo que `[n for n in names if fnmatch(n, pattern)]`, pero implementado de manera más eficiente.

`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 módulo 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: Las llamadas a sistema de copia rápida específicas 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*. If *dst* specifies a file that already exists, it will be replaced. 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: Las llamadas a sistema de copia rápida específicas 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.

When *follow_symlinks* is false, and *src* is a symbolic link, *copy2()* attempts to copy all metadata from the *src* symbolic link to the newly created *dst* symbolic link. However, this functionality is not available on all platforms. On platforms where some or all of this functionality is unavailable, *copy2()* will preserve all the metadata it can; *copy2()* never raises an exception because it cannot preserve file metadata.

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: Las llamadas a sistema de copia rápida específicas 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*)
Recursively copy an entire directory tree rooted at *src* to a directory named *dst* and return the destination directory. All intermediate directories needed to contain *dst* will also be created by default.

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 gob.

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()`).

If *dirs_exist_ok* is false (the default) and *dst* already exists, a `FileExistsError` is raised. If *dirs_exist_ok* is true, the copying operation will continue if it encounters existing directories, and files within the *dst* tree will be overwritten by corresponding files from the *src* tree.

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: Added the *copy_function* argument to be able to provide a custom copy function. Added the *ignore_dangling_symlinks* argument to silence dangling symlinks errors when *symlinks* is false.

Distinto en la versión 3.8: Las llamadas a sistema de copia rápida específicas 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.

Si *copy_function* se proporciona, debe ser un invocable que toma dos argumentos *src* y *dst*, y se usará para copiar *src* a *dest* si no se puede usar *os.rename()*. Si el origen es un directorio, se invoca *copytree()* y se le pasa `copy_function()`. El *copy_function* por defecto es *copy2()*. El uso de *copy()* como *copy_function* permite que el movimiento se realice correctamente cuando no es posible copiar los metadatos también, a expensas de no copiar ninguno de los metadatos.

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: Las llamadas a sistema de copia rápida específicas 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*.

Distinto en la versión 3.9: Acepta un *objeto tipo ruta* como *src* y *dst*.

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 (*scrname*, *dstname*, *exception*).

Operaciones de copia eficientes dependientes de la plataforma

A partir de Python 3.8, todas las funciones que incluyen una copia de archivo (*copyfile()*, *copy()*, *copy2()*, *copytree()*, y *move()*) puede usar llamadas a sistema de «copia rápida» específicas a cada plataforma para poder copiar el archivo de forma más eficiente (véase bpo-33671). «copia rápida» significa que la operación de copia ocurre dentro del núcleo, evitando el uso de búferes en espacio de usuario en Python como en «`outfd.write(infd.read())`».

En macOS, se usa 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*

An example that uses the *ignore_patterns()* helper:

```
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 es un directorio que será el directorio raíz del archivo, todas las rutas en el archivo serán relativas a él; por ejemplo, típicamente nos cambiaremos de directorio (*chdir*) a *root_dir* antes de crear el archivo.

base_dir es el directorio desde donde iniciamos el archivado; i.e, *base_dir* será el prefijo común para todos los archivos y directorios en el archivo. *base_dir* debe ser dado de forma relativa a *root_dir*. En *Ejemplo de archivado con base_dir* se ve un ejemplo de cómo usar *base_dir* y *root_dir* juntos.

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 when custom archivers registered with `register_archive_format()` are used. In this case it temporarily changes the current working directory of the process to perform archiving.

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"`.

Distinto en la versión 3.10.6: This function is now made thread-safe during creation of standard `.zip` and tar archives.

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 el formato pax POSIX.1-2001 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 3.10.12, 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 3.10.12: Added the *filter* argument.

shutil.**register_unpack_format**(*name*, *extensions*, *function*[, *extra_args*[, *description*]])
Registra un formato de desempaque. *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 desempaquetado 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
```

Ejemplo de archivado con *base_dir*

En este ejemplo, similar al *de arriba*, mostramos cómo usar `make_archive()`, pero esta vez utilizando *base_dir*. Ahora tenemos la siguiente estructura de directorios:

```
$ tree tmp
tmp
└── root
    └── structure
        ├── content
        │   └── please_add.txt
        └── do_not_add.txt
```

En el archivo final, `please_add.txt` debería estar incluido, pero `do_not_add.txt` no. Por lo tanto usamos lo siguiente:

```
>>> 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'
```

Listando los archivos en el archivo resultante nos da:

```
$ 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`.

Ver también: *The Single UNIX Specification*, Versión 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.

CAPÍTULO 12

Persistencia de datos

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 modulo `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 modulo `pickle` **no es seguro**. Solo deserialize 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 deserialize 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 modulo *pickle* difiere de *marshal* en varias formas significativas:

- El modulo *pickle* realiza un seguimiento de los objetos que ya ha serializado, para que las referencias posteriores al mismo objeto no se serializen 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 limite único entre las versiones del lenguaje.

Comparación con `json`

There are fundamental differences between the pickle protocols and JSON (JavaScript Object Notation):

- 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 modulo *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 serialzados 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 modulo `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.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of *new-style classes*. Refer to **PEP 307** for information about improvements brought by protocol 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.

Arguments *fix_imports*, *encoding*, *errors*, *strict* and *buffers* have the same meaning as in the `Unpickler` constructor.

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 deserializacion 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 sobreescribirlo.

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(*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 reconstituidos 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 sobreescribir 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, and False;
- integers, floating-point numbers, complex numbers;
- strings, bytes, bytearrays;
- tuples, lists, sets, and dictionaries containing only picklable objects;
- functions (built-in and user-defined) accessible from the top level of a module (using def, not lambda);
- classes accessible from the top level of a module;
- 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* lanzaran 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()*.

Note that functions (built-in and user-defined) are pickled by fully *qualified name*, not by value.[2] This means that only the function name is pickled, along with the name of the containing module and classes. Neither the function's code, nor any of its function attributes are pickled. Thus the defining module must be importable in the unpickling environment, and the module must contain the named object, otherwise an exception will be raised.[3]

Similarly, classes are pickled by fully qualified name, so the same restrictions in the unpickling environment apply. Note that none of the class's code or data is pickled, so in the following example the class attribute attr is not restored in the unpickling environment:

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

These restrictions are why picklable functions and classes must be defined at the top level of a module.

De manera similar, cuando las instancias de clases son serializadas con *pickle*, el código y los datos de la clase no son serializadas 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 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__().

[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.

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 restore(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 subclases 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 subclases 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
        if type_tag == "MemoRecord":
```

(continué en la próxima página)

[5] The limitation on alphanumeric characters is due to the fact that persistent IDs in protocol 0 are delimited by the newline character. Therefore if any kind of newline characters occurs in persistent IDs, the resulting pickled data will become unreadable.

(proviene de la página anterior)

```
            # 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)
```

does the same but all instances of `MyPickler` will by default share the private dispatch table. On the other hand, the code

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

modifies the global dispatch table shared by all users of the *copyreg* module.

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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
        # Copy the object's state from self.__dict__ which contains
        # 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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
    def reducer_override(self, obj):
        """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()
```

A sample usage of our unpickler working as intended:

```
>>> 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 deserialize 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+4j],
    '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_ob=None*)
Declares that *function* should be used as a «reduction» function for objects of type *type*. *function* should return either a string or a tuple containing two or three elements. See the `dispatch_table` for more details on the interface of *function*.

The *constructor_ob* parameter is a legacy feature and is now ignored, but if passed it must be a callable.

Note that the `dispatch_table` attribute of a pickler object or subclass of `pickle.Pickler` can also be used for declaring reduction functions.

12.2.1 Ejemplo

El siguiente ejemplo pretende mostrar cómo registrar una función pickle y cómo se utilizará:

```
>>> import copyreg, copy, pickle
>>> class C:
...     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, *pickles* creados con `pickle.DEFAULT_PROTOCOL` 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).

Distinto en la versión 3.10: Ahora se usa `pickle.DEFAULT_PROTOCOL` como el protocolo *pickle* por defecto.

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 la mayoría de los métodos y operaciones admitidos por los diccionarios (excepto copiar, constructores y los operadores `|` y `|=`). 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, *pickles* creados con `pickle.DEFAULT_PROTOCOL` 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.

Distinto en la versión 3.10: Ahora se usa `pickle.DEFAULT_PROTOCOL` como el protocolo *pickle* por defecto.

class shelve.**BsdDbShelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)
Una subclase de `Shelf` que expone `first()`, `next()`, `previous()`, `last()` y `set_location()` que están disponibles en el módulo de terceros `bsddb` de pybsddb pero no en otros módulos de base de datos. El objeto *dict* que se pasa al constructor debe admitir esos métodos. Esto se logra generalmente llamando a uno de los siguientes `bsddb.hashopen()`, `bsddb.btopen()` o `bsddb.rnopen()`. Los parámetros opcionales *protocol*, *writeback* y *keyencoding* tienen la misma interpretación que para la clase `Shelf`.

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

Para resumir la interfaz (`key` es una cadena de caracteres, `data` es un objeto arbitrario):

```
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.
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 construidos 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).

Lanza un *evento de auditoría* marshal.dumps con argumentos value, version.

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

[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.

excepción *EOFError*, *ValueError* o *TypeError*. El archivo debe ser un *binary file* habilitado para lectura.

Lanza un *evento de auditoría* `marshal.load` sin argumentos.

Nota: Si un objeto que contiene un tipo no admitido se calcula con *dump()*, *load()* sustituirá `None` por el tipo «unmarshallable».

Distinto en la versión 3.10: Esta llamada solía lanzar un evento de auditoría `code.__new__` para cada objeto código. Ahora lanza un único evento `marshal.load` para toda la operación de carga.

`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).

Lanza un *evento de auditoría* `marshal.dumps` con argumentos `value`, `version`.

`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.

Lanza un *evento de auditoría* `marshal.loads` con argumento `bytes`.

Distinto en la versión 3.10: Esta llamada solía lanzar un evento de auditoría `code.__new__` para cada objeto código. Ahora lanza un único evento `marshal.loads` para toda la operación de carga.

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:

Valor	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 is not 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 bm.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.dymb 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.

The `sqlite3` module was written by Gerhard Häring. It provides an SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**, and requires SQLite 3.7.15 or newer.

This document includes four main sections:

- *Tutorial* teaches how to use the `sqlite3` module.
- *Reference* describes the classes and functions this module defines.
- *How-to guides* details how to handle specific tasks.
- *Explanation* provides in-depth background on transaction control.

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 Tutorial

In this tutorial, you will create a database of Monty Python movies using basic `sqlite3` functionality. It assumes a fundamental understanding of database concepts, including cursors and transactions.

First, we need to create a new database and open a database connection to allow `sqlite3` to work with it. Call *`sqlite3.connect()`* to create a connection to the database `tutorial.db` in the current working directory, implicitly creating it if it does not exist:

```
import sqlite3
con = sqlite3.connect("tutorial.db")
```

The returned *`Connection`* object `con` represents the connection to the on-disk database.

In order to execute SQL statements and fetch results from SQL queries, we will need to use a database cursor. Call *`con.cursor()`* to create the *`Cursor`*:

```
cur = con.cursor()
```

Now that we've got a database connection and a cursor, we can create a database table `movie` with columns for title, release year, and review score. For simplicity, we can just use column names in the table declaration – thanks to the flexible typing feature of SQLite, specifying the data types is optional. Execute the `CREATE TABLE` statement by calling *`cur.execute(...)`*:

```
cur.execute("CREATE TABLE movie(title, year, score)")
```

We can verify that the new table has been created by querying the `sqlite_master` table built-in to SQLite, which should now contain an entry for the `movie` table definition (see The Schema Table for details). Execute that query by calling *`cur.execute(...)`*, assign the result to `res`, and call *`res.fetchone()`* to fetch the resulting row:

```
>>> res = cur.execute("SELECT name FROM sqlite_master")
>>> res.fetchone()
('movie',)
```

We can see that the table has been created, as the query returns a *tuple* containing the table's name. If we query `sqlite_master` for a non-existent table `spam`, `res.fetchone()` will return `None`:

```
>>> res = cur.execute("SELECT name FROM sqlite_master WHERE name='spam'")
>>> res.fetchone() is None
True
```

Now, add two rows of data supplied as SQL literals by executing an `INSERT` statement, once again by calling *cur.execute(...)*:

```
cur.execute("""
    INSERT INTO movie VALUES
        ('Monty Python and the Holy Grail', 1975, 8.2),
        ('And Now for Something Completely Different', 1971, 7.5)
""")
```

The `INSERT` statement implicitly opens a transaction, which needs to be committed before changes are saved in the database (see *Transaction control* for details). Call *con.commit()* on the connection object to commit the transaction:

```
con.commit()
```

We can verify that the data was inserted correctly by executing a `SELECT` query. Use the now-familiar *cur.execute(...)* to assign the result to `res`, and call *res.fetchall()* to return all resulting rows:

```
>>> res = cur.execute("SELECT score FROM movie")
>>> res.fetchall()
[(8.2,), (7.5,)]
```

The result is a *list* of two `tuples`, one per row, each containing that row's `score` value.

Now, insert three more rows by calling *cur.executemany(...)*:

```
data = [
    ("Monty Python Live at the Hollywood Bowl", 1982, 7.9),
    ("Monty Python's The Meaning of Life", 1983, 7.5),
    ("Monty Python's Life of Brian", 1979, 8.0),
]
cur.executemany("INSERT INTO movie VALUES(?, ?, ?)", data)
con.commit()  # Remember to commit the transaction after executing INSERT.
```

Notice that `?` placeholders are used to bind `data` to the query. Always use placeholders instead of string formatting to bind Python values to SQL statements, to avoid SQL injection attacks (see *How to use placeholders to bind values in SQL queries* for more details).

We can verify that the new rows were inserted by executing a `SELECT` query, this time iterating over the results of the query:

```
>>> for row in cur.execute("SELECT year, title FROM movie ORDER BY year"):
...     print(row)
(1971, 'And Now for Something Completely Different')
(1975, 'Monty Python and the Holy Grail')
(1979, "Monty Python's Life of Brian")
(1982, 'Monty Python Live at the Hollywood Bowl')
(1983, "Monty Python's The Meaning of Life")
```

Each row is a two-item *tuple* of `(year, title)`, matching the columns selected in the query.

Finally, verify that the database has been written to disk by calling *con.close()* to close the existing connection, opening a new one, creating a new cursor, then querying the database:

```
>>> con.close()
>>> new_con = sqlite3.connect("tutorial.db")
>>> new_cur = new_con.cursor()
>>> res = new_cur.execute("SELECT title, year FROM movie ORDER BY score DESC")
>>> title, year = res.fetchone()
>>> print(f'The highest scoring Monty Python movie is {title!r}, released in {year}
↪')
The highest scoring Monty Python movie is 'Monty Python and the Holy Grail',␣
↪released in 1975
```

You've now created an SQLite database using the `sqlite3` module, inserted data and retrieved values from it in multiple ways.

Ver también:

- *How-to guides* for further reading:
  - *How to use placeholders to bind values in SQL queries*
  - *How to adapt custom Python types to SQLite values*
  - *How to convert SQLite values to custom Python types*
  - *How to use the connection context manager*
  - *How to create and use row factories*
- *Explanation* for in-depth background on transaction control.

12.6.2 Reference

Module functions

sqlite3.**connect**(*database*, *timeout=5.0*, *detect_types=0*, *isolation_level='DEFERRED'*, *check_same_thread=True*, *factory=sqlite3.Connection*, *cached_statements=100*, *uri=False*)

Open a connection to an SQLite database.

Parámetros

- **database** (*path-like object*) – The path to the database file to be opened. Pass `":memory:"` to open a connection to a database that is in RAM instead of on disk.
- **timeout** (`float`) – How many seconds the connection should wait before raising an *`OperationalError`* when a table is locked. If another connection opens a transaction to modify a table, that table will be locked until the transaction is committed. Default five seconds.
- **detect_types** (`int`) – Control whether and how data types not *natively supported by SQLite* are looked up to be converted to Python types, using the converters registered with *`register_converter()`*. Set it to any combination (using `|`, bitwise or) of *`PARSE_DECLTYPES`* and *`PARSE_COLNAMES`* to enable this. Column names takes precedence over declared types if both flags are set. Types cannot be detected for generated fields (for example `max(data)`), even when the *detect_types* parameter is set; *`str`* will be returned instead. By default (`0`), type detection is disabled.
- **isolation_level** (*`str | None`*) – The *`isolation_level`* of the connection, controlling whether and how transactions are implicitly opened. Can be `"DEFERRED"` (default), `"EXCLUSIVE"` or `"IMMEDIATE"`; or `None` to disable opening transactions implicitly. See *Transaction control* for more.
- **check_same_thread** (`bool`) – If `True` (default), *`ProgrammingError`* will be raised if the database connection is used by a thread other than the one that created it. If `False`, the connection may be accessed in multiple threads; write operations may need

to be serialized by the user to avoid data corruption. See *threadsafety* for more information.

- **factory** (Connection) – A custom subclass of *Connection* to create the connection with, if not the default *Connection* class.
- **cached_statements** (int) – The number of statements that sqlite3 should internally cache for this connection, to avoid parsing overhead. By default, 100 statements.
- **uri** (bool) – If set to True, *database* is interpreted as a URI (Uniform Resource Identifier) with a file path and an optional query string. The scheme part *must* be "file:", and the path can be relative or absolute. The query string allows passing parameters to SQLite, enabling various *How to work with SQLite URIs*.

Tipo del valor devuelto *Connection*

Lanza un *evento de auditoría* sqlite3.connect con argumento database.

Lanza un *auditing event* sqlite3.connect/handle con el argumento connection_handle.

Nuevo en la versión 3.4: The *uri* parameter.

Distinto en la versión 3.7: *database* ahora también puede ser un *path-like object*, no solo una cadena de caracteres.

Nuevo en la versión 3.10: The sqlite3.connect/handle auditing event.

sqlite3.**complete_statement**(*statement*)

Return True if the string *statement* appears to contain one or more complete SQL statements. No syntactic verification or parsing of any kind is performed, other than checking that there are no unclosed string literals and the statement is terminated by a semicolon.

For example:

```
>>> sqlite3.complete_statement("SELECT foo FROM bar;")
True
>>> sqlite3.complete_statement("SELECT foo")
False
```

This function may be useful during command-line input to determine if the entered text seems to form a complete SQL statement, or if additional input is needed before calling *execute()*.

sqlite3.**enable_callback_tracebacks**(*flag*, /)

Enable or disable callback tracebacks. By default you will not get any tracebacks in user-defined functions, aggregates, converters, authorizer callbacks etc. If you want to debug them, you can call this function with *flag* set to True. Afterwards, you will get tracebacks from callbacks on sys.stderr. Use False to disable the feature again.

sqlite3.**register_adapter**(*type*, *adapter*, /)

Register an *adapter* callable to adapt the Python type *type* into an SQLite type. The adapter is called with a Python object of type *type* as its sole argument, and must return a value of a *type that SQLite natively understands*.

sqlite3.**register_converter**(*typename*, *converter*, /)

Register the *converter* callable to convert SQLite objects of type *typename* into a Python object of a specific type. The converter is invoked for all SQLite values of type *typename*; it is passed a *bytes* object and should return an object of the desired Python type. Consult the parameter *detect_types* of *connect()* for information regarding how type detection works.

Note: *typename* and the name of the type in your query are matched case-insensitively.

Module constants

sqlite3.**PARSE_COLNAMES**
Pass this flag value to the *detect_types* parameter of *connect()* to look up a converter function by using the type name, parsed from the query column name, as the converter dictionary key. The type name must be wrapped in square brackets (`[]`).

```
SELECT p as "p [point]" FROM test;  ! will look up converter "point"
```

This flag may be combined with *PARSE_DECLTYPES* using the `|` (bitwise or) operator.

sqlite3.**PARSE_DECLTYPES**
Pass this flag value to the *detect_types* parameter of *connect()* to look up a converter function using the declared types for each column. The types are declared when the database table is created. `sqlite3` will look up a converter function using the first word of the declared type as the converter dictionary key. For example:

```
CREATE TABLE test(
   i integer primary key,   ! will look up a converter named "integer"
   p point,                 ! will look up a converter named "point"
   n number(10)             ! will look up a converter named "number"
 )
```

This flag may be combined with *PARSE_COLNAMES* using the `|` (bitwise or) operator.

sqlite3.**SQLITE_OK**
sqlite3.**SQLITE_DENY**
sqlite3.**SQLITE_IGNORE**
Flags that should be returned by the *authorizer_callback* callable passed to *Connection.set_authorizer()*, to indicate whether:

- Access is allowed (`SQLITE_OK`),
- The SQL statement should be aborted with an error (`SQLITE_DENY`)
- The column should be treated as a `NULL` value (`SQLITE_IGNORE`)

sqlite3.**apilevel**
String constant stating the supported DB-API level. Required by the DB-API. Hard-coded to `"2.0"`.

sqlite3.**paramstyle**
String constant stating the type of parameter marker formatting expected by the `sqlite3` module. Required by the DB-API. Hard-coded to `"qmark"`.

Nota: The `named` DB-API parameter style is also supported.

---

sqlite3.**sqlite_version**
Version number of the runtime SQLite library as a *string*.

sqlite3.**sqlite_version_info**
Version number of the runtime SQLite library as a *tuple* of *integers*.

sqlite3.**threadsafety**
Integer constant required by the DB-API, stating the level of thread safety the `sqlite3` module supports. Currently hard-coded to `1`, meaning *«Threads may share the module, but not connections.»* However, this may not always be true. You can check the underlying SQLite library's compile-time threaded mode using the following query:

```
import sqlite3
con = sqlite3.connect(":memory:")
con.execute("""
    select * from pragma_compile_options
    where compile_options like 'THREADSAFE=%'
""").fetchall()
```

Note that the SQLITE_THREADSAFE levels do not match the DB-API 2.0 `threadsafety` levels.

`sqlite3.`**`version`**
Version number of this module as a *`string`*. This is not the version of the SQLite library.

`sqlite3.`**`version_info`**
Version number of this module as a *`tuple`* of *`integers`*. This is not the version of the SQLite library.

Connection objects

`class` `sqlite3.`**`Connection`**
Each open SQLite database is represented by a `Connection` object, which is created using *`sqlite3.connect()`*. Their main purpose is creating *`Cursor`* objects, and *Transaction control*.

Ver también:

- *How to use connection shortcut methods*
- *How to use the connection context manager*

An SQLite database connection has the following attributes and methods:

`cursor`(*factory=Cursor*)
Create and return a *`Cursor`* object. The cursor method accepts a single optional parameter *factory*. If supplied, this must be a callable returning an instance of *`Cursor`* or its subclasses.

`commit`()
Commit any pending transaction to the database. If there is no open transaction, this method is a no-op.

`rollback`()
Roll back to the start of any pending transaction. If there is no open transaction, this method is a no-op.

`close`()
Close the database connection. Any pending transaction is not committed implicitly; make sure to *`commit()`* before closing to avoid losing pending changes.

`execute`(*sql*, *parameters=()*, */*)
Create a new *`Cursor`* object and call *`execute()`* on it with the given *sql* and *parameters*. Return the new cursor object.

`executemany`(*sql*, *parameters*, */*)
Create a new *`Cursor`* object and call *`executemany()`* on it with the given *sql* and *parameters*. Return the new cursor object.

`executescript`(*sql_script*, */*)
Create a new *`Cursor`* object and call *`executescript()`* on it with the given *sql_script*. Return the new cursor object.

`create_function`(*name*, *narg*, *func*, *, *deterministic=False*)
Create or remove a user-defined SQL function.

Parámetros

- **`name`** (`str`) – The name of the SQL function.
- **`narg`** (`int`) – The number of arguments the SQL function can accept. If `-1`, it may take any number of arguments.
- **`func`** (*callback* | None) – A callable that is called when the SQL function is invoked. The callable must return *a type natively supported by SQLite*. Set to `None` to remove an existing SQL function.
- **`deterministic`** (`bool`) – If `True`, the created SQL function is marked as deterministic, which allows SQLite to perform additional optimizations.

Muestra ***NotSupportedError*** – If *deterministic* is used with SQLite versions older than 3.8.3.

Nuevo en la versión 3.8: The *deterministic* parameter.

Ejemplo:

```
>>> import hashlib
>>> def md5sum(t):
...     return hashlib.md5(t).hexdigest()
>>> con = sqlite3.connect(":memory:")
>>> con.create_function("md5", 1, md5sum)
>>> for row in con.execute("SELECT md5(?)", (b"foo",)):
...     print(row)
('acbd18db4cc2f85cedef654fccc4a4d8',)
```

create_aggregate(*name*, */*, *n_arg*, *aggregate_class*)

Create or remove a user-defined SQL aggregate function.

Parámetros

- **name** (str) – The name of the SQL aggregate function.
- **n_arg** (int) – The number of arguments the SQL aggregate function can accept. If -1, it may take any number of arguments.
- **aggregate_class** (*class* | None) – A class must implement the following methods:
  - step(): Add a row to the aggregate.
  - finalize(): Return the final result of the aggregate as *a type natively supported by SQLite*.

  The number of arguments that the step() method must accept is controlled by *n_arg*.

  Set to None to remove an existing SQL aggregate function.

Ejemplo:

```
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.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])

con.close()
```

create_collation(*name*, *callable*)

Create a collation named *name* using the collating function *callable*. *callable* is passed two *string* arguments, and it should return an *integer*:

- 1 if the first is ordered higher than the second
- -1 if the first is ordered lower than the second
- 0 if they are ordered equal

The following example shows a reverse sorting collation:

```
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.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()
```

Remove a collation function by setting *callable* to `None`.

interrupt()
Call this method from a different thread to abort any queries that might be executing on the connection. Aborted queries will raise an *OperationalError*.

set_authorizer(*authorizer_callback*)
Register callable *authorizer_callback* to be invoked for each attempt to access a column of a table in the database. The callback should return one of *SQLITE_OK*, *SQLITE_DENY*, or *SQLITE_IGNORE* to signal how access to the column should be handled by the underlying SQLite library.

The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or `None` depending on the first argument. The 4th argument is the name of the database («main», «temp», etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or `None` if this access attempt is directly from input SQL code.

Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the `sqlite3` module.

set_progress_handler(*progress_handler*, *n*)
Register callable *progress_handler* to be invoked for every *n* instructions of the SQLite virtual machine. This is useful if you want to get called from SQLite during long-running operations, for example to update a GUI.

If you want to clear any previously installed progress handler, call the method with `None` for *progress_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*)
Register callable *trace_callback* to be invoked for each SQL statement that is actually executed by the SQLite backend.

The only argument passed to the callback is the statement (as *str*) that is being executed. The return value of the callback is ignored. Note that the backend does not only run statements passed to the *Cursor.execute()* methods. Other sources include the *transaction management* of the `sqlite3` module and the execution of triggers defined in the current database.

Passing `None` as *trace_callback* will disable the trace callback.

Nota: Las excepciones que se producen en la llamada de retorno no se propagan. Como ayuda para el

desarrollo y la depuración, utilice *enable_callback_tracebacks()* para habilitar la impresión de las trazas de las excepciones que se producen en la llamada de retorno.

Nuevo en la versión 3.3.

enable_load_extension(*enabled*, /)
Enable the SQLite engine to load SQLite extensions from shared libraries if *enabled* is `True`; else, disallow loading SQLite extensions. SQLite extensions can define new functions, aggregates or whole new virtual table implementations. One well-known extension is the fulltext-search extension distributed with SQLite.

Nota: The `sqlite3` module is not built with loadable extension support by default, because some platforms (notably macOS) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must pass the `--enable-loadable-sqlite-extensions` option to **configure**.

Lanza un *auditing event* `sqlite3.enable_load_extension` con argumentos `connection`, `enabled`.

Nuevo en la versión 3.2.

Distinto en la versión 3.10: Añadido el evento de auditoría `sqlite3.enable_load_extension`

```
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'"):
    print(row)

con.close()
```

load_extension(*path*, /)
Load an SQLite extension from a shared library located at *path*. Enable extension loading with *enable_load_extension()* before calling this method.

Lanza un *auditing event* `sqlite3.load_extension` con argumentos `connection`, `path`.

Nuevo en la versión 3.2.

Distinto en la versión 3.10: Añadido el evento de auditoría `sqlite3.load_extension`

iterdump()
Return an *iterator* to dump the database as SQL source code. Useful when saving an in-memory database for later restoration. Similar to the `.dump` command in the **sqlite3** shell.

Ejemplo:

```
# Convert file example.db to SQL dump file dump.sql
con = sqlite3.connect('example.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*)
Create a backup of an SQLite database.

Works even if the database is being accessed by other clients or concurrently by the same connection.

Parámetros

- **target** (`Connection`) – The database connection to save the backup to.
- **pages** (`int`) – The number of pages to copy at a time. If equal to or less than `0`, the entire database is copied in a single step. Defaults to `-1`.
- **progress** (*callback* | None) – If set to a callable, it is invoked with three integer arguments for every backup iteration: the *status* of the last iteration, the *remaining* number of pages still to be copied, and the *total* number of pages. Defaults to `None`.
- **name** (`str`) – The name of the database to back up. Either `"main"` (the default) for the main database, `"temp"` for the temporary database, or the name of a custom database as attached using the `ATTACH DATABASE` SQL statement.
- **sleep** (`float`) – The number of seconds to sleep between successive attempts to back up remaining pages.

Example 1, copy an existing database into another:

```
def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total} pages...')

src = sqlite3.connect('example.db')
dst = sqlite3.connect('backup.db')
with dst:
    src.backup(dst, pages=1, progress=progress)
dst.close()
src.close()
```

Example 2, copy an existing database into a transient copy:

```
src = sqlite3.connect('example.db')
dst = sqlite3.connect(':memory:')
src.backup(dst)
```

Nuevo en la versión 3.7.

in_transaction
This read-only attribute corresponds to the low-level SQLite autocommit mode.

`True` if a transaction is active (there are uncommitted changes), `False` otherwise.

Nuevo en la versión 3.2.

isolation_level
This attribute controls the *transaction handling* performed by `sqlite3`. If set to `None`, transactions are never implicitly opened. If set to one of `"DEFERRED"`, `"IMMEDIATE"`, or `"EXCLUSIVE"`, corresponding to the underlying SQLite transaction behaviour, implicit *transaction management* is performed.

If not overridden by the *isolation_level* parameter of *connect()*, the default is "", which is an alias for "DEFERRED".

row_factory
The initial *row_factory* for *Cursor* objects created from this connection. Assigning to this attribute does not affect the row_factory of existing cursors belonging to this connection, only new ones. Is None by default, meaning each row is returned as a *tuple*.

See *How to create and use row factories* for more details.

text_factory
A callable that accepts a *bytes* parameter and returns a text representation of it. The callable is invoked for SQLite values with the TEXT data type. By default, this attribute is set to *str*. If you want to return bytes instead, set *text_factory* to bytes.

Ejemplo:

```
con = sqlite3.connect(":memory:")
cur = con.cursor()

AUSTRIA = "Österreich"

# by default, rows are returned as str
cur.execute("SELECT ?", (AUSTRIA,))
row = cur.fetchone()
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
Return the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.

Cursor objects

A Cursor object represents a database cursor which is used to execute SQL statements, and manage the context of a fetch operation. Cursors are created using *Connection.cursor()*, or by using any of the *connection shortcut methods*.

Cursor objects are *iterators*, meaning that if you *execute()* a SELECT query, you can simply iterate over the cursor to fetch the resulting rows:

```
for row in cur.execute("SELECT t FROM data"):
    print(row)
```

class sqlite3.**Cursor**

Una instancia de *Cursor* tiene los siguientes atributos y métodos.

execute(*sql*, *parameters=()*, */*)

Execute SQL a single SQL statement, optionally binding Python values using *placeholders*.

Parámetros

- **sql** (str) – A single SQL statement.
- **parameters** (*dict* | *sequence*) – Python values to bind to placeholders in *sql*. A dict if named placeholders are used. A sequence if unnamed placeholders are used. See *How to use placeholders to bind values in SQL queries*.

Muestra ***Warning*** – If *sql* contains more than one SQL statement.

If *isolation_level* is not None, *sql* is an INSERT, UPDATE, DELETE, or REPLACE statement, and there is no open transaction, a transaction is implicitly opened before executing *sql*.

Use *executescript()* to execute multiple SQL statements.

executemany(*sql*, *parameters*, */*)

For every item in *parameters*, repeatedly execute the *parameterized* SQL statement *sql*.

Uses the same implicit transaction handling as *execute()*.

Parámetros

- **sql** (str) – A single SQL DML (Data Manipulation Language) statement.
- **parameters** (*iterable*) – An iterable of parameters to bind with the placeholders in *sql*. See *How to use placeholders to bind values in SQL queries*.

Muestra

- ***ProgrammingError*** – If *sql* is not a DML statment.
- ***Warning*** – If *sql* contains more than one SQL statement.

Ejemplo:

```
rows = [
    ("row1",),
    ("row2",),
]
# cur is an sqlite3.Cursor object
cur.executemany("INSERT INTO data VALUES(?)", rows)
```

executescript(*sql_script*, */*)

Execute the SQL statements in *sql_script*. If there is a pending transaction, an implicit COMMIT statement is executed first. No other implicit transaction control is performed; any transaction control must be added to *sql_script*.

sql_script must be a *string*.

Ejemplo:

```
# cur is an sqlite3.Cursor object
cur.executescript("""
    BEGIN;
    CREATE TABLE person(firstname, lastname, age);
    CREATE TABLE book(title, author, published);
    CREATE TABLE publisher(name, address);
    COMMIT;
""")
```

fetchone()

If *row_factory* is None, return the next row query result set as a *tuple*. Else, pass it to the row factory and return its result. Return None if no more data is available.

fetchmany(*size=cursor.arraysize*)
Return the next set of rows of a query result as a *list*. Return an empty list if no more rows are available.

The number of rows to fetch per call is specified by the *size* parameter. If *size* is not given, *arraysize* determines the number of rows to be fetched. If fewer than *size* rows are available, as many rows as are available are returned.

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()
Return all (remaining) rows of a query result as a *list*. Return an empty list if no rows are available. Note that the *arraysize* attribute can affect the performance of this operation.

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.

setinputsizes(*sizes*, /)
Required by the DB-API. Does nothing in sqlite3.

setoutputsize(*size*, *column=None*, /)
Required by the DB-API. Does nothing in sqlite3.

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.

connection
Read-only attribute that provides the SQLite database *Connection* belonging to the cursor. A *Cursor* object created by calling *con.cursor()* will have a *connection* attribute that refers to *con*:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```

description
Read-only attribute that provides the column names of the last query. To remain compatible with the Python DB API, it returns a 7-tuple for each column where the last six items of each tuple are None.

También es configurado para sentencias SELECT sin ninguna fila coincidente.

lastrowid
Read-only attribute that provides the row id of the last inserted row. It is only updated after successful INSERT or REPLACE statements using the *execute()* method. For other statements, after *executemany()* or *executescript()*, or if the insertion failed, the value of lastrowid is left unchanged. The initial value of lastrowid is None.

Nota: Inserts into WITHOUT ROWID tables are not recorded.

Distinto en la versión 3.6: Se agregó soporte para sentencias REPLACE.

rowcount
Read-only attribute that provides the number of modified rows for INSERT, UPDATE, DELETE, and REPLACE statements; is -1 for other statements, including CTE (Common Table Expression) queries. It is only updated by the *execute()* and *executemany()* methods.

row_factory
Control how a row fetched from this Cursor is represented. If None, a row is represented as a *tuple*.

Can be set to the included *sqlite3.Row*; or a *callable* that accepts two arguments, a *Cursor* object and the tuple of row values, and returns a custom object representing an SQLite row.

Defaults to what *Connection.row_factory* was set to when the Cursor was created. Assigning to this attribute does not affect *Connection.row_factory* of the parent connection.

See *How to create and use row factories* for more details.

Row objects

class sqlite3.**Row**

A Row instance serves as a highly optimized *row_factory* for *Connection* objects. It supports iteration, equality testing, *len()*, and *mapping* access by column name and index.

Two Row objects compare equal if they have identical column names and values.

See *How to create and use row factories* for more details.

keys()

Return a *list* of column names as *strings*. Immediately after a query, it is the first member of each tuple in *Cursor.description*.

Distinto en la versión 3.5: Agrega soporte de segmentación.

PrepareProtocol objects

class sqlite3.**PrepareProtocol**

The PrepareProtocol type's single purpose is to act as a **PEP 246** style adaption protocol for objects that can *adapt themselves* to *native SQLite types*.

Excepciones

The exception hierarchy is defined by the DB-API 2.0 (**PEP 249**).

exception sqlite3.**Warning**

This exception is raised by sqlite3 if an SQL query is not a *string*, or if multiple statements are passed to *execute()* or *executemany()*. Warning is a subclass of *Exception*.

exception sqlite3.**Error**

The base class of the other exceptions in this module. Use this to catch all errors with one single except statement. Error is a subclass of *Exception*.

exception sqlite3.**InterfaceError**

This exception is raised by sqlite3 for fetch across rollback, or if sqlite3 is unable to bind parameters. InterfaceError is a subclass of *Error*.

exception sqlite3.**DatabaseError**

Exception raised for errors that are related to the database. This serves as the base exception for several types of database errors. It is only raised implicitly through the specialised subclasses. DatabaseError is a subclass of *Error*.

exception sqlite3.**DataError**

Exception raised for errors caused by problems with the processed data, like numeric values out of range, and strings which are too long. DataError is a subclass of *DatabaseError*.

exception sqlite3.**OperationalError**

Exception raised for errors that are related to the database's operation, and not necessarily under the control of the programmer. For example, the database path is not found, or a transaction could not be processed. OperationalError is a subclass of *DatabaseError*.

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.**InternalError**
Exception raised when SQLite encounters an internal error. If this is raised, it may indicate that there is a problem with the runtime SQLite library. InternalError is a subclass of *DatabaseError*.

exception sqlite3.**ProgrammingError**
Exception raised for sqlite3 API programming errors, for example trying to operate on a closed *Connection*, or trying to execute non-DML statements with *executemany()*. ProgrammingError is a subclass of *DatabaseError*.

exception sqlite3.**NotSupportedError**
Exception raised in case a method or database API is not supported by the underlying SQLite library. For example, setting *deterministic* to True in *create_function()*, if the underlying SQLite library does not support deterministic functions. NotSupportedError is a subclass of *DatabaseError*.

SQLite y tipos de Python

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
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	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	None
INTEGER	*int*
REAL	*float*
TEXT	depende de *text_factory*, *str* por defecto
BLOB	*bytes*

The type system of the sqlite3 module is extensible in two ways: you can store additional Python types in an SQLite database via *object adapters*, and you can let the sqlite3 module convert SQLite types to Python types via *converters*.

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*.

Nota: The default «timestamp» converter ignores UTC offsets in the database and always returns a naive *datetime.datetime* object. To preserve UTC offsets in timestamps, either leave converters disabled, or register an offset-aware converter with *register_converter()*.

12.6.3 How-to guides

How to use placeholders to bind values in SQL queries

SQL operations usually need to use values from Python variables. However, beware of using Python's string operations to assemble queries, as they are vulnerable to SQL injection attacks. For example, an attacker can simply close the single quote and inject `OR TRUE` to select all rows:

```
>>> # Never do this -- insecure!
>>> symbol = input()
' OR TRUE; --
>>> sql = "SELECT * FROM stocks WHERE symbol = '%s'" % symbol
>>> print(sql)
SELECT * FROM stocks WHERE symbol = '' OR TRUE; --'
>>> cur.execute(sql)
```

Instead, use the DB-API's parameter substitution. To insert a variable into a query string, use a placeholder in the string, and substitute the actual values into the query by providing them as a *tuple* of values to the second argument of the cursor's *execute()* method.

An SQL statement may use one of two kinds of placeholders: question marks (qmark style) or named placeholders (named style). For the qmark style, *parameters* must be a *sequence* whose length must match the number of placeholders, or a *ProgrammingError* is raised. For the named style, *parameters* should be an instance of a *dict* (or a subclass), which must contain keys for all named parameters; any extra items are ignored. Here's an example of both styles:

```
con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE lang(name, first_appeared)")

# This is the named style used with executemany():
data = (
    {"name": "C", "year": 1972},
    {"name": "Fortran", "year": 1957},
    {"name": "Python", "year": 1991},
    {"name": "Go", "year": 2009},
)
cur.executemany("INSERT INTO lang VALUES(:name, :year)", data)

# This is the qmark style used in a SELECT query:
params = (1972,)
cur.execute("SELECT * FROM lang WHERE first_appeared = ?", params)
print(cur.fetchall())
```

Nota: **PEP 249** numeric placeholders are *not* supported. If used, they will be interpreted as named placeholders.

How to adapt custom Python types to SQLite values

SQLite supports only a limited set of data types natively. To store custom Python types in SQLite databases, *adapt* them to one of the *Python types SQLite natively understands*.

There are two ways to adapt Python objects to SQLite types: letting your object adapt itself, or using an *adapter callable*. The latter will take precedence above the former. For a library that exports a custom type, it may make sense to enable that type to adapt itself. As an application developer, it may make more sense to take direct control by registering custom adapter functions.

How to write adaptable objects

Suppose we have a `Point` class that represents a pair of coordinates, `x` and `y`, in a Cartesian coordinate system. The coordinate pair will be stored as a text string in the database, using a semicolon to separate the coordinates. This can be implemented by adding a `__conform__(self, protocol)` method which returns the adapted value. The object passed to *protocol* will be of type *`PrepareProtocol`*.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return f"{self.x};{self.y}"

con = sqlite3.connect(":memory:")
cur = con.cursor()

cur.execute("SELECT ?", (Point(4.0, -3.2),))
print(cur.fetchone()[0])
```

How to register adapter callables

The other possibility is to create a function that converts the Python object to an SQLite-compatible type. This function can then be registered using *register_adapter()*.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return f"{point.x};{point.y}"

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

cur.execute("SELECT ?", (Point(1.0, 2.5),))
print(cur.fetchone()[0])
```

How to convert SQLite values to custom Python types

Writing an adapter lets you convert *from* custom Python types *to* SQLite values. To be able to convert *from* SQLite values *to* custom Python types, we use *converters*.

Let's go back to the `Point` class. We stored the x and y coordinates separated via semicolons as strings in SQLite.

First, we'll define a converter function that accepts the string as a parameter and constructs a `Point` object from it.

Nota: Converter functions are **always** passed a *bytes* object, no matter the underlying SQLite data type.

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

We now need to tell `sqlite3` when it should convert a given SQLite value. This is done when connecting to a database, using the *detect_types* parameter of *connect()*. There are three options:

- Implicit: set *detect_types* to *PARSE_DECLTYPES*
- Explicit: set *detect_types* to *PARSE_COLNAMES*
- Both: set *detect_types* to `sqlite3.PARSE_DECLTYPES | sqlite3.PARSE_COLNAMES`. Column names take precedence over declared types.

The following example illustrates the implicit and explicit approaches:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

def adapt_point(point):
    return f"{point.x};{point.y}"

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)
```

(continué en la próxima página)

(proviene de la página anterior)

```
# Register the adapter and converter
sqlite3.register_adapter(Point, adapt_point)
sqlite3.register_converter("point", convert_point)

# 1) Parse using declared types
p = Point(4.0, -3.2)
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.execute("CREATE TABLE test(p point)")

cur.execute("INSERT INTO test(p) VALUES(?)", (p,))
cur.execute("SELECT p FROM test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

# 2) Parse using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.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])
```

Adapter and converter recipes

This section shows recipes for common adapters and converters.

```
import datetime
import sqlite3

def adapt_date_iso(val):
    """Adapt datetime.date to ISO 8601 date."""
    return val.isoformat()

def adapt_datetime_iso(val):
    """Adapt datetime.datetime to timezone-naive ISO 8601 date."""
    return val.isoformat()

def adapt_datetime_epoch(val):
    """Adapt datetime.datetime to Unix timestamp."""
    return int(val.timestamp())

sqlite3.register_adapter(datetime.date, adapt_date_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_epoch)

def convert_date(val):
    """Convert ISO 8601 date to datetime.date object."""
    return datetime.date.fromisoformat(val.decode())

def convert_datetime(val):
    """Convert ISO 8601 datetime to datetime.datetime object."""
    return datetime.datetime.fromisoformat(val.decode())

def convert_timestamp(val):
    """Convert Unix epoch timestamp to datetime.datetime object."""
    return datetime.datetime.fromtimestamp(int(val))

sqlite3.register_converter("date", convert_date)
```

(continué en la próxima página)

(proviene de la página anterior)

```
sqlite3.register_converter("datetime", convert_datetime)
sqlite3.register_converter("timestamp", convert_timestamp)
```

How to use connection shortcut methods

Using the *execute()*, *executemany()*, and *executescript()* methods of the *Connection* class, your code can be written more concisely because you don't have to create the (often superfluous) *Cursor* objects explicitly. Instead, the *Cursor* objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a `SELECT` statement and iterate over it directly using only a single call on the *Connection* object.

```
# Create and fill the table.
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(name, first_appeared)")
data = [
    ("C++", 1985),
    ("Objective-C", 1984),
]
con.executemany("INSERT INTO lang(name, first_appeared) VALUES(?, ?)", data)

# Print the table contents
for row in con.execute("SELECT name, first_appeared FROM lang"):
    print(row)

print("I just deleted", con.execute("DELETE FROM lang").rowcount, "rows")

# close() is not a shortcut method and it's not called automatically;
# the connection object should be closed manually
con.close()
```

How to use the connection context manager

A *Connection* object can be used as a context manager that automatically commits or rolls back open transactions when leaving the body of the context manager. If the body of the `with` statement finishes without exceptions, the transaction is committed. If this commit fails, or if the body of the `with` statement raises an uncaught exception, the transaction is rolled back.

If there is no open transaction upon leaving the body of the `with` statement, the context manager is a no-op.

Nota: The context manager neither implicitly opens a new transaction nor closes the connection.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(id INTEGER PRIMARY KEY, name VARCHAR UNIQUE)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))

# 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 lang(name) VALUES(?)", ("Python",))
except sqlite3.IntegrityError:
    print("couldn't add Python twice")

# Connection object used as context manager only commits or rollbacks transactions,
```

(continué en la próxima página)

(proviene de la página anterior)

```
# so the connection object should be closed manually
con.close()
```

How to work with SQLite URIs

Some useful URI tricks include:

- Open a database in read-only mode:

```
>>> con = sqlite3.connect("file:tutorial.db?mode=ro", uri=True)
>>> con.execute("CREATE TABLE readonly(data)")
Traceback (most recent call last):
OperationalError: attempt to write a readonly database
```

- Do not implicitly create a new database file if it does not already exist; will raise *OperationalError* if unable to create a new file:

```
>>> con = sqlite3.connect("file:nosuchdb.db?mode=rw", uri=True)
Traceback (most recent call last):
OperationalError: unable to open database file
```

- Create a shared named in-memory database:

```
db = "file:mem1?mode=memory&cache=shared"
con1 = sqlite3.connect(db, uri=True)
con2 = sqlite3.connect(db, uri=True)
with con1:
    con1.execute("CREATE TABLE shared(data)")
    con1.execute("INSERT INTO shared VALUES(28)")
res = con2.execute("SELECT data FROM shared")
assert res.fetchone() == (28,)
```

More information about this feature, including a list of parameters, can be found in the SQLite URI documentation.

How to create and use row factories

By default, `sqlite3` represents each row as a *tuple*. If a `tuple` does not suit your needs, you can use the *sqlite3.Row* class or a custom *row_factory*.

While `row_factory` exists as an attribute both on the *Cursor* and the *Connection*, it is recommended to set *Connection.row_factory*, so all cursors created from the connection will use the same row factory.

`Row` provides indexed and case-insensitive named access to columns, with minimal memory overhead and performance impact over a `tuple`. To use `Row` as a row factory, assign it to the `row_factory` attribute:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = sqlite3.Row
```

Queries now return `Row` objects:

```
>>> res = con.execute("SELECT 'Earth' AS name, 6378 AS radius")
>>> row = res.fetchone()
>>> row.keys()
['name', 'radius']
>>> row[0]          # Access by index.
'Earth'
>>> row["name"]     # Access by name.
'Earth'
>>> row["RADIUS"]   # Column names are case-insensitive.
6378
```

You can create a custom *row_factory* that returns each row as a *dict*, with column names mapped to values:

```
def dict_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    return {key: value for key, value in zip(fields, row)}
```

Using it, queries now return a `dict` instead of a `tuple`:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = dict_factory
>>> for row in con.execute("SELECT 1 AS a, 2 AS b"):
...     print(row)
{'a': 1, 'b': 2}
```

The following row factory returns a *named tuple*:

```
from collections import namedtuple

def namedtuple_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    cls = namedtuple("Row", fields)
    return cls._make(row)
```

`namedtuple_factory()` can be used as follows:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = namedtuple_factory
>>> cur = con.execute("SELECT 1 AS a, 2 AS b")
>>> row = cur.fetchone()
>>> row
Row(a=1, b=2)
>>> row[0]  # Indexed access.
1
>>> row.b   # Attribute access.
2
```

With some adjustments, the above recipe can be adapted to use a *dataclass*, or any other custom class, instead of a *namedtuple*.

12.6.4 Explanation

Transaction control

The `sqlite3` module does not adhere to the transaction handling recommended by **PEP 249**.

If the connection attribute *isolation_level* is not `None`, new transactions are implicitly opened before *execute()* and *executemany()* executes `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statements; for other statements, no implicit transaction handling is performed. Use the *commit()* and *rollback()* methods to respectively commit and roll back pending transactions. You can choose the underlying SQLite transaction behaviour — that is, whether and what type of `BEGIN` statements `sqlite3` implicitly executes – via the *isolation_level* attribute.

If *isolation_level* is set to `None`, no transactions are implicitly opened at all. This leaves the underlying SQLite library in autocommit mode, but also allows the user to perform their own transaction handling using explicit SQL statements. The underlying SQLite library autocommit mode can be queried using the *in_transaction* attribute.

The *executescript()* method implicitly commits any pending transaction before execution of the given SQL script, regardless of the value of *isolation_level*.

Distinto en la versión 3.6: `sqlite3` used to implicitly commit an open transaction before DDL statements. This is no longer the case.

CAPÍTULO 13

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 aplicaciones que requieren compresión de datos, las funciones en este módulo permiten la compresión y descompresión mediante la biblioteca zlib. La biblioteca zlib tiene su propia página de inicio en https://www.zlib.net. Existen incompatibilidades conocidas entre el módulo de Python y las versiones de la biblioteca zlib anteriores a la 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, deben usarse en un orden particular. Esta documentación no intenta cubrir todas las permutaciones; consulte 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: The result is always unsigned. To generate the same numeric value when using Python 2 or earlier, use `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: The result is always unsigned. To generate the same numeric value when using Python 2 or earlier, 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.

  This makes it possible to distinguish between a properly formed compressed stream, and an incomplete or truncated one.

  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*. `OFError` 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'`, or `'xb'`, dependiendo de si el archivo se leerá o escribirá. El valor predeterminado es el modo de *fileobj* si es discernible; de lo contrario, el valor predeterminado es `'rb'`. En futuras versiones de Python, no se utilizará el modo *fileobj*. Es mejor especificar siempre *mode* para escribir.

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 anexar 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 admite la interfaz *io.BufferedIOBase*, incluida la iteración y la declaración with. Solo el método truncate() no está implementado.

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()*.

name
The path to the gzip file on disk, as a *str* or *bytes*. Equivalent to the output of *os.fspath()* on the original input path, with no other normalization, resolution or expansion.

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*.

Obsoleto desde la versión 3.9: Abriendo *GzipFile* para escribir sin especificar el argumento *mode* está obsoleto.

gzip.**compress**(*data*, *compresslevel=9*, *, *mtime=None*)
Comprima 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.

13.3.1 (De)compresión de archivos

bz2.**open**(*filename*, *mode='rb'*, *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 linea).

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'*, ***, *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 anexar. 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'`.

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.

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*).

Distinto en la versión 3.9: El parámetro *buffering* ha sido retirado. Desde Python 3.0 era ignorado y estaba obsoleto. Pasa un objeto archivo abierto para controlar cómo el archivo es abierto.

El parámetro *compresslevel* se convirtió en un argumento sólo por palabra clave.

Distinto en la versión 3.10: Esta clase es insegura en hilos frente a múltiples lectores o escritores, al igual que sus clases equivalentes en *gzip* y *lzma* siempre lo han sido.

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`.

Attempting to decompress data after the end of stream is reached raises an *EOFError*. Any data found after the end of the stream is ignored and saved in the *unused_data* attribute.

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 modulo *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)
>>> 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()
```

The example above uses a very «nonrandom» stream of data (a stream of `b"z"` chunks). Random data tends to compress poorly, while ordered, repetitive data usually yields a high compression ratio.

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 proporcionada por este módulo es muy similar a la del módulo *bz2*. Tenga en cuenta que *LZMAFile* y *bz2.BZ2File* son seguros para subprocesos *not*, por lo que si necesita utilizar una única instancia *LZMAFile* de varios subprocesos, es necesario protegerla con un candado.

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*: `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 sobreescritura, `"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`.

Attempting to decompress data after the end of stream is reached raises an *EOFError*. Any data found after the end of the stream is ignored and saved in the *unused_data* attribute.

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()
# 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**
Class that implements a subset of the interface provided by *pathlib.Path*, including the full importlib.resources.abc.Traversable interface.

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 lanzará un *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* lanzará 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 lanzará 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*`)`
Access a member of the archive as a binary file-like object. *name* can be either the name of a file within the

archive or a *ZipInfo* object. The *mode* parameter, if included, must be `'r'` (the default) or `'w'`. *pwd* is the password used to decrypt encrypted ZIP files as a *bytes* object.

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())
```

With *mode* `'r'` the file-like object (`ZipExtFile`) is read-only and provides the following methods: *read()*, *readline()*, *readlines()*, *seek()*, *tell()*, *__iter__()*, *__next__()*. These objects can operate independently of the ZipFile.

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 lanzará 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: *ZipFile.open()* can now be used to write files into the archive with the `mode='w'` option.

Distinto en la versión 3.6: Llamar a *open()* en un ZipFile cerrado lanzará un *ValueError*. Anteriormente, se planteó a *RuntimeError*.

`ZipFile.`**`extract`**(*member*, *path=None*, *pwd=None*)

Extract a member from the archive to the current working directory; *member* must be its full name or a *ZipInfo* object. Its file information is extracted as accurately as possible. *path* specifies a different directory to extract to. *member* can be a filename or a *ZipInfo* object. *pwd* is the password used for encrypted files as a *bytes* object.

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 lanzará 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*)

Extract all members from the archive to the current working directory. *path* specifies a different directory to extract to. *members* is optional and must be a subset of the list returned by *namelist()*. *pwd* is the password used for encrypted files as a *bytes* object.

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 lanzará 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*`)`
Set *pwd* (a `bytes` object) as default password to extract encrypted files.

`ZipFile.`**`read`**`(`*name*, *pwd=None*`)`
Return the bytes of the file *name* in the archive. *name* is the name of the file in the archive, or a `ZipInfo` object. The archive must be open for read or append. *pwd* is the password used for encrypted files as a `bytes` object and, if specified, overrides the default password set with `setpassword()`. Calling `read()` on a ZipFile that uses a compression method other than `ZIP_STORED`, `ZIP_DEFLATED`, `ZIP_BZIP2` or `ZIP_LZMA` will raise a `NotImplementedError`. An error will also be raised if the corresponding compression module is not available.

Distinto en la versión 3.6: Llamando `read()` en un ZipFile cerrado lanzará 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: Llamar a `testzip()` en un ZipFile cerrado lanzará 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.

Nota: Un barra al frente en el nombre del archivo puede hacer que el archivo sea imposible de abrir en algunos programas zip en sistemas Windows.

Distinto en la versión 3.6: Llamando `write()` en un ZipFile creado con el modo `'r'` o un ZipFile cerrado lanzará 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 lanzará 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 son transitables utilizando el operador `/` o utilizando `joinpath`.

`Path.`**`name`**
El componente final de la ruta.

`Path.`**`open`**(*mode='r'*, ***, *pwd*, ****)
Invoca `ZipFile.open()` en la ruta actual. Permite la apertura para lectura o escritura, texto o binario a través de los modos admitidos: "r", "w", "rb", "wb". Los argumentos posicionales y de palabras clave se pasan a través de `io.TextIOWrapper` cuando se abren como texto y se ignoran en caso contrario. `pwd` es el parámetro `pwd` para `ZipFile.open()`.

Distinto en la versión 3.9: Se agregó soporte para modos de texto y binarios para abrir. El modo predeterminado ahora es texto.

Distinto en la versión 3.10.10: The `encoding` parameter can be supplied as a positional argument without causing a `TypeError`. As it could in 3.9. Code needing to be compatible with unpatched 3.10 and 3.11 versions must pass all `io.TextIOWrapper` arguments, `encoding` included, as keywords.

`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).

Distinto en la versión 3.10.10: The `encoding` parameter can be supplied as a positional argument without causing a *TypeError*. As it could in 3.9. Code needing to be compatible with unpatched 3.10 and 3.11 versions must pass all *io.TextIOWrapper* arguments, `encoding` included, as keywords.

Path.**read_bytes**()
Lee el archivo actual como bytes.

Path.**joinpath**(*other*)
Retorna un nuevo objeto de ruta con cada argumentos *other* unidos. Los siguientes son equivalentes:

```
>>> Path(...).joinpath('child').joinpath('grandchild')
>>> Path(...).joinpath('child', 'grandchild')
>>> Path(...) / 'child' / 'grandchild'
```

Distinto en la versión 3.10: Antes de 3.10, `joinpath` no estaba documentado y aceptaba exactamente un parámetro.

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 ademas 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 lanzará *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

```
string.pyc                  # Top level name
test/__init__.pyc           # Package directory
test/testall.pyc            # Module test.testall
test/bogus/__init__.pyc     # Subpackage directory
test/bogus/myfile.pyc       # Submodule test.bogus.myfile
```

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 *tarfile* exclusivamente sin compresión. Lanza una excepción `FileExistsError` si el archivo ya existe.
`'x:gz'`	Crear un archivo tar con compresión gzip. Lanza una excepción `FileExistsError` si ya existe.
`'x:bz2'`	Crear un archivo tar con compresión bzip2. Lanza una excepción `FileExistsError` si ya existe.
`'x:xz'`	Crear un archivo tar con compresión lzma. Lanza una excepción `FileExistsError` 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 los modos `'w:xz'` y `'x:xz'`, `tarfile.open()` acepta el argumento de palabra clave *preset* 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 *stream* de bloques tar para leer con compresión transparente.
`'r|'`	Abre un *stream* de bloques tar sin comprimir para lectura.
`'r|gz'`	Abre un *stream* comprimido con gzip para lectura.
`'r|bz2'`	Abre un *stream* bzip2 comprimido para lectura.
`'r|xz'`	Abre un *stream* lzma comprimido para lectura.
`'w|'`	Abre un *stream* sin comprimir para escritura.
`'w|gz'`	Abre un *stream* gzip comprimido para escritura.
`'w|bz2'`	Abre un *stream* bzip2 comprimido para escritura.
`'w|xz'`	Abre un *stream* 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. *name* puede ser un *str*, archivo o un objeto similar a un archivo.

Distinto en la versión 3.9: Soporte para archivos y objetos similares a archivos.

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 3.10.12, 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 3.10.12: 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 3.10.12: Se agregó el parámetro *filter*.

`TarFile.`**`extractfile`**(*member*)

Extrae un miembro del archivo como un objeto de archivo. *member* puede ser un nombre de archivo o un objeto `TarInfo`. Si *member* es un archivo normal o un enlace, se retorna un objeto `io.BufferedReader`. Para todos los demás miembros existentes, se retorna `None`. Si *member* no aparece en el archivo, se lanza `KeyError`.

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 3.10.12.

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 `gettarinfo()`.

`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 utilizado 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 claves-valor de los encabezados globales pax.

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 si 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 3.10.12: 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 invalido.

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 3.10.12: 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 3.10.12: 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 3.10.12: 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 3.10.12: 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 3.10.12: 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 3.10.12: 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 3.10.12.

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 3.10.12.

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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
        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 3.10.12.

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.
- The POSIX.1-2001 pax format (`PAX_FORMAT`). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. Modern tar implementations, including GNU tar, bsdtar/libarchive and star, fully support extended *pax* features; some old or unmaintained libraries may not, but should treat *pax* archives as if they were in the universally supported *ustar* format. It is the current default format for new archives.

  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, las encabezados globales son validos 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.

CAPÍTULO 14

Formatos de archivo

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 escritor responsable de convertir los datos del usuario en cadenas delimitadas en el objeto similar a un archivo dado. *csvfile* puede ser cualquier objeto con un método `write()`. Si *csvfile* es un objeto de archivo, debe abrirse con `newline=''`[1]. Se puede proporcionar un parámetro *dialect* opcional que se usa para definir un conjunto de parámetros específicos para un dialecto 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 opcionales de la palabra clave *fmtparams* se pueden proporcionar para anular los parámetros de formato individuales en el dialecto actual. Para obtener detalles completos sobre dialectos y parámetros de formato, consulte la sección *Dialectos y parámetros de formato*. Para facilitar al máximo la interfaz con módulos que implementan la API de base de datos, el valor *None* se escribe como una cadena vacía. Si bien esta no es una transformación reversible, facilita el volcado de valores de datos SQL NULL en archivos CSV sin preprocesar los datos retornados por una llamada `cursor.fetch*`. Todos los demás datos que no son cadenas se clasifican con *str()* antes de escribirse.

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*]])
Asocie *dialect* con *name*. *name* debe ser una cadena. El dialecto se puede especificar pasando una subclase de *Dialect*, o mediante argumentos de palabra clave *fmtparams*, o ambos, con argumentos de palabra clave que anulan los parámetros del dialecto. Para obtener detalles completos sobre dialectos y parámetros de formato, consulte 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*, ***kwds*)
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: Las filas retornadas son ahora de tipo `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*, ***kwds*)
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 cuyos atributos contienen información sobre cómo manejar las comillas dobles, los espacios en blanco, los delimitadores, etc. Debido a la falta de una especificación estricta de CSV, diferentes aplicaciones producen datos CSV sutilmente diferentes. Las instancias de *Dialect* definen cómo se comportan las instancias de *reader* y *writer*.

Todos los nombres disponibles de *Dialect* son retornados por *list_dialects()*, y pueden ser registrados con clases específicas de *reader* y *writer* a través de sus funciones que inician (__init__) así:

```
import csv

with open('students.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, dialect='unix')
                                 ^^^^^^^^^^^^^^
```

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 (se supone que está en formato CSV) y retorna *True* si la primera fila parece ser una serie de encabezados de columna. Al inspeccionar cada columna, se considerará uno de dos criterios clave para estimar si la muestra contiene un encabezado:

- las filas segunda n-th contienen valores numéricos
- las segundas filas n-th contienen cadenas en las que la longitud de al menos un valor difiere de la de la supuesta cabecera de esa columna.

Se muestran veinte filas después de la primera; si más de la mitad de las columnas + filas cumplen los criterios, se retorna *True*.

Nota: Este método es una heurística aproximada y puede producir tanto falsos positivos como negativos.

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 esta 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 el *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* esta 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**
When *True*, spaces immediately following the *delimiter* are ignored. The default is *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 lector como una lista (si el objeto retornó desde *reader()*) o un dict (si es una instancia de *DictReader*), analizado de acuerdo con el *Dialect* actual. Por lo general, debería llamar a esto 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:

DictReader.**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*)
Escriba el parámetro *row* en el objeto de archivo del escritor, formateado de acuerdo con el *Dialect* 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
[forge.example]
User = hg

[topsecret.server.example]
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['forge.example'] = {}
>>> config['forge.example']['User'] = 'hg'
>>> config['topsecret.server.example'] = {}
>>> topsecret = config['topsecret.server.example']
>>> 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()
['forge.example', 'topsecret.server.example']
>>> 'forge.example' in config
True
>>> 'python.org' in config
False
>>> config['forge.example']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.example']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['forge.example']:
...     print(key)
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['forge.example']['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 secciones[1]. Observe también que las claves de las secciones son insensibles a mayúsculas y minúsculas, pero se almacenan en minúscula[1].

Es posible leer varias configuraciones en un único *ConfigParser*, siendo la configuración añadida más recientemente la que tendrá la prioridad más alta. Las claves que entren en conflicto se toman de la configuración más reciente mientras que las claves preexistentes se mantienen.

```
>>> another_config = configparser.ConfigParser()
>>> another_config.read('example.ini')
['example.ini']
>>> another_config['topsecret.server.example']['Port']
'50022'
>>> another_config.read_string("[topsecret.server.example]\nPort=48484")
>>> another_config['topsecret.server.example']['Port']
'48484'
>>> another_config.read_dict({"topsecret.server.example": {"Port": 21212}})
>>> another_config['topsecret.server.example']['Port']
'21212'
>>> another_config['topsecret.server.example']['ForwardX11']
'no'
```

Este comportamiento es equivalente a una llamada a *ConfigParser.read()* pasando varios ficheros en el parámetro *filenames*.

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:

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['forge.example'].getboolean('ForwardX11')
True
>>> config.getboolean('forge.example', '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]

[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*.

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'
```

Please note that default values have precedence over fallback values. For instance, in our example the `'CompressionLevel'` key was specified only in the `'DEFAULT'` section. If we try to get it from the section `'topsecret.server.example'`, we will always get the default, even if we specify a fallback:

```
>>> 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('forge.example', '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 consta de secciones, cada una iniciada por una cabecera `[section]`, seguida por registros clave-valor separados por una cadena de caracteres específica (= ó `:` por defecto[1]). De forma predeterminada, los nombres de sección son sensibles a mayúsculas y minúsculas pero las claves no[1]. Los espacios al inicio y final de las claves y valores son eliminados. Los valores pueden ser omitidos si el *parser* está configurado para permitirlo[1], en cuyo caso el delimitador del dato clave-valor puede ser omitido también. Los valores pueden ocupar varias líneas, siempre y cuando tengan una indentación mayor que la primera línea del valor. Dependiendo del modo del *parser*, las líneas en blanco pueden tratarse como parte de un valor multilínea o ser ignoradas.

By default, a valid section name can be any string that does not contain “\n” or “]”. To change this, see *`ConfigParser.SECTCRE`*.

Los archivos de configuración pueden incluir comentarios, con caracteres específicos como prefijos (`#` y `;` por defecto[1]). Los comentarios pueden ocupar su propia línea, posiblemente indentada.[1]

Por ejemplo:

```
[Simple Values]
key=value
```

(continué en la próxima página)

(proviene de la página anterior)

```
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]
# use a %% to escape the % sign (% is the only character that needs to be␣
↪escaped):
```

(continué en la próxima página)

(proviene de la página anterior)

```
gain: 80%%
```

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]
# use a $$ to escape the $ sign ($ is the only character that needs to be␣
↪escaped):
cost: $$80
```

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]
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 minúsculas[1]. 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
...   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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
  ...
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
...
... 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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()
>>> 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 corres-

ponder `[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
# 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:

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*, *default_section=configparser.DEFAULTSECT*, *interpolation=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 las 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.

Nota: Los comentarios presentes en el fichero original no se mantienen cuando se vuelve a escribir la configuración. Qué es considerado un comentario depende de los valores asignados a *comment_prefix* e *inline_comment_prefix*.

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 sobreescribirla 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*, *default_section=configparser.DEFAULTSECT*[, *interpolation*])

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` — procesado 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.

Distinto en la versión 3.10: *netrc* prueba la codificación UTF-8 antes de usar la codificación específica de la configuración regional.

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 `plistlib` — Genera y analiza archivos `.plist` de Apple

Código fuente: Lib/plistlib.py

Este módulo proporciona una interfaz para leer y escribir los archivos de «lista de propiedades» utilizados por Apple, principalmente en macOS e iOS. Este módulo admite archivos plist binarios y 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, enteros, flotantes, booleanos, tuplas, listas, diccionarios (pero solo con claves de cadena), *bytes*, *bytearray* 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.

Distinto en la versión 3.9: API antigua eliminada.

Ver también:

Página del manual PList Documentación de Apple del formato de archivo.

Este módulo define las siguientes funciones:

plistlib.**load**(*fp*, *, *fmt=None*, *dict_type=dict*)
Lee un archivo plist. *fp* debe ser un objeto de archivo binario y legible. Retorna el objeto raíz desempaquetado (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

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*, *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 lanza *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.

Las siguientes clases están disponibles:

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`, que se puede utilizar para recuperar el valor int del UID. `data` debe estar en el rango `0 <= 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.

14.4.1 Ejemplos

Generar un plist:

```
import datetime
import plistlib

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.now()
)
print(plistlib.dumps(pl).decode())
```

Analizar un plist:

```
import plistlib

plist = b"""<plist version="1.0">
<dict>
    <key>foo</key>
    <string>bar</string>
</dict>
</plist>"""
pl = plistlib.loads(plist)
print(pl["foo"])
```

CAPÍTULO 15

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 seguro y resúmenes de mensajes. Están incluidos los algoritmos de hash seguro FIPS 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 eran 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 hash que siempre están presentes en este módulo son `sha1()`, `sha224()`, `sha256()`, `sha384()`, `sha512()`, *`blake2b()`* y *`blake2s()`*. `md5()` normalmente también está disponible, aunque puede faltar o estar bloqueado si está utilizando una rara compilación de Python «compatible con FIPS «. También pueden estar disponibles algoritmos adicionales dependiendo de la biblioteca OpenSSL que Python use en su plataforma. En la mayoría de las plataformas también están disponibles `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()`.

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()`*. Distinto en la versión 3.9: Todos los constructores hashlib toman un argumento de solo palabra clave *usedforsecurity* con el valor predeterminado `True`. Un valor falso permite el uso de algoritmos hash inseguros y bloqueados en entornos restringidos. `False` indica que el algoritmo hash no se utiliza en un contexto de seguridad, por ejemplo, como una función de compresión unidireccional no criptográfica.

Hashlib ahora usa SHA3 y SHAKE de OpenSSL 1.1.1 y posteriores.

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\xdd}Ae\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*], *, *usedforsecurity=True*)
: 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('sha256')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

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()`*.

The number of *iterations* should be chosen based on the hash algorithm and computing power. As of 2022, hundreds of thousands of iterations of SHA-256 are suggested. For rationale as to why and how to choose what is best for your application, read *Appendix A.2.2* of NIST-SP-800-132. The answers on the stackexchange pbkdf2 iterations question explain in detail.

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.

```
>>> from hashlib import pbkdf2_hmac
>>> our_app_iters = 500_000  # Application specific, read above.
>>> dk = pbkdf2_hmac('sha256', b'password', b'bad salt'*2, our_app_iters)
>>> dk.hex()
'15530bba69924174860db778f2c6f8104d3aaf9d26241840c8c4a641c8d000a9'
```

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.

Obsoleto desde la versión 3.10: La implementación lenta en Python de *pbkdf2_hmac* está obsoleta. En el futuro la función estará disponible sólo cuando Python sea compilado con OpenSSL.

`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.unrandom()`.

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.

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*, *usedforsecurity=True*)

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*, *usedforsecurity=True*)

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

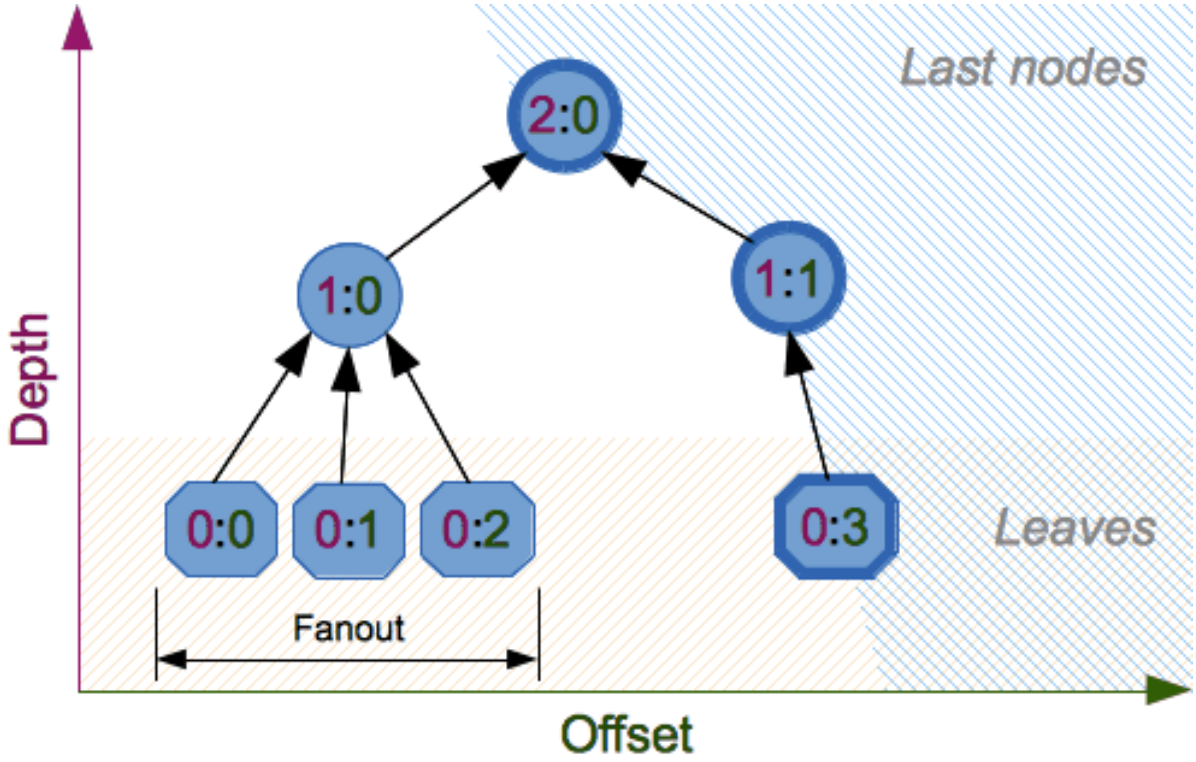
Nota: La especificación BLAKE2 define largos constantes para los parámetros de sal y personalización, sin embargo, por conveniencia, esta implementación acepta cadenas de bytes de cualquier tamaño hasta el largo especificado. Si el largo del parámetro es menor que el especificado, es acolchado con ceros, 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 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*: boolean indicating whether the processed node is the last one (`False` for sequential mode).

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 autentificación como un remplazo más rápido y simple para Código de autentificación 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 autentificación (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 autentificación 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/rfc8018.txt PKCS #5: Password-Based Cryptography Specification Version 2.1

https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf NIST Recommendation for Password-Based Key Derivation.

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: When comparing the output of *digest()* to an externally supplied digest during a verification routine, it is recommended to use the *compare_digest()* function instead of the == operator to reduce the vulnerability to timing attacks.

HMAC.**hexdigest**()
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: When comparing the output of *hexdigest()* to an externally supplied digest during a verification routine, it is recommended to use the *compare_digest()* function instead of the == operator to reduce the vulnerability to timing attacks.

HMAC.**copy**()
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.

Obsoleto desde la versión 3.9: Los atributos no documentados `HMAC.digest_cons`, `HMAC.inner` y `HMAC.outer` son detalles de implementación interna y se eliminarán en Python 3.10.

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.

Distinto en la versión 3.10: La función utiliza `CRYPTO_memcmp()` de OpenSSL internamente cuando está disponible.

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.

En particular, *secrets* se debe usar en lugar del generador de números pseudoaleatorios del módulo *random*, que está diseñado para el modelado y la simulación, no para la seguridad o la criptografía.

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*)
Return a randomly chosen element from a non-empty sequence.

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*)

Return True if strings or *bytes-like objects* *a* and *b* are equal, otherwise False, using a «constant-time compare» to reduce the risk of timing attacks. See *hmac.compare_digest()* for additional details.

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: Applications should not store passwords in a recoverable format, whether plain text or encrypted. They should be salted and hashed using a cryptographically strong one-way (irreversible) hash function.

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://example.com/reset=' + secrets.token_urlsafe()
```

CAPÍTULO 16

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, os.popen, os.fork, os.execv y os.spawn*p* no son compatibles.

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 se representan mediante el tipo de cadena de caracteres. En algunos sistemas, es necesario decodificar estas cadenas desde y hacia bytes antes de pasarlas al sistema operativo. Python usa la *codificación del sistema de archivos y manejador de errores* para realizar esta conversión (consulte *sys.getfilesystemencoding()*).

La *codificación del sistema de archivos y manejador de errores* se configuran al inicio de Python mediante la función PyConfig_Read(): consulte los miembros filesystem_encoding y filesystem_errors de PyConfig.

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 y manejador de errores* 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 API pueden generar *UnicodeError*.

Consulte también la *codificación de la configuración regional*.

16.1.2 Modo Python UTF-8

Nuevo en la versión 3.7: Consulte **PEP 540** para obtener más detalles.

El modo Python UTF-8 ignora la *codificación de la configuración regional* y fuerza el uso de la codificación UTF-8:

- Utilice UTF-8 como *codificación del sistema de archivos y manejador de errores*.
- *sys.getfilesystemencoding()* retorna 'UTF-8'.
- *locale.getpreferredencoding()* retorna 'UTF-8' (el argumento *do_setlocale* no tiene ningún efecto).
- *sys.stdin*, *sys.stdout* y *sys.stderr* usan UTF-8 como su codificación de texto, con surrogateescape *error handler* habilitado para *sys.stdin* y *sys.stdout* (*sys.stderr* continúa usando backslashreplace como lo hace en el modo predeterminado de reconocimiento de configuración regional)
- On Unix, *os.device_encoding()* returns 'UTF-8' rather than the device encoding.

Tenga en cuenta que PYTHONIOENCODING puede anular la configuración de transmisión estándar en el modo UTF-8 (al igual que en el modo predeterminado con reconocimiento de configuración regional).

Como consecuencia de los cambios en esas API de nivel inferior, otras API de nivel superior también exhiben diferentes comportamientos predeterminados:

- Los argumentos de la línea de comandos, las variables de entorno y los nombres de archivo se decodifican en texto utilizando la codificación UTF-8.
- *os.fsdecode()* y *os.fsencode()* utilizan la codificación UTF-8.

- *open()*, *io.open()* y *codecs.open()* utilizan la codificación UTF-8 de forma predeterminada. Sin embargo, todavía usan el controlador de errores estricto de forma predeterminada, por lo que es probable que intentar abrir un archivo binario en modo texto genere una excepción en lugar de producir datos sin sentido.

El *Modo Python UTF-8* se habilita si la configuración regional LC_CTYPE es `C` o `POSIX` al iniciar Python (consulte la función `PyConfig_Read()`).

Se puede habilitar o deshabilitar usando la opción de línea de comando `-X utf8` y la variable de entorno `PYTHONUTF8`.

Si la variable de entorno `PYTHONUTF8` no está configurada en absoluto, entonces el intérprete utiliza de forma predeterminada la configuración regional actual, *a no ser que* la configuración regional actual se identifica como una configuración regional heredada basada en ASCII (como se describe para `PYTHONCOERCECLOCALE`) y la coerción de la configuración regional está deshabilitada o falla . En tales configuraciones regionales heredadas, el intérprete habilitará de forma predeterminada el modo UTF-8 a menos que se le indique explícitamente que no lo haga.

El modo Python UTF-8 solo se puede habilitar al inicio de Python. Su valor se puede leer en *sys.flags.utf8_mode*.

Consulte también modo UTF-8 en Windows y *codificación del sistema de archivos y manejador de errores*.

16.1.3 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`**
A *mapping* object where keys and values are strings that represent the process environment. For example, `environ['HOME']` is the pathname of your home directory (on some platforms), and is equivalent to `getenv("HOME")` in C.

This mapping is captured the first time the *os* module is imported, typically during Python startup as part of processing `site.py`. Changes to the environment made after this time are not reflected in *os.environ*, except for changes made by modifying *os.environ* directly.

Este mapeo puede ser usado para modificar el entorno y también para consultarlo. *putenv()* será llamado automáticamente cuando el 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.

On Windows, the keys are converted to uppercase. This also applies when getting, setting, or deleting an item. For example, `environ['monty'] = 'python'` maps the key `'MONTY'` to the value `'python'`.

Nota: Calling *putenv()* directly does not change *os.environ*, so it's better to modify *os.environ*.

Nota: En algunas plataformas, incluidas FreeBSD y macOS, configurar `environ` puede provocar pérdidas de memoria. Consulte la documentación del sistema para `putenv()`.

You can delete items in this mapping to unset environment variables. *unsetenv()* will be called automatically when an item is deleted from *os.environ*, and when one of the `pop()` or `clear()` methods is called.

Distinto en la versión 3.9: Actualizado para soportar los operadores merge (`|`) y update (`|=`) de **PEP 584**'s.

os.**environb**
Bytes version of *environ*: a *mapping* object where both keys and values are *bytes* objects representing the process environment. *environ* and *environb* are synchronized (modifying *environb* updates *environ*, and vice versa).

environb está disponible sólo si *supports_bytes_environ* está establecido en `True`.

Nuevo en la versión 3.2.

Distinto en la versión 3.9: Actualizado para soportar los operadores merge (|) y update (|=) de **PEP 584**'s.

os.**chdir**(*path*)
os.**fchdir**(*fd*)
os.**getcwd**()
Estas funciones están detalladas en *Archivos y directorios*.

os.**fsencode**(*filename*)
Codifique *path-like filename* en *filesystem encoding and error handler*; retorna *bytes* sin cambios.

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 el *path-like filename* del *filesystem encoding and error handler*; retorna *str* sin cambios.

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* or *bytes*, preferentemente *str*.

os.**getenv**(*key*, *default=None*)
Return the value of the environment variable *key* as a string if it exists, or *default* if it doesn't. *key* is a string. Note that since *getenv()* uses *os.environ*, the mapping of *getenv()* is similarly also captured on import, and the function may not reflect future environment changes.

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.**getenvb**(*key*, *default=None*)
Return the value of the environment variable *key* as bytes if it exists, or *default* if it doesn't. *key* must be bytes. Note that since *getenvb()* uses *os.environb*, the mapping of *getenvb()* is similarly also captured on import, and the function may not reflect future environment changes.

getenvb() 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 el 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 macOS, el comportamiento de *getgroups()* difiere algo del de otras plataformas Unix. Si el intérprete de Python se creó con un destino de implementación de `10.5` o anterior, *getgroups()* retorna la lista de identificadores de grupo efectivos asociados con el proceso de usuario actual; esta lista está limitada a un número de entradas definido por el sistema, normalmente 16, y puede modificarse mediante llamadas a *setgroups()* si tiene los privilegios adecuados. Si se construyó con un objetivo de implementación mayor que `10.5`, *getgroups()* retorna la lista de acceso de grupo actual para el usuario asociado con el ID de usuario efectivo del proceso; la lista de acceso de grupo puede cambiar durante la vida útil del proceso, no se ve afectada por las llamadas a *setgroups()* y su longitud no está limitada a 16. El valor de destino de implementación, `MACOSX_DEPLOYMENT_TARGET`, se puede obtener 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()*.

Assignments to items in *os.environ* are automatically translated into corresponding calls to *putenv()*; however, calls to *putenv()* don't update *os.environ*, so it is actually preferable to assign to items of *os.environ*. This also applies to *getenv()* and *getenvb()*, which respectively use *os.environ* and *os.environb* in their implementations.

Nota: En algunas plataformas, incluidas FreeBSD y macOS, configurar `environ` puede provocar pérdidas de memoria. Consulte la documentación del sistema para `putenv()`.

Lanza un *evento de auditoría* `os.putenv` con argumentos `key`, `value`.

Distinto en la versión 3.9: Esta función actualmente siempre esta disponible.

`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 macOS, la longitud de *groups* no puede exceder el número máximo definido por el sistema de ID de grupo efectivos, por lo general 16. Consulte la documentación de *getgroups()* para ver los casos en los que puede no retornar la misma lista de grupos establecida 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()`.

Deletion of items in `os.environ` is automatically translated into a corresponding call to `unsetenv()`; however, calls to `unsetenv()` don't update `os.environ`, so it is actually preferable to delete items of `os.environ`.

Lanza un *evento de auditoría* `os.unsetenv` con argumento `key`.

Distinto en la versión 3.9: Estas funciones se encuentra ahora siempre disponible y también esta disponible en Windows

16.1.4 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.

16.1.5 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 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 *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*.

En Unix, si el *Modo Python UTF-8* está habilitado, retorna `'UTF-8'` en lugar de la codificación del dispositivo.

Distinto en la versión 3.10: En Unix, la función ahora implementa el modo Python UTF-8.

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 metadatos.

Disponibilidad: Unix.

Nota: Esta función no está disponible en MacOS.

---

os.**fpathconf**(*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 *rutas 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_EVTONLY**
os.**O_FSYNC**
os.**O_SYMLINK**
os.**O_NOFOLLOW_ANY**
Las constantes anteriores solo están disponibles en macOS.

Distinto en la versión 3.10: Agregue las constantes *O_EVTONLY*, *O_FSYNC*, *O_SYMLINK* y *O_NOFOLLOW_ANY*.

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.**preadv**(*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, AIX 7.1 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*
- *RWF_APPEND*

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, AIX 7.1 y posterior. El uso de flags requiere Linux 4.7 o posterior.

Nuevo en la versión 3.7.

os.**RWF_DSYNC**
Proporcione un equivalente por escritura del indicador *O_DSYNC* *os.open()*. Este efecto de bandera se aplica solo 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**
Proporcione un equivalente por escritura del indicador *O_SYNC* *os.open()*. Este efecto de bandera se aplica solo 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_APPEND**
Proporcione un equivalente por escritura del indicador *O_APPEND* *os.open()*. Esta bandera es significativa solo para *os.pwritev()* y su efecto se aplica solo al rango de datos escrito por la llamada al sistema. El argumento *offset* no afecta la operación de escritura; los datos siempre se añaden al final del archivo. Sin embargo, si el argumento *offset* es `-1`, se actualiza el *offset* actual del archivo.

Disponibilidad: Linux 4.16 y más nuevos.

Nuevo en la versión 3.10.

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_fd*, *in_fd*, *offset*, *count*)
os.**sendfile**(*out_fd*, *in_fd*, *offset*, *count*, *headers=()*, *trailers=()*, *flags=0*)
Copia *count* bytes del descriptor de archivo *in_fd* al descriptor de archivo *out_fd* 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 entrega como `None`, los bytes se leen desde la posición actual de *in_fd* y la posición de *in_fd* se actualiza.

El segundo caso se puede usar en macOS y FreeBSD donde *headers* y *trailers* son secuencias arbitrarias de búferes que se escriben antes y después de que se escriban los datos de *in_fd*. Retorna lo mismo que el primer caso.

En macOS y FreeBSD, un valor de `0` para *count* especifica enviar hasta que se alcanza el final de *in_fd*.

Todas las plataformas admiten sockets como descriptor de archivo *out_fd*, y algunas plataformas también permiten otros tipos (por ejemplo, archivo regular, 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.

Distinto en la versión 3.9: Parámetros *out* e *in* han sido renombrados a *out_fd* e *in_fd*.

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`, borre 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.**splice**(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*)
Transfiera *count* bytes desde el descriptor de archivo *src*, comenzando desde el desplazamiento *offset_src*, al descriptor de archivo *dst*, comenzando desde el desplazamiento *offset_dst*. Al menos uno de los descriptores de archivo debe hacer referencia a una tubería. Si *offset_src* es None, entonces *src* se lee desde la posición actual; respectivamente para *offset_dst*. El desplazamiento asociado al descriptor de archivo que hace referencia a una tubería debe ser None. Los archivos señalados por *src* y *dst* deben residir en el mismo sistema de archivos; de lo contrario, se genera un *OSError* con *errno* configurado 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.

Una vez completado con éxito, retorna el número de bytes empalmados hacia o desde la tubería. Un valor de retorno de 0 significa el final de la entrada. Si *src* se refiere a una tubería, esto significa que no hay datos para transferir y no tendría sentido bloquear porque no hay escritores conectados al extremo de escritura de la tubería.

Disponibilidad: Kernel de Linux >= 2.6.17 y glibc >= 2.5.

Nuevo en la versión 3.10.

os.**SPLICE_F_MOVE**
os.**SPLICE_F_NONBLOCK**
os.**SPLICE_F_MORE**
Nuevo en la versión 3.10.

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.

16.1.6 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 lanzará 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 lanzará 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 `l...` 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 lanzará 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 lanzará 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='.'*)
Retorna una lista que contiene los nombres de las entradas en el directorio dado por *path*. La lista está en un orden arbitrario y no incluye las entradas especiales `'.'` Y `'..'` incluso si están presentes en el directorio. Si un archivo es removido de o añadido al directorio mientras se llama a esta función, no se especifica si el nombre para el archivo será incluido.

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 *fsencode()*.

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 descriptores 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*.

If the directory already exists, *FileExistsError* is raised. If a parent directory in the path does not exist, *FileNotFoundError* is raised.

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.

Esta función también puede admitir *rutas relativas a descriptores 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*.

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.

The *mode* parameter is passed to *mkdir()* for creating the leaf directory; see *the mkdir() description* for how it is interpreted. To set the file permission bits of any newly created parent directories you can set the umask before invoking *makedirs()*. The file permission bits of existing parent directories are not changed.

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: The *mode* argument no longer affects the file permission bits of newly created intermediate-level directories.

`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 descriptores 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 *OSError* is raised. Use *rmdir()* to remove directories. If the file does not exist, a *FileNotFoundError* is raised.

Esta función puede admitir *rutas relativas a descriptores 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, `removeirs()` 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:

On Windows, if *dst* exists a `FileExistsError` is always raised. The operation may fail if *src* and *dst* are on different filesystems. Use `shutil.move()` to support moves to a different filesystem.

On Unix, if *src* is a file and *dst* is a directory or vice-versa, an `IsADirectoryError` or a `NotADirectoryError` will be raised respectively. If both are directories and *dst* is empty, *dst* will be silently replaced. If *dst* is a non-empty directory, an `OSError` is raised. If both are files, *dst* will be replaced silently if the user has permission. The operation may fail on some Unix flavors if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

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*)

Rename the file or directory *src* to *dst*. If *dst* is a non-empty directory, `OSError` will be raised. If *dst* exists and is a file, it will be replaced silently if the user has permission. The operation may fail if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

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='.'*)

Retorna un iterador de objetos `os.DirEntry` correspondientes a las entradas en el directorio dado por *path*. Las entradas se entregan en orden arbitrario, y las entradas especiales `'.'` Y `'..'` no están incluidas. Si un archivo es removido de o añadido al directorio después de crear el iterador, no se especifica si una entrada para el archivo será incluido.

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: On Unix-based systems, *scandir()* uses the system's opendir() and readdir() functions. On Windows, it uses the Win32 FindFirstFileW and FindNextFileW functions.

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 lanzará 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 inodo.

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 macOS, 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 a 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 lanzarán 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 *fective_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*.

Note that the exact times you set here may not be returned by a subsequent *stat()* call, depending on the resolution with which your operating system records access and modification times; see *stat()*. The best way to preserve exact times is to use the *st_atime_ns* and *st_mtime_ns* fields from the *os.stat()* result object with the *ns* parameter to *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* (including symlinks to directories, and 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")
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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))
```

Lanza un *evento de auditoría* `os.spawn` con argumentos `top`, `topdown`, `onerror`, `followlinks`.

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 descriptores 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 descriptores 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:
```

(continué en la próxima página)

(proviene de la página anterior)

```
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

Lanza un *evento de auditoría* `os.chown` con argumentos `top`, `topdown`, `onerror`, `follow_symlinks`, `dir_fd`.

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.

`os.`**`eventfd`**(*initval*[, *flags=os.EFD_CLOEXEC*])

Crea y retorna un descriptor de archivo de eventos. Los descriptores de archivo admiten `read()` y `write()` sin procesar con un tamaño de búfer de 8, `select()`, `poll()` y similares. Consulte la página de manual `eventfd(2)` para obtener más información. De forma predeterminada, el nuevo descriptor de archivo *no es heredable*.

initval es el valor inicial del contador de eventos. El valor inicial debe ser un entero sin signo de 32 bits. Tenga en cuenta que el valor inicial está limitado a un entero sin signo de 32 bits, aunque el contador de eventos es un entero de 64 bits sin signo con un valor máximo de 2^{64}-2.

flags se puede construir a partir de `EFD_CLOEXEC`, `EFD_NONBLOCK` y `EFD_SEMAPHORE`.

Si se especifica *EFD_SEMAPHORE* y el contador de eventos no es cero, *eventfd_read()* retorna 1 y reduce el contador en uno.

Si no se especifica *EFD_SEMAPHORE* y el contador de eventos no es cero, *eventfd_read()* retorna el valor actual del contador de eventos y restablece el contador a cero.

Si el contador de eventos es cero y no se especifica *EFD_NONBLOCK*, *eventfd_read()* se bloquea.

eventfd_write() incrementa el contador de eventos. Escribe bloques si la operación de escritura incrementaría el contador a un valor mayor que ^{64}-2.

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

# semaphore with start value '1'
fd = os.eventfd(1, os.EFD_SEMAPHORE | os.EFC_CLOEXEC)
try:
    # acquire semaphore
    v = os.eventfd_read(fd)
    try:
        do_work()
    finally:
        # release semaphore
        os.eventfd_write(fd, v)
finally:
    os.close(fd)
```

Disponibilidad: Linux 2.6.27 o posterior con glibc 2.8 o posterior.

Nuevo en la versión 3.10.

`os.`**`eventfd_read`**(*fd*)
Lee el valor de un descriptor de archivo *eventfd()* y retorna un int sin signo de 64 bits. La función no verifica que *fd* sea un *eventfd()*.

Disponibilidad: Ver *eventfd()*

Nuevo en la versión 3.10.

`os.`**`eventfd_write`**(*fd*, *value*)
Agregue valor a un descriptor de archivo *eventfd()*. *value* debe ser un int sin signo de 64 bits. La función no verifica que *fd* sea un *eventfd()*.

Disponibilidad: Ver *eventfd()*

Nuevo en la versión 3.10.

`os.`**`EFD_CLOEXEC`**
Establezca el indicador close-on-exec para el nuevo descriptor de archivo *eventfd()*.

Disponibilidad: Ver *eventfd()*

Nuevo en la versión 3.10.

`os.`**`EFD_NONBLOCK`**
Establezca el indicador de estado *O_NONBLOCK* para el nuevo descriptor de archivo *eventfd()*.

Disponibilidad: Ver *eventfd()*

Nuevo en la versión 3.10.

`os.`**`EFD_SEMAPHORE`**
Proporcione semántica similar a un semáforo para las lecturas de un descriptor de archivo *eventfd()*. Al leer, el contador interno se reduce en uno.

Disponibilidad: Linux 2.6.30 o posterior con glibc 2.8 o posterior.

Nuevo en la versión 3.10.

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 *atrribute* 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 extendido del sistema de archivos *attribute* de *path*. *attribute* debe ser bytes o str (directa o indirectamente a través de la interfaz *PathLike*). Si es una cadena, está codificada con *filesystem encoding and error handler*.

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*)
Establece el atributo del sistema de archivos extendido *attribute* en *path* en *value*. *attribute* debe ser un bytes o una cadena sin NUL incrustados (directa o indirectamente a través de la interfaz *PathLike*). Si es una cadena, está codificado con *filesystem encoding and error handler*. *flags* puede ser *XATTR_REPLACE* o *XATTR_CREATE*. Si se proporciona *XATTR_REPLACE* y el atributo no existe, se lanzará `ENODATA`. Si se proporciona *XATTR_CREATE* y el atributo ya existe, no se creará el atributo y se lanza `EEXISTS`.

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.

16.1.7 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.

This search path is used when resolving dependencies for imported extension modules (the module itself is resolved through *sys.path*), and also by *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.**execle**(*path*, *arg0*, *arg1*, ..., *env*)
os.**execlp**(*file*, *arg0*, *arg1*, ...)
os.**execlpe**(*file*, *arg0*, *arg1*, ..., *env*)
os.**execv**(*path*, *args*)
os.**execve**(*path*, *args*, *env*)
os.**execvp**(*file*, *args*)
os.**execvpe**(*file*, *args*, *env*)
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 lanzará 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: The standard way to exit is *sys.exit(n)*. `_exit()` should normally only be used in the child process after a *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.**pidfd_open**(*pid*, *flags=0*)
Retorna un descriptor de archivo referente al *pid* del proceso. Este descriptor puede ser usado para realizar gestión de procesos sin necesidad de señales y carreras. El argumento *flags* es proporcionado para extensiones futuras; ningún valor se encuentra definido en las banderas actualmente.

Ver la página manual de *pidfd_open(2)* para mas detalles.

Disponibilidad: Linux 5.3+

Nuevo en la versión 3.9.

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.

En Unix, *waitstatus_to_exitcode()* puede ser usado para convertir el resultado del método `close` (estado de salida) en un código de salida si es que no es `None`. En Windows, el resultado del método `close` es directamente el código de salida (o `None`).

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()*.

El parámetro *path* es la ruta al archivo ejecutable. La *path* debe contener un directorio. Utilice *posix_spawnp()* para pasar un archivo ejecutable sin directorio.

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.

If the *setsid* argument is True, it will create a new session ID for posix_spawn. *setsid* requires POSIX_SPAWN_SETSID or POSIX_SPAWN_SETSID_NP flag. Otherwise, NotImplementedError is raised.

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_spawnp**(*path*, *argv*, *env*, *, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

Envuelve la API de la biblioteca posix_spawnp() 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 subprocesos 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*][, *arguments*][, *cwd*][, *show_cmd*])
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).

Al iniciar una aplicación, especifique *arguments* para que se pase como una sola cadena. Es posible que este argumento no tenga ningún efecto cuando se utiliza esta función para iniciar un documento.

El directorio de trabajo predeterminado se hereda, pero el argumento *cwd* puede anularlo. Este debería ser un camino absoluto. Un *path* relativo se resolverá contra este argumento.

Utilice *show_cmd* para anular el estilo de ventana predeterminado. Si esto tiene algún efecto dependerá de la aplicación que se esté iniciando. Los valores son números enteros admitidos por la función Win32 ShellExecute().

startfile() retorna tan pronto como se inicia la aplicación asociada. No hay opción para esperar a que se cierre la aplicación y no hay forma de recuperar el estado de salida de la aplicación. El parámetro *path* es relativo al directorio actual o *cwd*. Si desea utilizar una ruta absoluta, asegúrese de que el primer carácter no sea una barra ('/') Utilice *pathlib* o la función *os.path.normpath()* para asegurarse de que las rutas estén codificadas 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 lanzará *NotImplementedError*.

Lanza un *evento de auditoría* os.startfile con argumentos path, operation.

Lanza un *evento de auditoria* os.startfile/2 con argumentos path, operation, arguments, cwd, show_cmd.

Disponibilidad: Windows.

Distinto en la versión 3.10: Se agregaron los argumentos *arguments*, *cwd* y *show_cmd*, y el evento de auditoría os.startfile/2.

os.**system**(*command*)
Ejecute el comando (una cadena) en una subcapa. Esto se implementa llamando a la función C estándar 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. El estándar C no especifica el significado del valor de retorno de la función C, por lo que el valor de retorno de la función de Python depende del sistema.

En Unix, el valor de retorno es el estado de salida del proceso codificado en el formato especificado para *wait()*.

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.

En Unix, *`waitstatus_to_exitcode()`* puede ser usado para convertir el resultado (estado de salida) en un código de salida. En Windows, el resultado es directamente el código de salida.

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` - user time
- `system` - system time
- `children_user` - user time of all child processes
- `children_system` - system time of all child processes
- `elapsed` - elapsed real time since a fixed point in the past

For backwards compatibility, this object also behaves like a five-tuple containing `user`, `system`, `children_user`, `children_system`, and `elapsed` in that order.

See the Unix manual page *`times(2)`* and times(3) manual page on Unix or the GetProcessTimes MSDN on Windows. On Windows, only `user` and `system` are known; the other attributes are zero.

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.

`waitstatus_to_exitcode()` puede ser usado para convertir el estado de salida en el código de salida.

Disponibilidad: Unix.

Ver también:

`waitpid()` puede ser usado para esperar a la terminación de algún proceso hijo en específico y tiene más opciones.

`os.`**`waitid`**`(`*idtype*, *id*, *options*`)`
Espere la finalización de uno o más procesos secundarios. *idtype* puede ser *`P_PID`*, *`P_PGID`*, *`P_ALL`* o *`P_PIDFD`* en Linux. *id* especifica el pid para esperar. *options* se construye a partir de ORing de uno o más de *`WEXITED`*, *`WSTOPPED`* o *`WCONTINUED`* y adicionalmente se puede ORed 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` si *`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.**P_PIDFD**
Esto es un *idtype* especifico de Linux que indica que ese *id* es un descriptor de archivo que hace referencia a un proceso.

Disponibilidad: Linux 5.4+

Nuevo en la versión 3.9.

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_KILLED**
os.**CLD_DUMPED**
os.**CLD_TRAPPED**
os.**CLD_STOPPED**
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.

Distinto en la versión 3.9: Agregado los valores `CLD_KILLED` y `CLD_STOPPED`.

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.

`waitstatus_to_exitcode()` puede ser usado para convertir el estado de salida en el código de salida.

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()*.

waitstatus_to_exitcode() puede ser usado para convertir el estado de salida al código de salida.

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()*.

waitstatus_to_exitcode() puede ser usado para convertir el estado de salida al código de salida.

Disponibilidad: Unix.

os.**waitstatus_to_exitcode**(*status*)
Convertir un estado de espera a un código de salida.

En Unix:

- Si el proceso termino con normalidad (if `WIFEXITED(status)` is true, se retornan el estado de salida del proceso (return `WEXITSTATUS(status)`): resultado mayor o igual a 0.
- Si el proceso fue terminado por una señal (if `WIFSIGNALED(status)` is true), retorna `-signum` donde *signum* es el numero de la señal que cause que el proceso termine (return `-WTERMSIG(status)`): resultado menor que 0.
- En el caso contrario, se lanza un *ValueError*.

En Windows, retorna *status* desplazado a la derecha en 8 bits.

En Unix, si el proceso esta siendo rastreado o si *waitpid()* fue llamado con la opción *WUNTRACED*, el que llama debe revisar primero si `WIFSTOPPED(status)` es verdadero. La función no debe de ser llamada si `WIFSTOPPED(status)` es verdadero.

Ver también:

Funciones *WIFEXITED()*, *WEXITSTATUS()*, *WIFSIGNALED()*, *WTERMSIG()*, *WIFSTOPPED()*, *WSTOPSIG()*.

Nuevo en la versión 3.9.

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.

16.1.8 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 los 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.

16.1.9 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_DEEPBIND**
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.

16.1.10 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*.

See also the Linux getrandom() manual page.

Disponibilidad: Linux 3.17 y más reciente.

Nuevo en la versión 3.6.

os.**urandom**(*size*)
Return a bytestring of *size* random bytes suitable for cryptographic use.

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 descodificación 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, descodificación, 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 Codificación de texto

La codificación predeterminada de `TextIOWrapper` y `open()` es específica de la configuración regional (`locale.getpreferredencoding(False)`).

Sin embargo, muchos desarrolladores olvidan especificar la codificación al abrir archivos de texto codificados en UTF-8 (por ejemplo, JSON, TOML, Markdown, etc.) ya que la mayoría de las plataformas Unix usan la configuración regional UTF-8 de forma predeterminada. Esto causa errores porque la codificación local no es UTF-8 para la mayoría de los usuarios de Windows. Por ejemplo:

```
# May not work on Windows when non-ASCII characters in the file.
with open("README.md") as f:
    long_description = f.read()
```

Además, aunque todavía no hay un plan concreto, Python puede cambiar la codificación predeterminada del archivo de texto a UTF-8 en el futuro.

En consecuencia, se recomienda encarecidamente que especifique la codificación explícitamente al abrir archivos de texto. Si desea utilizar UTF-8, pase `encoding="utf-8"`. Para usar la codificación de la configuración regional actual, `encoding="locale"` es compatible con Python 3.10.

When you need to run existing code on Windows that attempts to open UTF-8 files using the default locale encoding, you can enable the UTF-8 mode. See UTF-8 mode on Windows.

EncodingWarning opcional

Nuevo en la versión 3.10: Consulte **PEP 597** para obtener más detalles.

Para encontrar dónde se usa la codificación de configuración regional predeterminada, puede habilitar la opción de línea de comando `-X warn_default_encoding` o establecer la variable de entorno `PYTHONWARNDEFAULTENCODING`, que emitirá un `EncodingWarning` cuando se use la codificación predeterminada.

Si está proporcionando una API que usa `open()` o `TextIOWrapper` y pasa `encoding=None` como parámetro, puede usar `text_encoding()` para que las personas que llaman a la API emitan un `EncodingWarning` si no pasan un `encoding`. Sin embargo, considere usar UTF-8 de forma predeterminada (es decir, `encoding="utf-8"`) para las nuevas API.

16.2.3 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*, *closefd=True*, *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.

io.**text_encoding**(*encoding*, *stacklevel=2*, */*)
Esta es una función auxiliar para personas que llaman que usan *open()* o *TextIOWrapper* y tienen un parámetro `encoding=None`.

Esta función retorna *encoding* si no es `None` y `"locale"` si *encoding* es `None`.

Esta función emite un *EncodingWarning* si *sys.flags.warn_default_encoding* es verdadero y *encoding* es None. *stacklevel* especifica dónde se emite la advertencia. Por ejemplo:

```
def read_text(path, encoding=None):
    encoding = io.text_encoding(encoding)  # stacklevel=2
    with open(path, encoding) as f:
        return f.read()
```

En este ejemplo, se emite un *EncodingWarning* para el llamador de `read_text()`.

Consulte *Codificación de texto* para obtener más información.

Nuevo en la versión 3.10.

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*.

16.2.4 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.

The *BufferedIOBase* ABC extends *IOBase*. It deals with buffering on a raw binary stream (*RawIOBase*). Its subclasses, *BufferedWriter*, *BufferedReader*, and *BufferedRWPair* buffer raw binary streams that are

writable, readable, and both readable and writable, respectively. *BufferedRandom* provides a buffered interface to seekable streams. Another *BufferedIOBase* subclass, *BytesIO*, is a stream of in-memory bytes.

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 *TextIOBase*, 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 (*Stub*)	Métodos Mixin y propiedades
IOBase		`fileno`, `seek`, and `truncate`	`close`, `closed`, `__enter__`, `__exit__`, `flush`, `isatty`, `__iter__`, `__next__`, `readable`, `readline`, `readlines`, `seekable`, `tell`, `writable`, and `writelines`
RawIOBase	*IOBase*	`readinto` and `write`	Métodos *IOBase* heredados, `read`, and `readall`
BufferedIOBase	*IOBase*	`detach`, `read`, `read1`, and `write`	Métodos *IOBase* heredados, `readinto`, and `readinto1`
TextIOBase	*IOBase*	`detach`, `read`, `readline`, and `write`	Métodos *IOBase* heredados, `encoding`, `errors`, and `newlines`

Clases base E/S

class io.**IOBase**

The abstract base class for all I/O classes.

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 subclasificaciones) 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. Cuándo 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*.

Los valores *hint* de `0` o menos, así como `None`, se tratan como sin pista.

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**
Base class for raw binary streams. It inherits *IOBase*.

Los flujos binarios sin procesar generalmente brindan acceso de bajo nivel a un dispositivo OS subyacente o API, y no intentan encapsularlos en primitivas de alto nivel (esta funcionalidad se realiza en un nivel superior en flujos binarios almacenados en búfer y flujos de texto, que se describen más adelante en esta página).

RawIOBase proporciona estos métodos además de los de *IOBase*:

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.**BufferedIOBase**
Base class for binary streams that support some kind of buffering. It inherits *IOBase*.

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 or 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 *BufferedIOBase* no debería heredar una implementación de *RawIOBase*, es más, debería envolver como uno, así como hacen las clases *BufferedWriter* y *BufferedReader*.

BufferedIOBase proporciona o anula estos atributos y métodos de datos además de los de *IOBase*:

raw
El *stream* sin formato subyacente (una instancia *RawIOBase*) que *BufferedIOBase* maneja. Esto no es parte de la API *BufferedIOBase* 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=-1*, /)
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*)

Una secuencia binaria sin procesar que representa un archivo de nivel de sistema operativo que contiene datos en bytes. Hereda `RawIOBase`.

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 anexando. Si no existe el archivo se creará cuando se abra para grabar o anexar; 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.

`FileIO` proporciona estos atributos de datos además de los de `RawIOBase` y `IOBase`:

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=b''*)

Una secuencia binaria que utiliza un búfer de bytes en memoria. Hereda `BufferedIOBase`. El búfer se descarta 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=-1*, */*)
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*)
Una secuencia binaria almacenada en búfer que proporciona acceso de nivel superior a una secuencia binaria sin procesar *RawIOBase* legible y no buscable. Hereda *BufferedIOBase*.

Al leer datos de este objeto, es posible que se solicite una mayor cantidad de datos del flujo sin procesar subyacente y se mantenga en un búfer interno. Los datos almacenados en búfer se pueden retornar 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=0*, */*)
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=-1*, */*)
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=-1*, */*)
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.**BufferedWriter**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
Un flujo binario almacenado en búfer que proporciona acceso de nivel superior a un flujo binario sin formato *RawIOBase* que se puede escribir y no se puede buscar. Hereda *BufferedIOBase*.

Al escribir en este objeto, los datos normalmente se colocan en un búfer interno. El búfer se escribirá en el objeto *RawIOBase* subyacente en varias condiciones, que incluyen:

- cuando el búfer se vuelve demasiado pequeño para todos los datos pendientes;
- cuando se llama *flush()*;

- cuando se pide un método `seek()` (para objetos *BufferedRandom*);
- cuando el objeto *BufferedWriter* is cerrado o anulado.

El constructor crea un *BufferedWriter* para el *stream* grabable *raw*. Si no es dado el *buffer_size*, recurre el valor *DEFAULT_BUFFER_SIZE*.

BufferedWriter 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.**BufferedRandom**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
Un flujo binario almacenado en búfer que proporciona acceso de nivel superior a un flujo binario sin formato *RawIOBase* que se puede buscar. 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*.

BufferedRandom es capaz de todo lo que puede hacer *BufferedReader* o *BufferedWriter*. Adicionalmente, se garantiza implementar `seek()` y `tell()`.

class io.**BufferedRWPair**(*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*, /)
Un flujo binario almacenado en búfer que proporciona acceso de nivel superior a dos flujos binarios sin procesar *RawIOBase* no buscables, uno legible y el otro escribible. 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*.

BufferedRWPair implementa todos los métodos de *BufferedIOBase* excepto por *detach()*, que lanza un *UnsupportedOperation*.

> **Advertencia:** *BufferedRWPair* no intenta sincronizar accesos al *stream* sin formato subyacente. No debes pasar el mismo objeto como legible y escribible; usa *BufferedRandom* en su lugar.

E/S Texto

class io.**TextIOBase**
Base class for text streams. This class provides a character and line based interface to stream I/O. It inherits *IOBase*.

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*)
Un flujo de texto almacenado en búfer que proporciona acceso de nivel superior a un flujo binario almacenado en búfer *BufferedIOBase*. Hereda *TextIOBase*.

encoding proporciona el nombre de la codificación con la que se decodificará o codificará el flujo. Su valor predeterminado es *locale.getpreferredencoding(False)*. `encoding="locale"` se puede utilizar para especificar explícitamente la codificación de la configuración regional actual. Consulte *Codificación de texto* para obtener más información.

errors es una cadena de caracteres opcional que especifica cómo se manejan los errores codificados y descodificados. 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 validas.

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 de la secuencia, si *newline* es `None`, el modo *universal newlines* está habilitado. Las líneas de la entrada pueden terminar en `'\n'`, `'\r'` o `'\r\n'`, y se traducen a `'\n'` antes de retornarse al llamador. Si *newline* es `''`, el modo de salto de línea universal está habilitado, pero los finales de línea se retornan a la persona que llama sin traducir. Si *newline* tiene alguno de los otros valores legales, las líneas de entrada solo terminan con la cadena dada y la línea final se retorna al llamador 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.

Distinto en la versión 3.10: El argumento *encoding* ahora admite el nombre de codificación ficticia `"locale"`.

TextIOWrapper proporciona estos atributos y métodos de datos además de los de *TextIOBase* y *IOBase*:

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=None*, *errors=None*, *newline=None*, *line_buffering=None*, *write_through=None*)
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'*)
Una secuencia de texto que utiliza un búfer de texto en memoria. Hereda *TextIOBase*.

El búfer de texto se descarta 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 el de la clase *TextIOWrapper*, excepto que cuando escribe al flujo de salida, si *newline* es `None`, se escriben líneas nuevas como `\n` en todas las plataformas

StringIO proporciona este método además de los de *TextIOBase* y *IOBase*:

getvalue()
Retorna un `str` que contiene el contenido entero de los búfer. Nuevas lineas 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*.

16.2.5 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 subprocesos en la medida en que las llamadas al sistema operativo(como `read(2)` en Unix) que envuelven también son seguros para subprocesos.

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 subprocesos.

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 anterior se extiende implícitamente a los archivos de texto, ya que la función *open()* envolverá un objeto almacenado en búfer dentro de un *TextIOWrapper*. Esto incluye transmisiones estándar y, por lo tanto, también afecta a la función *print()* incorporada.

16.3 `time` — Acceso a tiempo 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 funciones C de la biblioteca de la plataforma 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 intercalares 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 muy adelante 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.
- La 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 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	*struct_time* en UTC	*gmtime()*
segundos desde la época	*struct_time* en hora local	*localtime()*
struct_time en UTC	segundos desde la época	*calendar.timegm()*
struct_time en hora local	segundos desde la época	*mktime()*

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.**pthread_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 *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*.

Utilice *clock_gettime_ns()* para evitar la pérdida de precisión que causa el tipo *float*.

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*.

Utilice *clock_settime_ns()* para evitar la pérdida de precisión que causa el tipo *float*.

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()*

El resultado tiene los siguientes atributos:

- *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 un *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 lanzará *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*
Retorna el valor (en segundos fraccionarios) de un reloj monótono, es decir, un reloj que no puede retroceder. El reloj no se ve afectado por las actualizaciones del reloj del sistema. El punto de referencia del valor retornado no está definido, de modo que sólo la diferencia entre los resultados de dos llamadas es válida.

Utilice *monotonic_ns()* para evitar la pérdida de precisión que causa el tipo *float*.

Nuevo en la versión 3.3.

Distinto en la versión 3.5: La función ahora está siempre disponible y siempre en todo el sistema.

Distinto en la versión 3.10: En macOS, la función ahora es para 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*
Retorna el valor (en fracciones de segundo) de un contador de rendimiento, es decir, un reloj con la resolución más alta disponible para medir una corta duración. Incluye el tiempo transcurrido durante el sueño y abarca todo el sistema. El punto de referencia del valor retornado no está definido, de modo que sólo la diferencia entre los resultados de dos llamadas es válida.

Utilice la función *perf_counter_ns()* para evitar la pérdida de precisión que causa el tipo *float*.

Nuevo en la versión 3.3.

Distinto en la versión 3.10: En Windows, la función ahora es para todo el sistema.

`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*
Retorna el valor (en fracciones de segundo) de la suma del sistema y el tiempo de CPU del usuario del proceso actual. No incluye el tiempo transcurrido durante el sueño. Es todo el proceso por definición. El punto de referencia del valor retornado no está definido, de modo que sólo la diferencia entre los resultados de dos llamadas es válida.

Utilice *process_time_ns()* para evitar la pérdida de precisión que causa el tipo *float*.

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 planificación 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. Se lanza un `ValueError` 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()`:

Directiva	Significado	Notas
`%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`	Time zone offset indicating a positive or negative time difference from UTC/GMT of the form +HHMM or -HHMM, where H represents decimal hour digits and M represents decimal minute digits [-23:59, +23:59].[1]	
`%Z`	Time zone name (no characters if no time zone exists). Deprecated.[1]	
`%%`	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.

[1] The use of `%Z` is now deprecated, but the `%z` escape that expands to the preferred hour/minute offset is not supported by all ANSI C libraries. Also, a strict reading of the original 1982 **RFC 822** standard calls for a two-digit year (`%y` rather than `%Y`), but practice moved to 4-digit years long before the year 2000. After that, **RFC 822** became obsolete and the 4-digit year has been first recommended by **RFC 1123** and then mandated by **RFC 2822**.

(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.

Here is an example, a format for dates compatible with that specified in the **RFC 2822** Internet email standard.[1]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

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 *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 lanza un *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, 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:

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 *strftime()* 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 lanza un *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 intercalares 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.

Utilice *time_ns()* para evitar la pérdida de precisión que causa el tipo *float*.

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.**thread_time**() → *float*
Retorna el valor (en fracciones de segundo) de la suma del sistema y el tiempo de CPU del usuario del subproceso actual. No incluye el tiempo transcurrido durante el sueño. Es específico del hilo por definición. El punto de referencia del valor retornado no está definido, de modo que sólo la diferencia entre los resultados de dos llamadas en el mismo hilo es válida.

Utilice *thread_time_ns()* para evitar la pérdida de precisión que causa el tipo *float*.

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.**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: `± 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 juliano * n * (1 <= * n * <= 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 juliano basado en cero (0 <= * n * <= 365). Los días bisiestos se cuentan y es posible referirse al 29 de febrero.

Mm.n.d El día d (0 <= *d* <= 6) de la semana *n* del mes *m* del año (1 <= *n* <= 5, 1 <= *m* <= 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 monótono 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_TAI**
Tiempo Atómico Internacional

El sistema debe contar con una tabla de segundos intercalares para que éste dé la respuesta correcta. Software PTP o NTP puede mantener una tabla de segundos intercalares.

Disponibilidad: Linux.

Nuevo en la versión 3.9.

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 monótonamente, 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

options:
 -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*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*, *exit_on_error=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* - The name of the program (default: `os.path.basename(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* - Text to display before the argument help (by default, no text)
- *epilog* - Text to display after the argument help (by default, no text)
- *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`)
- *exit_on_error* - Determina si ArgumentParser sale o no con información de error cuando se produce un error. (predeterminado: `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`.

Distinto en la versión 3.9: Se agregó el parámetro *exit_on_error*.

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]
```

(continué en la próxima página)

(proviene de la página anterior)

```
options:
 -h, --help  show this help message and exit
 --foo FOO   foo help
$ cd ..
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

options:
 -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]

options:
 -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]

options:
 -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

options:
 -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:
```

(continué en la próxima página)

(proviene de la página anterior)

```
 bar           bar help

options:
 -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

options:
 -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

options:
 -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)

>>> 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

options:
 -h, --help  show this help message and exit
```

(continué en la próxima página)

(proviene de la página anterior)

```
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
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

options:
 -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 ...]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

options:
 -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
```

(continué en la próxima página)

(proviene de la página anterior)

```
positional arguments:
  float

options:
  -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

Sometimes, for example when dealing with a particularly long argument list, it may make sense to keep the list of arguments in a file rather than typing it out at the command line. If the `fromfile_prefix_chars=` argument is given to the *ArgumentParser* constructor, then arguments that start with any of the specified characters will be treated as files, and will be replaced by the arguments they contain. For example:

```
>>> with open('args.txt', 'w') as fp:
...     fp.write('-f\nbar')
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

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=` a *ArgumentParser*. 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='?')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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]

options:
 -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]

options:
 -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]

options:
 --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]

options:
  +h, ++help  show this help message and exit
```

exit_on_error

Normalmente, cuando pasa una lista de argumentos no válidos al método *parse_args()* de un *ArgumentParser*, saldrá con información de error.

Si el usuario desea detectar errores manualmente, la función se puede habilitar configurando `exit_on_error` en `False`

```
>>> parser = argparse.ArgumentParser(exit_on_error=False)
>>> parser.add_argument('--integers', type=int)
_StoreAction(option_strings=['--integers'], dest='integers', nargs=None,␣
↪const=None, default=None, type=<class 'int'>, choices=None, help=None,␣
↪metavar=None)
>>> try:
...     parser.parse_args('--integers a'.split())
... except argparse.ArgumentError:
...     print('Catching an argumentError')
...
Catching an argumentError
```

Nuevo en la versión 3.9.

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 producido si el argumento está ausente en la línea de comando y si está ausente en el objeto de espacio de nombres.
- *type* - El tipo al que debe convertirse el argumento de la línea de comandos.
- *choices* - A sequence of the allowable values for the argument.
- *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. Esta 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')
>>> 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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 puede especificar una acción arbitraria pasando una subclase de Acción u otro objeto que implemente la misma interfaz. `BooleanOptionalAction` está disponible en `argparse` y agrega soporte para acciones booleanas como `--foo` y `--no-foo`:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=argparse.BooleanOptionalAction)
>>> parser.parse_args(['--no-foo'])
Namespace(foo=False)
```

Nuevo en la versión 3.9.

La forma recomendada de crear una acción personalizada es extender *Action*, anulando el método `__call__` y, opcionalmente, los métodos `__init__` y `format_usage`.

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'])
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
```

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 espacio de nombres de destino ya tiene un atributo establecido, la acción *default* no lo sobrescribirá:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args([], namespace=argparse.Namespace(foo=101))
Namespace(foo=101)
```

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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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

De forma predeterminada, el analizador lee los argumentos de la línea de comandos como cadenas simples. Sin embargo, a menudo, la cadena de la línea de comandos debe interpretarse como otro tipo, como *`float`* o *`int`*. La palabra clave `type` para *`add_argument()`* permite realizar cualquier verificación de tipo y conversión de tipo necesaria.

Si la palabra clave *type* se usa con la palabra clave *default*, el convertidor de tipos solo se aplica si el valor predeterminado es una cadena de caracteres.

El argumento para `type` puede ser cualquier invocable que acepte una sola cadena de caracteres. Si la función lanza `ArgumentTypeError`, *`TypeError`*, o *`ValueError`*, se detecta la excepción y se muestra un mensaje de error con un formato agradable. No se manejan otros tipos de excepciones.

Los tipos y funciones incorporados comunes se pueden utilizar como convertidores de tipos:

```
import argparse
import pathlib

parser = argparse.ArgumentParser()
parser.add_argument('count', type=int)
parser.add_argument('distance', type=float)
parser.add_argument('street', type=ascii)
parser.add_argument('code_point', type=ord)
parser.add_argument('source_file', type=open)
parser.add_argument('dest_file', type=argparse.FileType('w', encoding='latin-1'))
parser.add_argument('datapath', type=pathlib.Path)
```

Las funciones definidas por el usuario también se pueden utilizar:

```
>>> def hyphenated(string):
...     return '-'.join([word[:4] for word in string.casefold().split()])
...
>>> parser = argparse.ArgumentParser()
>>> _ = parser.add_argument('short_title', type=hyphenated)
>>> parser.parse_args(['"The Tale of Two Cities"'])
Namespace(short_title='"the-tale-of-two-citi')
```

La función *bool()* no se recomienda como convertidor de tipos. Todo lo que hace es convertir cadenas de caracteres vacías en `False` y cadenas de caracteres no vacías en `True`. Por lo general, esto no es lo que se desea.

En general, la palabra clave `type` es una conveniencia que solo debe usarse para conversiones simples que solo pueden generar una de las tres excepciones admitidas. Cualquier cosa con un manejo de errores o de recursos más interesante debe hacerse en sentido descendente después de analizar los argumentos.

Por ejemplo, las conversiones JSON o YAML tienen casos de error complejos que requieren mejores informes que los que puede proporcionar la palabra clave `type`. Un *JSONDecodeError* no estaría bien formateado y una excepción `FileNotFound` no se manejaría en absoluto.

Even *FileType* tiene sus limitaciones para su uso con la palabra clave `type`. Si un argumento usa *FileType* y luego falla un argumento posterior, se informa un error pero el archivo no se cierra automáticamente. En este caso, sería mejor esperar hasta que se haya ejecutado el analizador y luego usar la declaración `with` para administrar los archivos.

Para los verificadores de tipo que simplemente verifican un conjunto fijo de valores, considere usar la palabra clave *choice* en su lugar.

choices

Some command-line arguments should be selected from a restricted set of values. These can be handled by passing a sequence object as the *choices* keyword argument to *add_argument()*. When the command line is parsed, argument values will be checked, and an error message will be displayed if the argument was not one of the acceptable values:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

Note that inclusion in the *choices* sequence is checked after any *type* conversions have been performed, so the type of the objects in the *choices* sequence should match the *type* specified:

```
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

Any sequence can be passed as the *choices* value, so *list* objects, *tuple* objects, and custom sequences are all supported.

No se recomienda el uso de *enum.Enum* porque es difícil controlar su apariencia en el uso, la ayuda y los mensajes de error.

Las opciones formateadas anulan el *metavar* predeterminado que normalmente se deriva de *dest*. Esto suele ser lo que desea porque el usuario nunca ve el parámetro *dest*. Si esta visualización no es deseable (quizás porque hay muchas opciones), simplemente especifique un *metavar* explícito.

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

options:
 -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)

options:
 -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]

options:
  -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

options:
 -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:
 XXX

options:
 -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]

options:
 -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 subclases 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 estable atributos en `namespace` basados en `dest` y `values`.

Las subclases de acción pueden definir un método `format_usage` que no toma ningún argumento y devuelve una cadena que se utilizará al imprimir el uso del programa. Si no se proporciona dicho método, se utilizará un valor predeterminado razonable.

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')

>>> # 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='?')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> # 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_strings*][, *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} ...

positional arguments:
  {a,b}   sub-command help
    a     a help
    b     b help

options:
  -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

options:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]
```

(continué en la próxima página)

(proviene de la página anterior)

```
options:
  -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} ...

options:
  -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(required=True)
>>>
>>> # 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')
```

(continué en la próxima página)

(proviene de la página anterior)

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

FileType objects understand the pseudo-argument `'-'` and automatically convert this into *sys.stdin* for readable *FileType* objects and *sys.stdout* for writable *FileType* objects:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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[:]`. *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 `--fo` 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` — Instalación de logging para Python

Source code: Lib/logging/__init__.py

Important

Esta página contiene la información de referencia de la API. Para información sobre tutorial y discusión de temas más avanzados, ver

- Tutorial básico
- Tutorial avanzado
- Libro de recetas de Logging

Este módulo define funciones y clases que implementan un sistema flexible de logging de eventos para aplicaciones y bibliotecas.

El beneficio clave de tener la API de logging proporcionada por un módulo de la biblioteca estándar es que todos los módulos de Python pueden participar en el logging, por lo que el registro de su aplicación puede incluir sus propios mensajes integrados con mensajes de módulos de terceros.

The simplest example:

```
>>> import logging
>>> logging.warning('Watch out!')
WARNING:root:Watch out!
```

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 view the tutorials (**see the links above and on the right**).

Las clases básicas definidas por el módulo, junto con sus funciones, se enumeran a continuación.

- Los loggers exponen la interfaz que el código de la aplicación usa directamente.
- Los gestores envían los registros (creados por los loggers) al destino apropiado.
- Los filtros proporcionan una facilidad de ajuste preciso para determinar que registros generar.
- Los formateadores especifican el diseño de los registros en el resultado final.

16.6.1 Objetos Logger

Los loggers tienen los siguientes atributos y métodos. Tenga en cuenta que los loggers *NUNCA* deben ser instanciados directamente, siempre a través de la función de nivel de módulo `logging.getLogger(name)`. Múltiples llamadas a *`getLogger()`* con el mismo nombre siempre retornarán una referencia al mismo objeto Logger.

El `name` es potencialmente un valor jerárquico separado por puntos, como `foo.bar.baz` (aunque también podría ser simplemente `foo, por ejemplo). Los loggers que están más abajo en la lista jerárquica son hijos de los loggers que están más arriba en la lista. Por ejemplo, dado un logger con el nombre de `foo`, los logger con los nombres de `foo.bar`, `foo.bar.baz` y `foo.bam` son descendientes de `foo`. La jerarquía del nombre del logger es análoga a la jerarquía del paquete Python e idéntica si organiza los logger por módulo utilizando la construcción recomendada `logging.getLogger(__name__)`. Debido que en un módulo, `__name__` es el nombre del módulo en el espacio de nombres del paquete Python.

class logging.**Logger**

propagate
Si este atributo se evalúa como verdadero, los eventos registrados en este logger se pasarán a los gestores de los loggers de nivel superior (ancestro), además de los gestores asociados a este logger. Los mensajes

se pasan directamente a los gestores de los loggers ancestrales; no se consideran ni el nivel ni los filtros de los loggers ancestrales en cuestión.

Si esto se evalúa como falso, los mensajes de registro no se pasan a los gestores de los logger ancestrales.

Spelling it out with an example: If the propagate attribute of the logger named `A.B.C` evaluates to true, any event logged to `A.B.C` via a method call such as `logging.getLogger('A.B.C').error(...)` will [subject to passing that logger's level and filter settings] be passed in turn to any handlers attached to loggers named `A.B`, `A` and the root logger, after first being passed to any handlers attached to `A.B.C`. If any logger in the chain `A.B.C`, `A.B`, `A` has its `propagate` attribute set to false, then that is the last logger whose handlers are offered the event to handle, and propagation stops at that point.

El constructor establece este atributo en `True`.

Nota: Si adjunta un controlador a un logger *y* uno o más de sus ancestros, puede emitir el mismo registro varias veces. En general, no debería necesitar adjuntar un gestor a más de un logger; si solo lo adjunta al logger apropiado que está más arriba en la jerarquía del logger, verá todos los eventos registrados por todos los logger descendientes, siempre que la configuración de propagación sea `True`. Un escenario común es adjuntar gestores solo al logger raíz y dejar que la propagación se encargue del resto.

---

setLevel(*level*)
Establece el umbral para este logger en *level*. Los mensajes de logging que son menos severos que *level* serán ignorados; los mensajes de logging que tengan un nivel de severidad *level* o superior serán emitidos por cualquier gestor o gestores que atiendan este logger, a menos que el nivel de un gestor haya sido configurado en un nivel de severidad más alto que *level*.

Cuando se crea un logger, el nivel se establece en `NOTSET` (que hace que todos los mensajes se procesen cuando el logger es el logger raíz, o la delegación al padre cuando el logger no es un logger raíz). Tenga en cuenta que el logger raíz se crea con el nivel `WARNING`.

El término "delegación al padre" significa que si un logger tiene un nivel de NOTSET, su cadena de logger ancestrales se atraviesa hasta que se encuentra un ancestro con un nivel diferente a NOTSET o se alcanza la raíz.

Si se encuentra un antepasado con un nivel distinto de NOTSET, entonces el nivel de ese antepasado se trata como el nivel efectivo del logger donde comenzó la búsqueda de antepasados, y se utiliza para determinar cómo se maneja un evento de registro.

Si se alcanza la raíz y tiene un nivel de NOTSET, se procesarán todos los mensajes. De lo contrario, el nivel de la raíz se utilizará como el nivel efectivo.

Ver *Niveles de logging* para obtener una lista de niveles.

Distinto en la versión 3.2: El parámetro *level* ahora acepta una representación de cadena del nivel como "INFO" como alternativa a las constantes de enteros como `INFO`. Sin embargo, tenga en cuenta que los niveles se almacenan internamente como e.j. *`getEffectiveLevel()`* y *`isEnabledFor()`* retornará/esperará que se pasen enteros.

isEnabledFor(*level*)
Indica si este logger procesará un mensaje de gravedad *level*. Este método verifica primero el nivel de nivel de módulo establecido por `logging.disable(level)` y luego el nivel efectivo del logger según lo determinado por *`getEffectiveLevel()`*.

getEffectiveLevel()
Indica el nivel efectivo para este logger. Si se ha establecido un valor distinto de `NOTSET` utilizando *`setLevel()`*, se retorna. De lo contrario, la jerarquía se atraviesa hacia la raíz hasta que se encuentre un valor que no sea `NOTSET` y se retorna ese valor. El valor retornado es un entero, típicamente uno de `logging.DEBUG`, `logging.INFO` etc.

getChild(*suffix*)
Retorna un logger que es descendiente de este logger, según lo determinado por el sufijo. Por lo tanto, `logging.getLogger('abc').getChild('def.ghi')` retornaría el mismo logger que

retornaría `logging.getLogger('abc.def.ghi')`. Este es un método convenientemente útil cuando el logger principal se nombra usando e.j. `__name__` en lugar de una cadena literal.

Nuevo en la versión 3.2.

debug (*msg*, **args*, ***kwargs*)

Registra un mensaje con el nivel `DEBUG` en este logger. El *msg* es la cadena de formato del mensaje, y los *args* son los argumentos que se fusionan en *msg* utilizando el operador de formato de cadena. (Tenga en cuenta que esto significa que puede usar palabras clave en la cadena de formato, junto con un solo argumento de diccionario). No se realiza ninguna operación de formateo % en *msg* cuando no se suministran *args*.

Hay cuatro argumentos de palabras clave *kwargs* que se inspeccionan: *exc_info*, *stack_info*, *stacklevel* y *extra*.

Si *exc_info* no se evalúa como falso, hace que se agregue información de excepción al mensaje de registro. Si se proporciona una tupla de excepción (en el formato retornado por *sys.exc_info()*) o se proporciona una instancia de excepción, se utiliza; de lo contrario, se llama a *sys.exc_info()* para obtener la información de excepción.

El segundo argumento opcional con la palabra clave *stack_info*, que por defecto es `False`. Si es verdadero, la información de la pila agregara el mensaje de registro, incluida la actual llamada del registro. Tenga en cuenta que esta no es la misma información de la pila que se muestra al especificar *exc_info*: la primera son los cuadros de la pila desde la parte inferior de la pila hasta la llamada de registro en el hilo actual, mientras que la segunda es la información sobre los cuadros de la pila que se han desenrollado, siguiendo una excepción, mientras busca gestores de excepción.

Puede especificar *stack_info* independientemente de *exc_info*, por ejemplo solo para mostrar cómo llegaste a cierto punto en tu código, incluso cuando no se lanzaron excepciones. Los marcos de la pila se imprimen siguiendo una línea de encabezado que dice:

```
Stack (most recent call last):
```

Esto imita el `Traceback (most recent call last):` que se usa cuando se muestran marcos de excepción.

El tercer argumento opcional con la palabra clave *stacklevel*, que por defecto es `1`. Si es mayor que 1, se omite el número correspondiente de cuadros de pila al calcular el número de línea y el nombre de la función establecidos en *LogRecord* creado para el evento de registro. Esto se puede utilizar en el registro de ayudantes para que el nombre de la función, el nombre de archivo y el número de línea registrados no sean la información para la función/método auxiliar, sino más bien su llamador. El nombre de este parámetro refleja el equivalente en el modulo *warnings*.

El cuarto argumento de palabra clave es *extra*, que se puede usar para pasar un diccionario que se usa para completar el __dict__ de *LogRecord* creado para el evento de registro con atributos definidos por el usuario. Estos atributos personalizados se pueden usar a su gusto. Podrían incorporarse en mensajes registrados. Por ejemplo:

```
FORMAT = '%(asctime)s %(clientip)-15s %(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)
```

imprimiría algo como

```
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 section on *Atributos LogRecord* for more information on which keys are used by the logging system.)

Si elige usar estos atributos en los mensajes registrados, debe tener cuidado. En el ejemplo anterior, se ha configurado *Formatter* con una cadena de formato que espera *clientip* y "usuario" en el diccionario

de atributos de *LogRecord*. Si faltan, el mensaje no se registrará porque se producirá una excepción de formato de cadena. En este caso, siempre debe pasar el diccionario *extra* con estas teclas.

Si bien esto puede ser molesto, esta función está diseñada para su uso en circunstancias especializadas, como servidores de subprocesos múltiples donde el mismo código se ejecuta en muchos contextos, y las condiciones interesantes que surgen dependen de este contexto (como la dirección IP del cliente remoto y autenticado nombre de usuario, en el ejemplo anterior). En tales circunstancias, es probable que se especialice *Formatter*s con particular *Handler*s.

If no handler is attached to this logger (or any of its ancestors, taking into account the relevant *Logger.propagate* attributes), the message will be sent to the handler set on *lastResort*.

Distinto en la versión 3.2: Se agregó el parámetro *stack_info*.

Distinto en la versión 3.5: El parámetro *exc_info* ahora puede aceptar instancias de excepción.

Distinto en la versión 3.8: Se agregó el parámetro *stacklevel*.

info(*msg*, **args*, ***kwargs*)
Registra un mensaje con el nivel `INFO` en este logger. Los argumentos se interpretan como *debug()*.

warning(*msg*, **args*, ***kwargs*)
Registra un mensaje con el nivel `WARNING` en este logger. Los argumentos se interpretan como *debug()*.

Nota: Hay un método obsoleto `warn` que es funcionalmente idéntico a `warning`. Como `warn` está en desuso, no lo use, use `warning` en su lugar.

---

error(*msg*, **args*, ***kwargs*)
Registra un mensaje con nivel `ERROR` en este logger. Los argumentos se interpretan como *debug()*.

critical(*msg*, **args*, ***kwargs*)
Registra un mensaje con el nivel `CRITICAL` en este logger. Los argumentos se interpretan como *debug()*.

log(*level*, *msg*, **args*, ***kwargs*)
Registra un mensaje con nivel entero *level* en este logger. Los otros argumentos se interpretan como *debug()*.

exception(*msg*, **args*, ***kwargs*)
Registra un mensaje con nivel `ERROR` en este logger. Los argumentos se interpretan como *debug()*. La información de excepción se agrega al mensaje de registro. Este método solo debe llamarse desde un gestor de excepciones.

addFilter(*filter*)
Agrega el filtro *filter* especificado a este logger.

removeFilter(*filter*)
Elimina el filtro *filter* especificado de este logger.

filter(*record*)
Aplique los filtros de este logger al registro y retorna `True` si se va a procesar el registro. Los filtros se consultan a su vez, hasta que uno de ellos retorna un valor falso. Si ninguno de ellos retorna un valor falso, el registro será procesado (pasado a los gestores). Si se retorna un valor falso, no se produce más procesamiento del registro.

addHandler(*hdlr*)
Agrega el gestor especificado *hdlr* a este logger.

removeHandler(*hdlr*)
Elimina el gestor especificado *hdlr* de este logger.

findCaller(*stack_info=False*, *stacklevel=1*)
Encuentra el nombre de archivo de origen de la invoca y el número de línea. Retorna el nombre del

archivo, el número de línea, el nombre de la función y la información de la pila como una tupla de 4 elementos. La información de la pila se retorna como `None` a menos que *stack_info* sea `True`.

El parámetro *stacklevel* se pasa del código que llama a *debug()* y otras API. Si es mayor que 1, el exceso se utiliza para omitir los cuadros de la pila antes de determinar los valores que se retornarán. Esto generalmente será útil al llamar a las API de registro desde un *helper/wrapper*, de modo que la información en el registro de eventos no se refiera al *helper/wrapper*, sino al código que lo llama.

handle(*record*)
Gestiona un registro pasándolo a todos los gestores asociados con este logger y sus antepasados (hasta que se encuentre un valor falso de *propagar*). Este método se utiliza para registros no empaquetados recibidos de un socket, así como para aquellos creados localmente. El filtrado a nivel de logger se aplica usando *filter()*.

makeRecord(*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)
Este es un método *factory* que se puede sobreescribir en subclases para crear instancias especializadas *LogRecord*.

hasHandlers()
Comprueba si este logger tiene algún controlador configurado. Esto se hace buscando gestores en este logger y sus padres en la jerarquía del logger. Retorna `True` si se encontró un gestor, de lo contrario, `False` . El método deja de buscar en la jerarquía cada vez que se encuentra un logger con el atributo *propagate* establecido en falso: ese será el último logger que verificará la existencia de gestores.

Nuevo en la versión 3.2.

Distinto en la versión 3.7: Los logger ahora se pueden serializar y deserializar (*pickled and unpickled*).

16.6.2 Niveles de logging

Los valores numéricos de los niveles de logging se dan en la siguiente tabla. Estos son principalmente de interés si desea definir sus propios niveles y necesita que tengan valores específicos en relación con los niveles predefinidos. Si define un nivel con el mismo valor numérico, sobrescribe el valor predefinido; se pierde el nombre predefinido.

Nivel	Valor numérico
`CRITICAL`	50
`ERROR`	40
`WARNING`	30
`INFO`	20
`DEBUG`	10
`NOTSET`	0

16.6.3 Gestor de objetos

Los gestores tienen los siguientes atributos y métodos. Tenga en cuenta que *Handler* nunca se instancia directamente; Esta clase actúa como base para subclases más útiles. Sin embargo, el método `__init__()` en las subclases debe llamar a *Handler.__init__()*.

class logging.**Handler**

__init__(*level=NOTSET*)
Inicializa la instancia *Handler* estableciendo su nivel, configurando la lista de filtros en la lista vacía y creando un bloqueo (usando *createLock()*) para serializar el acceso a un mecanismo de E/S.

createLock()
Inicializa un bloqueo de subprocesos que se puede utilizar para serializar el acceso a la funcionalidad de E/S subyacente que puede no ser segura para subprocesos.

acquire()
Adquiere el bloqueo de hilo creado con *createLock()*.

release()
Libera el bloqueo de hilo adquirido con *acquire()*.

setLevel(*level*)
Establece el umbral para este gestor en *level*. Los mensajes de registro que son menos severos que *level* serán ignorados. Cuando se crea un controlador, el nivel se establece en NOTSET (lo que hace que se procesen todos los mensajes).

Ver *Niveles de logging* para obtener una lista de niveles.

Distinto en la versión 3.2: El parámetro *level* ahora acepta una representación de cadena del nivel como "INFO" como alternativa a las constantes de enteros como INFO.

setFormatter(*fmt*)
Establece *Formatter* para este controlador en *fmt*.

addFilter(*filter*)
Agrega el filtro *filter* especificado a este gestor.

removeFilter(*filter*)
Elimina el filtro especificado *filter* de este gestor.

filter(*record*)
Aplique los filtros de este gestor al registro y retorna True si se va a procesar el registro. Los filtros se consultan a su vez, hasta que uno de ellos retorna un valor falso. Si ninguno de ellos retorna un valor falso, se emitirá el registro. Si uno retorna un valor falso, el controlador no emitirá el registro.

flush()
Asegúrese de que toda la salida de logging se haya vaciado. Esta versión no hace nada y está destinada a ser implementada por subclases.

close()
Poner en orden los recursos utilizados por el gestor. Esta versión no genera salida, pero elimina el controlador de una lista interna de gestores que se cierra cuando se llama a *shutdown()*. Las subclases deben garantizar que esto se llame desde métodos *close()* sobreescritos.

handle(*record*)
Emite condicionalmente el registro especifico, según los filtros que se hayan agregado al controlador. Envuelve la actual emisión del registro con *acquisition/release* del hilo de bloqueo E/S.

handleError(*record*)
Este método debe llamarse desde los gestores cuando se encuentra una excepción durante una llamada a *emit()*. Si el atributo de nivel de módulo raiseExceptions es " False", las excepciones se ignoran silenciosamente. Esto es lo que más se necesita para un sistema de registro: a la mayoría de los usuarios no les importan los errores en el sistema de registro, están más interesados en los errores de la aplicación. Sin embargo, puede reemplazar esto con un gestor personalizado si lo desea. El registro especificado es el que se estaba procesando cuando se produjo la excepción. (El valor predeterminado de raiseExceptions es " True", ya que es más útil durante el desarrollo).

format(*record*)
Formato para un registro - si se configura un formateador, úselo. De lo contrario, use el formateador predeterminado para el módulo.

emit(*record*)
Haga lo que sea necesario para registrar de forma especifica el registro. Esta versión está destinada a ser implementada por subclases y, por lo tanto, lanza un *NotImplementedError*.

> **Advertencia:** This method is called after a handler-level lock is acquired, which is released after this method returns. When you override this method, note that you should be careful when calling anything that invokes other parts of the logging API which might do locking, because that might result in a deadlock. Specifically:
>
> - Logging configuration APIs acquire the module-level lock, and then individual handler-level locks as those handlers are configured.

- Many logging APIs lock the module-level lock. If such an API is called from this method, it could cause a deadlock if a configuration call is made on another thread, because that thread will try to acquire the module-level lock *before* the handler-level lock, whereas this thread tries to acquire the module-level lock *after* the handler-level lock (because in this method, the handler-level lock has already been acquired).

Para obtener una lista de gestores incluidos como estándar, consulte *logging.handlers*.

16.6.4 Objetos formateadores

Formatter tiene los siguientes atributos y métodos. Son responsables de convertir una *LogRecord* a (generalmente) una cadena que puede ser interpretada por un sistema humano o externo. La base *Formatter* permite especificar una cadena de formato. Si no se proporciona ninguno, se utiliza el valor predeterminado de `'%(message)s'`, que solo incluye el mensaje en la llamada de registro. Para tener elementos de información adicionales en la salida formateada (como una marca de tiempo), siga leyendo.

Un formateador se puede inicializar con una cadena de formato que utiliza el conocimiento de los atributos *LogRecord*, como el valor predeterminado mencionado anteriormente, haciendo uso del hecho de que el mensaje y los argumentos del usuario están formateados previamente en *LogRecord*'s con *message* como atributo. Esta cadena de formato contiene claves de mapeo de Python %-style estándar. Ver la sección *Formateo de cadenas al estilo *printf** para obtener más información sobre el formato de cadenas.

Las claves de mapeo útiles en a *LogRecord* se dan en la sección sobre *Atributos LogRecord*.

class logging.**Formatter**(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*, ***, *defaults=None*)

Retorna una nueva instancia de *Formatter*. La instancia se inicializa con una cadena de formato para el mensaje en su conjunto, así como una cadena de formato para la porción fecha/hora de un mensaje. Si no se especifica *fmt*, se utiliza `'%(message)s'`. Si no se especifica *datefmt*, se utiliza un formato que se describe en la documentación *formatTime()*.

El parámetro *style* puede ser uno de "%", "{"" o "$" y determina cómo se fusionará la cadena de formato con sus datos: usando uno de %-formatting, *str.format()* o *string.Template*. Esto solo aplica al formato de cadenas de caracteres *fmt* (e.j. `'%(message)s'` o `{message}`), no al mensaje pasado actualmente al `Logger.debug` etc; ver formatting-styles para más información sobre usar {- y formateado-$ para mensajes de log.

El parámetro *defaults* puede ser un diccionario con valores por defecto para usar en campos personalizados. Por ejemplo: `logging.Formatter('%(ip)s %(message)s', defaults={"ip": None})`

Distinto en la versión 3.2: Se agregó el parámetro *style*.

Distinto en la versión 3.8: Se agregó el parámetro *validate*. Si el estilo es incorrecto o no coincidente, *fmt* lanzará un `ValueError`. Por ejemplo: `logging.Formatter('%(asctime)s - %(message)s', style='{')`.

Distinto en la versión 3.10: Se agregó el parámetro *defaults*.

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 (by setting the *exc_text* attribute to `None`) 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.

Si la información de la pila está disponible, se agrega después de la información de la excepción, usando *formatStack()* para transformarla si es necesario.

formatTime(*record*, *datefmt=None*)

Este método debe ser llamado desde *format()* por un formateador que espera un tiempo formateado . Este método se puede reemplazar en formateadores para proporcionar cualquier requisito específico, pero el comportamiento básico es el siguiente: if *datefmt* (una cadena), se usa con *time.strftime()* para formatear el tiempo de creación del registro De lo contrario, se utiliza el formato "%Y-%m-%d %H:%M:%S,uuu", donde la parte *uuu* es un valor de milisegundos y las otras letras son *time.strftime()* . Un ejemplo de tiempo en este formato es `2003-01-23 00:29:50,411`. Se retorna la cadena resultante.

Esta función utiliza una función configurable por el usuario para convertir el tiempo de creación en una tupla. Por defecto, se utiliza *time.localtime()*; Para cambiar esto para una instancia de formateador particular, se agrega el atributo `converter` en una función con igual firma como *time.localtime()* o *time.gmtime()*. Para cambiarlo en todos los formateadores, por ejemplo, si desea que todos los tiempos de registro se muestren en GMT, agregue el atributo `converter` en la clase `Formatter`.

Distinto en la versión 3.3: Anteriormente, el formato predeterminado estaba codificado como en este ejemplo: `2010-09-06 22:38:15,292` donde la parte anterior a la coma es manejada por una cadena de formato strptime (`'%Y-%m-%d %H:%M:%S'`), y la parte después de la coma es un valor de milisegundos. Debido a que strptime no tiene una posición de formato para milisegundos, el valor de milisegundos se agrega usando otra cadena de formato, `'%s,%03d'`— ambas cadenas de formato se han codificado en este método. Con el cambio, estas cadenas se definen como atributos de nivel de clase que pueden *overridden* a nivel de instancia cuando se desee. Los nombres de los atributos son `default_time_format` (para una cadena de formato strptime) y `default_msec_format` (para agregar el valor de milisegundos).

Distinto en la versión 3.9: El formato `default_msec_format` puede ser `None`.

formatException(*exc_info*)

Formatea la información de una excepción especificada (una excepción como una tupla estándar es retornada por *sys.exc_info()*) como una cadena. Esta implementación predeterminada solo usa *traceback.print_exception()*. La cadena resultantes retornada.

formatStack(*stack_info*)

Formatea la información de una pila especificada (una cadena es retornada por *traceback.print_stack()*, pero con la ultima línea removida) como una cadena. Esta implementación predeterminada solo retorna el valor de entrada.

class logging.**BufferingFormatter**(*linefmt=None*)

A base formatter class suitable for subclassing when you want to format a number of records. You can pass a *Formatter* instance which you want to use to format each line (that corresponds to a single record). If not specified, the default formatter (which just outputs the event message) is used as the line formatter.

formatHeader(*records*)

Return a header for a list of *records*. The base implementation just returns the empty string. You will need to override this method if you want specific behaviour, e.g. to show the count of records, a title or a separator line.

formatFooter(*records*)

Return a footer for a list of *records*. The base implementation just returns the empty string. You will need to override this method if you want specific behaviour, e.g. to show the count of records or a separator line.

format(*records*)

Return formatted text for a list of *records*. The base implementation just returns the empty string if there are no records; otherwise, it returns the concatenation of the header, each record formatted with the line formatter, and the footer.

16.6.5 Filtro de Objetos

Los Manejadores y los Registradores pueden usar los Filtros para un filtrado más sofisticado que el proporcionado por los niveles. La clase de filtro base solo permite eventos que están por debajo de cierto punto en la jerarquía del logger. Por ejemplo, un filtro inicializado con "A.B" permitirá los eventos registrados por los logger "A.B", "A.B.C", "A.B.C.D", "A.B.D" etc., pero no "A.BB", "B.A.B", etc. Si se inicializa con una cadena vacía, se pasan todos los eventos.

class logging.**Filter**(*name=''*)

Retorna una instancia de la clase *Filter*. Si se especifica *name*, nombra un logger que, junto con sus hijos, tendrá sus eventos permitidos a través del filtro. Si *name* es una cadena vacía, permite todos los eventos.

filter(*record*)

¿Se apuntará el registro especificado? Retorna cero para no, distinto de cero para sí. Si se considera apropiado, el registro puede modificarse in situ mediante este método.

Tenga en cuenta que los filtros adjuntos a los gestores se consultan antes de que el gestor emita un evento, mientras que los filtros adjuntos a los loggers se consultan cada vez que se registra un evento (usando *debug()*, *info()*, etc.), antes de enviar un evento a los gestores. Esto significa que los eventos que han sido generados por loggers descendientes no serán filtrados por la configuración del filtro del logger, a menos que el filtro también se haya aplicado a esos loggers descendientes.

En realidad, no se necesita la subclase Filtro: se puede pasar cualquier instancia que tenga un método de filter con la misma semántica.

Distinto en la versión 3.2: No es necesario crear clases especializadas de Filter ni usar otras clases con un método filter: puede usar una función (u otra invocable) como filtro. La lógica de filtrado verificará si el objeto de filtro tiene un atributo filter: si lo tiene, se asume que es un Filter y se llama a su método *filter()*. De lo contrario, se supone que es invocable y se llama con el registro como único parámetro. El valor retornado debe ajustarse al retornado por *filter()*.

Aunque los filtros se utilizan principalmente para filtrar registros basados en criterios más sofisticados que los niveles, son capaces de ver cada registro que es procesado por el gestor o logger al que están adjuntos: esto puede ser útil si desea hacer cosas como contar cuántos registros fueron procesados por un logger o gestor en particular, o agregando, cambiando o quitando atributos en *LogRecord* que se está procesando. Obviamente, el cambio de LogRecord debe hacerse con cierto cuidado, pero permite la inyección de información contextual en los registros (ver filters-contextual).

16.6.6 Objetos LogRecord

Las instancias *LogRecord* son creadas automáticamente por *Logger* cada vez que se registra algo, y se pueden crear manualmente a través de *makeLogRecord()* (por ejemplo, a partir de un evento serializado (*pickled*) recibido en la transmisión).

class logging.**LogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*)

Contiene toda la información pertinente al evento que se registra.

The primary information is passed in *msg* and *args*, which are combined using msg % args to create the message attribute of the record.

Parámetros

- **name** (str) – The name of the logger used to log the event represented by this LogRecord. Note that the logger name in the LogRecord will always have this value, even though it may be emitted by a handler attached to a different (ancestor) logger.
- **level** (int) – The *numeric level* of the logging event (such as 10 for DEBUG, 20 for INFO, etc). Note that this is converted to *two* attributes of the LogRecord: levelno for the numeric value and levelname for the corresponding level name.
- **pathname** (str) – The full string path of the source file where the logging call was made.

- **lineno** (int) – El número de línea en el archivo de origen donde se realizó la llamada logging.
- **msg** (*typing.Any*) – The event description message, which can be a %-format string with placeholders for variable data, or an arbitrary object (see arbitrary-object-messages).
- **args** (*tuple | dict[str, typing.Any]*) – Datos variables para fusionar en el argumento *msg* para obtener la descripción del evento.
- **exc_info** (tuple[type[BaseException], BaseException, types.TracebackType] | *None*) – An exception tuple with the current exception information, as returned by *sys.exc_info()*, or None if no exception information is available.
- **func** (*str | None*) – El nombre de la función o método desde el que se invocó la llamada de logging.
- **sinfo** (*str | None*) – Una cadena de texto que representa la información de la pila desde la base de la pila en el hilo actual, hasta la llamada de logging.

getMessage()

Retorna el mensaje para la instancia *LogRecord* después de fusionar cualquier argumento proporcionado por el usuario con el mensaje. Si el argumento del mensaje proporcionado por el usuario para la llamada de logging no es una cadena de caracteres, se invoca *str()* para convertirlo en una cadena. Esto permite el uso de clases definidas por el usuario como mensajes, cuyo método __str__ puede retornar la cadena de formato real que se utilizará.

Distinto en la versión 3.2: La creación de *LogRecord* se ha hecho más configurable al proporcionar una fábrica que se utiliza para crear el registro. La fábrica se puede configurar usando *getLogRecordFactory()* y *setLogRecordFactory()* (ver esto para la firma de la fábrica).

Esta funcionalidad se puede utilizar para inyectar sus propios valores en *LogRecord* en el momento de la creación. Puede utilizar el siguiente patrón:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

Con este patrón, se podrían encadenar varias fábricas y, siempre que no sobrescriban los atributos de las demás o se sobrescriban involuntariamente los atributos estándar enumerados anteriormente, no debería haber sorpresas.

16.6.7 Atributos LogRecord

El LogRecord tiene una serie de atributos, la mayoría de los cuales se derivan de los parámetros del constructor. (Tenga en cuenta que los nombres no siempre se corresponden exactamente entre los parámetros del constructor de LogRecord y los atributos de LogRecord). Estos atributos pueden utilizarse para combinar los datos del registro en la cadena de formato. La siguiente tabla enumera (en orden alfabético) los nombres de los atributos, sus significados y el correspondiente marcador de posición en una cadena de formato de estilo con %.

Si utilizas formato-{} (*str.format()*), puedes usar {attrname} como marcador de posición en la cadena de caracteres de formato. Si está utilizando formato-$ (*string.Template*), use la forma ${attrname}. En ambos casos, por supuesto, reemplace‘ *attrname*‘ con el nombre de atributo real que desea utilizar.

En el caso del formato-{}, puede especificar *flags* de formato colocándolos después del nombre del atributo, separados con dos puntos. Por ejemplo: un marcador de posición de {msecs:03d} formateará un valor de milisegundos de 4 como 004. Consulte la documentación *str.format()* para obtener detalles completos sobre las opciones disponibles.

Nombre del atributo	Formato	Descripción
args	No debería necesitar formatear esto usted mismo.	La tupla de argumentos se fusionó en `msg` para producir un `messsage`, o un dict cuyos valores se utilizan para la fusión (cuando solo hay un argumento y es un diccionario).
asctime	`%(asctime)s`	Fecha y Hora en formato legible por humanos cuando se creó *LogRecord*. De forma predeterminada, tiene el formato "2003-07-08 16: 49: 45,896" (los números después de la coma son milisegundos).
created	`%(created)f`	Tiempo en que se creó *LogRecord* (según lo retornado por *time.time()*).
exc_info	No debería necesitar formatear esto usted mismo.	Tupla de excepción (al modo `sys.exc_info`) o, si no se ha producido ninguna excepción, `None`.
filename	`%(filename)s`	Parte del nombre de archivo de `pathname`.
funcName	`%(funcName)s`	Nombre de la función que contiene la llamada de logging.
levelname	`%(levelname)s`	Texto de nivel de logging para el mensaje (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`).
levelno	`%(levelno)s`	Número de nivel de logging para el mensaje (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`).
lineno	`%(lineno)d`	Número de línea original donde se emitió la llamada de logging (si está disponible).
message	`%(message)s`	El mensaje registrado, computado como `msg % args`. Esto se establece cuando se invoca *Formatter.format()*.
module	`%(module)s`	Módulo (parte del nombre de `filename`).
msecs	`%(msecs)d`	Porción de milisegundos del tiempo en que se creó *LogRecord*.
msg	No debería necesitar formatear esto usted mismo.	La cadena de caracteres de formato pasada en la llamada logging original. Se fusionó con `args` para producir un `message`, o un objeto arbitrario (ver arbitrary-object-messages).
name	`%(name)s`	Nombre del logger usado para registrar la llamada.
pathname	`%(pathname)s`	Nombre de ruta completo del archivo de origen donde se emitió la llamada de logging (si está disponible).
process	`%(process)d`	ID de proceso (si está disponible).
processName	`%(processName)s`	Nombre del proceso (si está disponible).
relativeCreated	`%(relativeCreated)d`	Tiempo en milisegundos cuando se creó el LogRecord, en relación con el tiempo en que se cargó el módulo logging.
stack_info	No debería necesitar formatear esto usted mismo.	Apila la información del marco (si está disponible) desde la parte inferior de la pila en el hilo actual hasta la llamada de registro que dio como resultado la generación de este registro.
thread	`%(thread)d`	ID de hilo (si está disponible).
threadName	`%(threadName)s`	Nombre del hilo (si está disponible).

Distinto en la versión 3.1: *processName* fue agregado.

16.6.8 Objetos LoggerAdapter

Las instancias *LoggerAdapter* se utilizan para pasar convenientemente información contextual en las llamadas de logging. Para ver un ejemplo de uso, consulte la sección sobre agregar información contextual a su salida de logging.

class logging.**LoggerAdapter**(*logger*, *extra*)

Retorna una instancia de *LoggerAdapter* inicializada con una instancia subyacente *Logger* y un objeto del tipo *dict*.

process(*msg*, *kwargs*)

Modifica el mensaje y/o los argumentos de palabra clave pasados a una llamada de logging para insertar información contextual. Esta implementación toma el objeto pasado como *extra* al constructor y lo agrega a *kwargs* usando la clave "extra". El valor de retorno es una tupla (*msg*, *kwargs*) que tiene las versiones (posiblemente modificadas) de los argumentos pasados.

Además de lo anterior, *LoggerAdapter* admite los siguientes métodos de *Logger*: *debug()*, *info()*, *warning()*, *error()*, *exception()*, *critical()*, *log()*, *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* y *hasHandlers()*. Estos métodos tienen las mismas firmas que sus contrapartes en *Logger*, por lo que puede usar los dos tipos de instancias indistintamente.

Distinto en la versión 3.2: Los métodos *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* y *hasHandlers()* se agregaron a *LoggerAdapter* . Estos métodos se delegan al logger subyacente.

Distinto en la versión 3.6: Se añadieron el atributo Attribute `manager` y el método `_log()`, que delegan al logger subyacente y permiten que los adaptadores se aniden.

16.6.9 Seguridad del hilo

El módulo logging está diseñado para ser seguro para subprocesos sin que sus clientes tengan que realizar ningún trabajo especial. Lo logra mediante el uso de bloqueos de hilos; hay un bloqueo para serializar el acceso a los datos compartidos del módulo, y cada gestor también crea un bloqueo para serializar el acceso a su E/S subyacente.

Si está implementando gestores de señales asíncronos usando el módulo *signal*, es posible que no pueda usar logging desde dichos gestores. Esto se debe a que las implementaciones de bloqueo en el módulo *threading* no siempre son reentrantes y, por lo tanto, no se pueden invocar desde dichos gestores de señales.

16.6.10 Funciones a nivel de módulo

Además de las clases descritas anteriormente, hay una serie de funciones a nivel de módulo.

logging.**getLogger**(*name=None*)

Retorna un logger con el nombre especificado o, si el nombre es `None`, retorna un logger que es el logger principal de la jerarquía. Si se especifica, el nombre suele ser un nombre jerárquico separado por puntos como *"a"*, *"a.b"* or *"a.b.c.d"*. La elección de estos nombres depende totalmente del desarrollador que utiliza logging.

Todas las llamadas a esta función con un nombre dado retornan la misma instancia de logger. Esto significa que las instancias del logger nunca necesitan pasar entre diferentes partes de una aplicación.

logging.**getLoggerClass**()

Retorna ya sea la clase estándar *Logger*, o la última clase pasada a *setLoggerClass()*. Esta función se puede llamar desde una nueva definición de clase, para garantizar que la instalación de una clase personalizada *Logger* no deshaga las *customizaciones* ya aplicadas por otro código. Por ejemplo:

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

logging.**getLogRecordFactory**()

Retorna un invocable que se usa para crear una *LogRecord*.

Nuevo en la versión 3.2: Esta función se ha proporcionado, junto con *setLogRecordFactory()*, para permitir a los desarrolladores un mayor control sobre cómo *LogRecord* representa un evento logging construido.

Consulte *setLogRecordFactory()* para obtener más información sobre cómo se llama a la fábrica.

logging.**debug**(*msg*, **args*, ***kwargs*)

Registra un mensaje con el nivel DEBUG en el logger raíz. *msg * es la cadena de carácteres de formato del mensaje y *args* son los argumentos que se fusionan en *msg* usando el operador de formato de cadena. (Tenga en cuenta que esto significa que puede utilizar palabras clave en la cadena de formato, junto con un único argumento de diccionario).

Hay tres argumentos de palabras clave en *kwargs* que se inspeccionan: *exc_info* que, si no se evalúa como falso, hace que se agregue información de excepción al mensaje de registro. Si se proporciona una tupla de excepción (en el formato retornado por *sys.exc_info()*) o una instancia de excepción, se utiliza; de lo contrario, se llama a *sys.exc_info()* para obtener la información de la excepción.

El segundo argumento opcional con la palabra clave *stack_info*, que por defecto es False. Si es verdadero, la información de la pila agregara el mensaje de registro, incluida la actual llamada del registro. Tenga en cuenta que esta no es la misma información de la pila que se muestra al especificar *exc_info*: la primera son los cuadros de la pila desde la parte inferior de la pila hasta la llamada de registro en el hilo actual, mientras que la segunda es la información sobre los cuadros de la pila que se han desenrollado, siguiendo una excepción, mientras busca gestores de excepción.

Puede especificar *stack_info* independientemente de *exc_info*, por ejemplo solo para mostrar cómo llegaste a cierto punto en tu código, incluso cuando no se lanzaron excepciones. Los marcos de la pila se imprimen siguiendo una línea de encabezado que dice:

```
Stack (most recent call last):
```

Esto imita el Traceback (most recent call last): que se usa cuando se muestran marcos de excepción.

El tercer argumento de palabra clave opcional es *extra* que se puede usar para pasar un diccionario que se usa para completar el __dict__ de LogRecord creado para el evento de registro con atributos definidos por el usuario. Estos atributos personalizados se pueden utilizar como desee. Por ejemplo, podrían incorporarse a mensajes registrados. Por ejemplo:

```
FORMAT = '%(asctime)s %(clientip)-15s %(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)
```

imprimiría algo como:

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

Las claves en el diccionario pasado *extra* no deben entrar en conflicto con las claves utilizadas por el sistema de registro. (Ver la documentación de *Formatter* para obtener más información sobre qué claves utiliza el sistema de registro).

Si opta por utilizar estos atributos en los mensajes registrados, debemos tener cuidado. En el caso anterior, por ejemplo, *Formatter* se ha configurado con una cadena de caracteres de formato que espera “clientip” y “user” en el diccionario de atributos de LogRecord. Si faltan, el mensaje no se registrará porque se producirá una excepción de formato de cadena. Entonces, en este caso, siempre debe pasar el diccionario *extra* con estas claves.

Si bien esto puede ser molesto, esta función está diseñada para su uso en circunstancias especializadas, como servidores de subprocesos múltiples donde el mismo código se ejecuta en muchos contextos, y las condiciones interesantes que surgen dependen de este contexto (como la dirección IP del cliente remoto y autenticado nombre de usuario, en el ejemplo anterior). En tales circunstancias, es probable que se especialice *Formatter*s con particular *Handler*s.

This function (as well as *info()*, *warning()*, *error()* and *critical()*) will call *basicConfig()* if the root logger doesn't have any handler attached.

Distinto en la versión 3.2: Se agregó el parámetro *stack_info*.

logging.**info**(*msg*, **args*, ***kwargs*)
Registra un mensaje con nivel INFO en el logger raíz. Los argumentos se interpretan como *debug()*.

logging.**warning**(*msg*, **args*, ***kwargs*)
Registra un mensaje con nivel WARNING en el logger raíz. Los argumentos se interpretan como *debug()*.

Nota: Hay una función obsoleta *warn'* que es funcionalmente idéntica a warning. Como warn está deprecado, por favor no lo use, use warning en su lugar.

---

logging.**error**(*msg*, **args*, ***kwargs*)
Registra un mensaje con nivel ERROR en el logger raíz. Los argumentos se interpretan como *debug()*.

logging.**critical**(*msg*, **args*, ***kwargs*)
Registra un mensaje con nivel CRITICAL en el logger raíz. Los argumentos se interpretan como *debug()*.

logging.**exception**(*msg*, **args*, ***kwargs*)
Registra un mensaje con nivel ERROR en el logger raíz. Los argumentos se interpretan como *debug()*. Se agrega información de excepción al mensaje de logging. Esta función solo se debe llamar desde un gestor de excepciones.

logging.**log**(*level*, *msg*, **args*, ***kwargs*)
Registra un mensaje con nivel *level* en el logger raíz. Los argumentos restantes se interpretan como *debug()*.

logging.**disable**(*level=CRITICAL*)
Proporciona un nivel superior de *level* para todos los loggers que tienen prioridad sobre el propio nivel del logger. Cuando surge la necesidad de reducir temporalmente la salida de logging en toda la aplicación, esta función puede resultar útil. Su efecto es deshabilitar todas las llamadas de gravedad *level* e inferior, de modo que si lo llaman con un valor de INFO, todos los eventos INFO y DEBUG se descartarán, mientras que los de gravedad WARNING y superiores se procesarán de acuerdo con el nivel efectivo del logger. Si se llama a logging.disable(logging.NOTSET) , elimina efectivamente este nivel primordial, de modo que la salida del registro depende nuevamente de los niveles efectivos de los loggers individuales.

Tenga en cuenta que si ha definido un nivel de logging personalizado superior a CRITICAL (esto no es recomendado), no podrá confiar en el valor predeterminado para el parámetro *level*, pero tendrá que proporcionar explícitamente un valor adecuado.

Distinto en la versión 3.7: El parámetro *level* se estableció por defecto en el nivel CRITICAL. Consulte el Issue #28524 para obtener más información sobre este cambio.

logging.**addLevelName**(*level*, *levelName*)
Asocia nivel *level* con el texto *levelName* en un diccionario interno, que se utiliza para asignar niveles numéricos a una representación textual, por ejemplo, cuando *Formatter* formatea un mensaje. Esta función también se puede utilizar para definir sus propios niveles. Las únicas restricciones son que todos los niveles utilizados deben registrarse utilizando esta función, los niveles deben ser números enteros positivos y deben aumentar en orden creciente de severidad.

Nota: Si está pensando en definir sus propios niveles, consulte la sección sobre custom-levels.

---

logging.**getLevelName**(*level*)
Retorna la representación textual o numérica del nivel de registro *level*.

Si *level* es uno de los niveles predefinidos CRITICAL, ERROR, WARNING, INFO o DEBUG entonces se obtiene la cadena correspondiente. Si has asociado niveles con nombres usando *addLevelName()* entonces se retorna el nombre que has asociado con *level*. Si se pasa un valor numérico correspondiente a uno de los niveles definidos, se retorna la representación de cadena correspondiente.

El parámetro *level* también acepta una representación de cadena del nivel como, por ejemplo, «INFO». En estos casos, esta función retorna el correspondiente valor numérico del nivel.

Si no se pasa un valor numérico o de cadena que coincida, se retorna la cadena de caracteres "Level %s" % nivel.

Nota: Los niveles internamente son números enteros (ya que deben compararse en la lógica de logging). Esta función se utiliza para convertir entre un nivel entero y el nombre del nivel que se muestra en la salida de logging formateado mediante el especificador de formato `%(levelname)s` (ver *Atributos LogRecord*).

Distinto en la versión 3.4: En las versiones de Python anteriores a la 3.4, esta función también podría pasar un nivel de texto y retornaría el valor numérico correspondiente del nivel. Este comportamiento indocumentado se consideró un error y se eliminó en Python 3.4, pero se restableció en 3.4.2 debido a que conserva la compatibilidad con versiones anteriores.

`logging.`**`makeLogRecord`**`(`*attrdict*`)`

Crea y retorna una nueva instancia `LogRecord` cuyos atributos están definidos por *attrdict*. Esta función es útil para tomar un diccionario de atributos serializado (*pickled*) `LogRecord`, enviado a través de un socket, y reconstituido como una instancia `LogRecord` en el extremo receptor.

`logging.`**`basicConfig`**`(`***kwargs*`)`

Realiza una configuración básica para el sistema de logging creando una `StreamHandler` con un `Formatter` predeterminado y agregándolo al logger raíz. Las funciones `debug()`, `info()`, `warning()`, `error()` y `critical()` llamarán `basicConfig()` automáticamente si no se definen gestores para el logger raíz.

Esta función no hace nada si el logger raíz ya tiene gestores configurados, a menos que el argumento de palabra clave *force* esté establecido como `True`.

Nota: Esta función debe llamarse desde el hilo principal antes de que se inicien otros hilos. En las versiones de Python anteriores a 2.7.1 y 3.2, si se llama a esta función desde varios subprocesos, es posible (en raras circunstancias) que se agregue un gestor al logger raíz más de una vez, lo que genera resultados inesperados como mensajes duplicados en el registro.

Se admiten los siguientes argumentos de palabras clave.

Formato	Descripción
filename	Specifies that a `FileHandler` be created, using the specified filename, rather than a `StreamHandler`.
filemode	Si se especifica *filename*, abre el archivo en *mode*. Por defecto es `'a'`.
format	Utiliza la cadena de caracteres de formato especificada para el gestor.Los atributos por defecto son `levelname`, `name` y `message` separado por dos puntos.
datefmt	Utiliza el formato de fecha/hora especificado, aceptado por `time.strftime()`.
style	Si *format* es especificado, utilice este estilo para la cadena de caracteres de formato. Uno de `'%'`, `'{'` o `'$'` para *printf-style*, `str.format()` o `string.Template` respectivamente. El valor predeterminado es `'%'`.
level	Establece el nivel del logger raíz en el *level* especificado.
stream	Utiliza la secuencia especificada para inicializar `StreamHandler`. Tenga en cuenta que este argumento es incompatible con *filename* - si ambos están presentes, se lanza un `ValueError`.
handlers	Si se especifica, debe ser una iteración de los gestores ya creados para agregar al logger raíz. A cualquier gestor que aún no tenga un formateador configurado se le asignará el formateador predeterminado creado en esta función. Tenga en cuenta que este argumento es incompatible con *filename* o *stream*; si ambos están presentes, se lanza un `ValueError`.
force	Si este argumento de palabra clave se especifica como verdadero, los gestores existentes adjuntos al logger raíz se eliminan y cierran antes de llevar a cabo la configuración tal como se especifica en los otros argumentos.
encoding	If this keyword argument is specified along with *filename*, its value is used when the `FileHandler` is created, and thus used when opening the output file.
errors	If this keyword argument is specified along with *filename*, its value is used when the `FileHandler` is created, and thus used when opening the output file. If not specified, the value “backslashreplace” is used. Note that if `None` is specified, it will be passed as such to `open()`, which means that it will be treated the same as passing “errors”.

Distinto en la versión 3.2: Se agregó el argumento *style*.

Distinto en la versión 3.3: Se agregó el argumento *handlers*. Se agregaron verificaciones adicionales para detectar situaciones en las que se especifican argumentos incompatibles (por ejemplo, *handlers* junto con *stream* o *filename*, o *stream* junto con *filename*).

Distinto en la versión 3.8: Se agregó el argumento *force*.

Distinto en la versión 3.9: Se han añadido los argumentos *encoding* y *errors*.

`logging.`**`shutdown`**`()`
Informa al sistema de logging para realizar un apagado ordenado descargando y cerrando todos los gestores. Esto se debe llamar al salir de la aplicación y no se debe hacer ningún uso posterior del sistema de logging después de esta llamada.

Cuando se importa el módulo de logging, registra esta función como un gestor de salida (ver *atexit*), por lo que normalmente no es necesario hacerlo manualmente.

`logging.`**`setLoggerClass`**`(`*klass*`)`
Le dice al sistema de logging que use la clase *klass* al crear una instancia de un logger. La clase debe definir `__init__()` tal que solo se requiera un argumento de nombre, y `__init__()` debe llamar `Logger.__init__()`. Por lo general, esta función se llama antes de cualquier loggers sea instanciado por las aplicaciones que necesitan utilizar un comportamiento de logger personalizado. Después de esta llamada, como en cualquier otro momento, no cree instancias de loggers directamente usando la subclase: continúe usando la API *logging.getLogger()* para obtener sus loggers.

`logging.`**`setLogRecordFactory`**`(`*factory*`)`
Establece un invocable que se utiliza para crear *LogRecord*.

Parámetros **`factory`** – La fábrica invocable que se utilizará para crear una instancia de un registro.

Nuevo en la versión 3.2: Esta función se ha proporcionado, junto con *getLogRecordFactory()*, para permitir a los desarrolladores un mayor control sobre cómo se construye *LogRecord* que representa un evento de logging.

La fábrica tiene la siguiente firma:

```
factory(name, level, fn, lno, msg, args, exc_info, func=None,
sinfo=None, **kwargs)
```

name El nombre del logger.

level El nivel de logging (numérico).

fn El nombre de ruta completo del archivo donde se realizó la llamada de logging.

lno El número de línea en el archivo donde se realizó la llamada de logging.

msg El mensaje de logging.

args Los argumentos para el mensaje de logging.

exc_info Una tupla de excepción o `None`.

func El nombre de la función o método que invocó la llamada de logging.

sinfo Un seguimiento de pila como el que proporciona *traceback.print_stack()*, que muestra la jerarquía de llamadas.

kwargs Argumentos de palabras clave adicionales.

16.6.11 Atributos a nivel de módulo

`logging.`**`lastResort`**
Un «gestor de último recurso» está disponible a través de este atributo. Esta es una *`StreamHandler`* que escribe en"sys.stderr" con un nivel `WARNING`, y se usa para gestionar eventos de logging en ausencia de cualquier configuración de logging. El resultado final es simplemente imprimir el mensaje en `sys.stderr`. Esto reemplaza el mensaje de error anterior que decía que «no se pudieron encontrar gestores para el logger XYZ». Si necesita el comportamiento anterior por alguna razón, `lastResort` se puede configurar en `None`.

Nuevo en la versión 3.2.

16.6.12 Integración con el módulo de advertencias

La función *`captureWarnings()`* se puede utilizar para integrar *`logging`* con el módulo *`warnings`*.

`logging.`**`captureWarnings`**(*capture*)
Esta función se utiliza para activar y desactivar la captura de advertencias (*warnings*).

Si *capture* es `True`, las advertencias emitidas por el módulo *`warnings`* serán redirigidas al sistema de logging. Específicamente, una advertencia se formateará usando *`warnings.formatwarning()`* y la cadena de caracteres resultante se registrará en un logger llamado `'py.warnings'` con severidad `WARNING`.

Si *capture* es `False`, la redirección de advertencias al sistema de logging se detendrá y las advertencias serán redirigidas a sus destinos originales (es decir, aquellos en vigor antes de que se llamara a `captureWarnings(True)`).

Ver también:

Módulo *`logging.config`* API de configuración para el módulo logging.

Módulo *`logging.handlers`* Gestores útiles incluidos con el módulo logging.

PEP 282 - A Logging System La propuesta que describió esta característica para su inclusión en la biblioteca estándar de Python.

Paquete logging original de Python Esta es la fuente original del paquete *`logging`*. La versión del paquete disponible en este sitio es adecuada para usar con Python 1.5.2, 2.1.xy 2.2.x, que no incluyen el paquete *`logging`* en la biblioteca estándar.

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 gestores que se declaran en *logging* o *logging.handlers*.

logging.config.**dictConfig**(*config*)

Toma la configuración de registro de un diccionario. Los contenidos de este diccionario se describen 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 lanzarán un error:

- Un `level` que no es una cadena o que es una cadena que no corresponde a un nivel de registro real.
- Un valor de `propagate` que no es booleano.
- Una identificación que no tiene un destino correspondiente.
- Una identificación de gestor 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*, *encoding=None*)

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 especifican como `False`, los registradores que existen cuando se hace esta llamada se deshabilitan. El valor por defecto es `True` porque esto habilita comportamientos antiguos de una manera compatible con versiones

anteriores. Este comportamiento consiste en deshabilitar cualquier registrador que no sea raíz a menos que ellos o sus antecesores sean explícitamente nombrados en la configuración del registro. :param encoding: La codificación que se usa para abrir archivos cuando *fname* es nombre del archivo.

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`.

Nuevo en la versión 3.10: El parámetro *codificación* se añade.

`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 `verify`, 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 `verify` invocable pueda realizar la verificación o descifrado de la firma. El llamado invocable `verify` 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 `verify` para *listen()* para escuchar y evitar que se apliquen configuraciones no reconocidas.

Distinto en la versión 3.4: Se agregó el argumento `verify`.

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 Security considerations

The logging configuration functionality tries to offer convenience, and in part this is done by offering the ability to convert text in configuration files into Python objects used in logging configuration - for example, as described in *Objetos definidos por el usuario*. However, these same mechanisms (importing callables from user-defined modules and calling them with parameters from the configuration) could be used to invoke any code you like, and for this reason you should treat configuration files from untrusted sources with *extreme caution* and satisfy yourself that nothing bad can happen if you load them, before actually loading them.

16.7.3 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 gestor llamado "consola" y luego decir que el registrador llamado "inicio" enviará sus mensajes al gestor "consola". Estos objetos no se limitan a los proporcionados por el módulo `logging` porque podría escribir su propia clase de formateador o gestor. 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`.

  Se busca el diccionario de configuración por las siguientes claves opcionales que corresponden a los argumentos pasados para crear un objeto `Formatter`:

  - `format`
  - `datefmt`
  - `style`
  - `validate` (desde la versión >=3.8)

  Una clave opcional `class` indica el nombre de la de clase del formateador (como un módulo punteado y nombre de clase). Por su parte los argumentos de instanciación son `Formatter`, de este modo esta clave es más útil para instanciar una subclase personalizada de `Formatter`. Por ejemplo, la clase alternativa presentaría excepciones rastreadas en un formato amplio o resumido. Si tu formateador necesita claves de configuración diferentes o extra debes usar *Objetos definidos por el usuario*.

- *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 `name` (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 gestor y cada valor es un diccionario que describe cómo configurar la instancia del gestor correspondiente.

  El diccionario de configuración busca las siguientes claves:

  - `clase` (obligatorio). Este es el nombre completo de la clase de gestor.

  - `level` (opcional). El nivel del gestor.
  - `formatter` (opcional). La identificación del formateador para este gestor.
  - `filters` (opcional). Una lista de identificadores de los filtros para este gestor.

  Todas las claves *other* se pasan como argumentos de palabras clave al constructor del gestor. 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 gestor con id `console` se instancia como *`logging.StreamHandler`*, usando `sys.stdout` como la secuencia subyacente. El gestor con la identificación `file` 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:

  - `level` (opcional). El nivel del registrador.
  - `propagate` (opcional). La configuración de propagación del registrador.
  - `filters` (opcional). Una lista de identificadores de los filtros para este registrador.
  - `handlers` (opcional). Una lista de identificadores de los gestores para este registrador.

  Los registradores especificados se configurarán de acuerdo con el nivel, la propagación, los filtros y los gestores especificados.

- *root* - Esta será la configuración para el registrador raíz. El procesamiento de la configuración será como para cualquier registrador, excepto que la configuración de `propagate` 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 que no sea raíz 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, gestores, filtros, formateadores en tiempo de ejecución, una vez que se configura una configuración; la verbosidad de los registradores y gestores 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 `formatters` y `filters`, y procesará solo el `level` configuraciones en las entradas de `handlers`, y las configuraciones de `level` y `propagate` en las entradas de `loggers` y `root`.

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, gestores, 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 gestor particular. A los fines de esta discusión, podemos decir que el registrador representa la fuente y el gestor 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 gestor. 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 gestores adjuntos, que se describen mediante los identificadores de gestor `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 gestores, 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 `custom`, 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 `custom` será retornado por la llamada:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

Advertencia: The values for keys such as `bar`, `spam` and `answer` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but passed to the callable as-is.

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.

You can also specify a special key `'.'` whose value is a dictionary is a mapping of attribute names to values. If found, the specified attributes will be set on the user-defined object before it is returned. Thus, with the following configuration:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42,
  '.' {
    'foo': 'bar',
    'baz': 'bozz'
  }
}
```

the returned formatter will have attribute `foo` set to `'bar'` and attribute `baz` set to `'bozz'`.

Advertencia: The values for attributes such as `foo` and `baz` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but set as attribute values as-is.

Handler configuration order

Handlers are configured in alphabetical order of their keys, and a configured handler replaces the configuration dictionary in (a working copy of) the `handlers` dictionary in the schema. If you use a construct such as `cfg://handlers.foo`, then initially `handlers['foo']` points to the configuration dictionary for the handler named `foo`, and later (once that handler has been configured) it points to the configured handler instance. Thus, `cfg://handlers.foo` could resolve to either a dictionary or a handler instance. In general, it is wise to name handlers in a way such that dependent handlers are configured _after_ any handlers they depend on; that allows something like `cfg://handlers.foo` to be used in configuring a handler that depends on handler `foo`. If that dependent handler were named `bar`, problems would result, because the configuration of `bar` would be attempted before that of `foo`, and `foo` would not yet have been configured. However, if the dependent handler were named `foobar`, it would be configured after `foo`, with the result that `cfg://handlers.foo` would resolve to configured handler `foo`, and not its configuration dictionary.

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 `level` en un registrador o gestor se convertirá automáticamente al valor `logging.DEBUG`, y las entradas `handlers`, `filters` y `formatter` tomarán una identificación de objeto y se resuelven 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 gestor para delegar. Dado que el sistema ya conoce esta clase, entonces en la configuración, el `target` dado solo necesita ser la identificación del objeto del gestor de destino relevante, y el sistema resolverá el gestor desde la identificación. Sin embargo, si un usuario define un `my.package.MyHandler` que tiene un gestor `alternate`, el sistema de configuración no sabría que el `alternate` se refería a un gestor. 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'` resolvería al diccionario con clave `handlers`, la cadena `'cfg://handlers.email` resolvería al diccionario con clave `email` en el diccionario `handlers`, y así sucesivamente. La cadena `'cfg://handlers.email.toaddrs[1]` resolvería a `'dev_team.domain.tld'` y la cadena `'cfg://handlers.email.toaddrs[0]` resolvería el valor

'support_team@domain.tld'. Se puede acceder al valor de subject 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*.

16.7.4 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 [formatters] 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 gestor llamado hand01 en la sección [handlers] tendrá su configuración en una sección llamada [handler_hand01], mientras que un formateador llamado form01 en el [formatters] 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

El registrador raíz debe especificar un nivel y una lista de gestores. A continuación se muestra un ejemplo de una sección de registrador raíz.

```
[logger_root]
level=NOTSET
handlers=hand01
```

La entrada `level` 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 `handlers` es una lista separada por comas de nombres de gestores, que debe aparecer en la sección `[handlers]`. Estos nombres deben aparecer en la sección `[handlers]` 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 `level` y `handlers` se interpretan como para el registrador raíz, excepto que si el nivel de un registrador que no sea 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 `propagate` se establece en 1 para indicar que los mensajes deben propagarse a los gestores que están más arriba en la jerarquía del registrador, o 0 para indicar que los mensajes **no** se propagan a los gestores 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 gestor se ejemplifican a continuación.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

La entrada `class` indica la clase del gestor (según lo determinado por *eval()* en el espacio de nombres del paquete `logging`). El `level` se interpreta como para los registradores, y `NOTSET` se entiende como "registrar todo".

La entrada `formatter` indica el nombre clave del formateador para este gestor. Si está en blanco, se utiliza un formateador predeterminado (`logging._defaultFormatter`). Si se especifica un nombre, debe aparecer en la sección `[formatters]` y tener una sección correspondiente en el archivo de configuración.

La entrada `args`, cuando *eval()* ua en el contexto del espacio de nombres del paquete `logging`, es la lista de argumentos para el constructor de la clase de gestor. Consulte los constructores de los gestores 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()*úa en el contexto del espacio de nombres del paquete `logging`, es el argumento de palabra clave diccionario al constructor para la clase de gestor. Si no se proporciona, el valor predeterminado es `{}`.

```
[handler_hand02]
class=FileHandler
level=DEBUG
```

(continué en la próxima página)

(proviene de la página anterior)

```
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)

[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=
style=%
validate=True
class=logging.Formatter
```

Los argumentos para la configuración del formateador son los mismos que las claves en el esquema del diccionario *formatters section*.

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* Gestores 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 se especifica un formateador, se utiliza para formatear el registro. Luego, el registro se escribe en el flujo seguido de *terminator*. Si hay información de excepción, se formatea con *traceback.print_exception()* y se agrega al flujo.

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.

terminator
Cadena utilizada como terminador al escribir un registro formateado en un flujo. El valor por defecto es `'\n'`.

Si no quieres una terminación de nueva línea, puedes establecer el atributo `terminator` de la instancia del manejador a la cadena vacía.

En versiones anteriores, el terminador estaba codificado como `'\n'`.

Nuevo en la versión 3.2.

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*, *errors=None*)
Retorna una nueva instancia de la clase *FileHandler*. El archivo especificado se abre y se utiliza como flujo para el registro. Si no se especifica *mode*, se utiliza `'a'`. Si *encoding* no es `None`, se utiliza para abrir el archivo con esa codificación. Si *delay* es verdadero, la apertura del archivo se aplaza hasta la primera llamada a *emit()*. De forma predeterminada, el archivo crece indefinidamente. Si se especifica *errors*, se usa para determinar cómo se manejan los errores de codificación.

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*».

Distinto en la versión 3.9: Se agregó el parámetro *errors*.

close()
Cierra el archivo.

emit(*record*)
Da la salida del registro al archivo.

Note that if the file was closed due to logging shutdown at exit and the file mode is "w", the record will not be emitted (see bpo-42378).

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*, *errors=None*)

Retorna una nueva instancia de la clase *WatchedFileHandler*. El archivo especificado se abre y se utiliza como flujo para el registro. Si no se especifica *mode*, se utiliza 'a'. Si *encoding* no es None, se utiliza para abrir el archivo con esa codificación. Si *delay* es verdadero, la apertura del archivo se aplaza hasta la primera llamada a *emit()*. De forma predeterminada, el archivo crece indefinidamente. Si se proporciona *errors*, determina cómo se manejan los errores de codificación.

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*».

Distinto en la versión 3.9: Se agregó el parámetro *errors*.

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*, *errors=None*)

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.

It's also worth noting that care should be taken when using a namer to preserve certain attributes in the filename which are used during rotation. For example, *RotatingFileHandler* expects to have a set of log files whose names contain successive integers, so that rotation works as expected, and *TimedRotatingFileHandler* deletes old log files (based on the backupCount parameter passed to the handler's initializer) by determining the oldest files to delete. For this to happen, the filenames should be sortable using the date/time portion of the filename, and a namer needs to respect

this. (If a namer is wanted that doesn't respect this scheme, it will need to be used in a subclass of `TimedRotatingFileHandler` which overrides the `getFilesToDelete()` method to fit in with the custom naming scheme.)

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*, *errors=None*)
Retorna una nueva instancia de la clase `RotatingFileHandler`. El archivo especificado se abre y se utiliza como flujo para el registro. Si no se especifica *mode*, se utiliza `'a'`. Si *encoding* no es `None`, se utiliza para abrir el archivo con esa codificación. Si *delay* es verdadero, la apertura del archivo se aplaza hasta la primera llamada a `emit()`. De forma predeterminada, el archivo crece indefinidamente. Si se proporciona *errors*, determina cómo se manejan los errores de codificación.

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 t `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*».

Distinto en la versión 3.9: Se agregó el parámetro *errors*.

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 descripto 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*, *errors=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 *atTime* 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 *rollover*
`'midnight'`	Volcado (*rollover*) a medianoche , si *atTime* no está especificado, sino el volcado se hará *atTime*	Usado para calcular la hora inicial del volcado *rollover*

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.

Si se especifica *errors*, se utiliza para determinar cómo se manejan los errores de codificación.

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*».

Distinto en la versión 3.9: Se agregó el parámetro *errors*.

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.

getFilesToDelete()
Returns a list of filenames which should be deleted as part of rollover. These are the absolute paths of the oldest backup log files written by the handler.

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 alternativamente 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*.

Nota: As UDP is not a streaming protocol, there is no persistent connection between an instance of this handler and *host*. For this reason, when using a network socket, a DNS lookup might have to be made each time an event is logged, which can introduce some latency into the system. If this affects you, you can do a lookup yourself and initialize this handler using the looked-up IP address rather than the hostname.

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.

Nota: On macOS 12.x (Monterey), Apple has changed the behaviour of their syslog daemon - it no longer listens on a domain socket. Therefore, you cannot expect `SysLogHandler` to work on this system.

See gh-91070 for more information.

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` o `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='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 usara `'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 especifica. 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*, ubicada en el módulo *logging.handlers*, admite el envío de mensajes de registro a un servidor web, utilizando 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 especifico. 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 codificado en URL. El método *mapLogRecord()* se utiliza para convertir el registro al diccionario que se enviará.

Nota: Dado que preparar un registro para enviarlo a un servidor web no es lo mismo que una operación de formateo genérico, el uso de *setFormatter()* para especificar un *Formatter* para un *HTTPHandler* no tiene ningún efecto. En lugar de llamar a *format()*, este controlador llama a *mapLogRecord()* y luego a *urllib.parse.urlencode()* para codificar el diccionario en una forma 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*, *QueueHandler* se puede usar para permitir que los controladores hagan su trabajo en un hilo separado del que realiza el registro. Esto es importante en aplicaciones web y también en otras aplicaciones de servicio donde los subprocesos que atienden a los clientes deben responder lo más rápido posible, mientras que cualquier operación potencialmente lenta (como enviar un correo electrónico a través de *SMTPHandler*) se realiza en un subproceso separado.

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*.

Nota: If you are using *multiprocessing*, you should avoid using *SimpleQueue* and instead use *multiprocessing.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.

The base implementation formats the record to merge the message, arguments, exception and stack information, if present. It also removes unpickleable items from the record in-place. Specifically, it overwrites the record's `msg` and `message` attributes with the merged message (obtained by calling the handler's *format()* method), and sets the `args`, `exc_info` and `exc_text` attributes to `None`.

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.

Nota: The base implementation formats the message with arguments, sets the `message` and `msg` attributes to the formatted message and sets the `args` and `exc_text` attributes to `None` to allow pickling and to prevent further attempts at formatting. This means that a handler on the *QueueListener* side won't have the information to do custom formatting, e.g. of exceptions. You may wish to subclass `QueueHandler` and override this method to e.g. avoid setting `exc_text` to `None`. Note that the `message` / `msg` / `args` changes are related to ensuring the record is pickleable, and you might or might not be able to avoid doing that depending on whether your `args` are pickleable. (Note that you may have to consider not only your own code but also code in any libraries that you use.)

---

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.

listener
When created via configuration using *dictConfig()*, this attribute will contain a *QueueListener* instance for use with this handler. Otherwise, it will be `None`.

Nuevo en la versión 3.12.

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*, *QueueListener* se puede usar para permitir que los controladores hagan su trabajo en un hilo separado del que realiza el registro. Esto es importante en aplicaciones web y también en otras aplicaciones de servicio donde los subprocesos que atienden a los clientes deben responder lo más rápido posible, mientras que cualquier operación potencialmente lenta (como enviar un correo electrónico a través de *SMTPHandler*) se realiza en un subproceso separado.

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*.

Nota: If you are using *multiprocessing*, you should avoid using *SimpleQueue* and instead use *multiprocessing.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

Source code: Lib/curses

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 locate.setlocate() 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*)
Retorna la intensidad de los componentes rojo, verde y azul (RGB) en el color *color_number*, que debe estar entre 0 y COLORS-1. Retorna una tupla de 3, que contiene los valores R,G,B para el color dado, que estará entre 0 (sin componente) y 1000 (cantidad máxima de componente).

curses.**color_pair**(*pair_number*)
Retorna el valor del atributo para mostrar texto en el par de colores especificado. Solo se admiten los primeros 256 pares de colores. Este valor de atributo se puede combinar con A_STANDOUT, A_REVERSE, y los otros atributos A_*. *pair_number()* es la contraparte de esta función.

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**()
Después de *getch()* retorna KEY_MOUSE para señalar un evento de mouse, se debe llamar a este método para recuperar el evento de mouse en cola, representado como una tupla de 5 (id, x, y, z, bstate). *id* es un valor de ID que se utiliza para distinguir varios dispositivos, y *x*, *y*, *z* son las coordenadas del evento. (*z* no se usa actualmente.) *bstate* es un valor entero cuyos bits se establecerán para indicar el tipo de evento, y será el OR bit a bit de una o más de las siguientes constantes, donde *n* es el botón número de 1 a 5: BUTTONn_PRESSED, BUTTONn_RELEASED, BUTTONn_CLICKED, BUTTONn_DOUBLE_CLICKED, BUTTONn_TRIPLE_CLICKED, BUTTON_SHIFT, BUTTON_CTRL, BUTTON_ALT.

Distinto en la versión 3.10: Las constantes BUTTON5_* ahora están expuestas si son proporcionadas por la biblioteca curses subyacente.

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_extended_color_support**()
Retorna True si el módulo admite soporte de color extendido; de lo contrario retorna False. El soporte de color extendido permite más de 256 pares de colores para terminales que admiten más de 16 colores (por ejemplo, xterm-256color).

El soporte de color extendido requiere ncurses versión 6.1 o posterior.

Nuevo en la versión 3.10.

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 lanza 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*)
Cambia la definición de un par de colores. Se necesitan tres argumentos: el número del par de colores que se va a cambiar, el número de color de primer plano y el número de color de fondo. El valor de *par_number* debe estar entre `1` y `COLOR_PAIRS-1` (el par de colores `0` está conectado a blanco sobre negro y no se puede cambiar). El valor de los argumentos *fg* y *bg* debe estar entre `0'' y ``COLORS-1` o, después de llamar a *use_default_colors()*, `-1`. Si el par de colores se inicializó previamente, la pantalla se actualiza y todas las apariciones de ese par de colores se cambian a la nueva definición.

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)`. *availmask* 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*)
Retorna una tupla `(fg, bg)` conteniendo los colores del par de color solicitado. El valor de *pair_number* debe ser entre `0` y `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.**get_escdelay**()
Recupera el valor establecido por *set_escdelay()*.

Nuevo en la versión 3.9.

curses.**set_escdelay**(*ms*)
Establece el número de milisegundos de espera después de leer un carácter de escape, para distinguir entre un carácter de escape individual ingresado en el teclado de las secuencias de escape enviadas por el cursor y las teclas de función.

Nuevo en la versión 3.9.

curses.**get_tabsize**()
Recupera el valor establecido por *set_tabsize()*.

Nuevo en la versión 3.9.

curses.**set_tabsize**(*size*)
Establece el número de columnas utilizadas por la biblioteca de curses al convertir un carácter de tabulación en espacios, ya que agrega la tabulación a una ventana.

Nuevo en la versión 3.9.

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, dónde *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 con-

trol 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*, */*, **args*, ***kwargs*)
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 un 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*])
Pinta el carácter *ch* en `(y, x)` con atributos *attr*, sobrescribiendo cualquier carácter pintado previamente en esa ubicación. De forma predeterminada, la posición y los atributos del carácter son la configuración actual del objeto ventana.

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 redimensionan 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
ls	Lado izquierdo	`ACS_VLINE`
rs	Lado derecho	`ACS_VLINE`
ts	Arriba	`ACS_HLINE`
bs	Abajo	`ACS_HLINE`
tl	Esquina superior izquierda	`ACS_ULCORNER`
tr	Esquina superior derecha	`ACS_URCORNER`
bl	Esquina inferior izquierda	`ACS_LLCORNER`
br	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.`**`clrtobot`**`()`
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 hacía 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.

Distinto en la versión 3.10: Anteriormente retornaba `1` o `0` en lugar de“True“ o `False`.

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 hacía 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.**scrollok**(*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 hacía 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 cero, 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, específica 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*[, *attr*])
window.**vline**(*y*, *x*, *ch*, *n*[, *attr*])
Display a vertical line starting at `(y, x)` with length *n* consisting of the character *ch* with attributes *attr*.

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
A_ALTCHARSET	Modo de conjunto de caracteres alternativo
A_BLINK	Modo parpadeo
A_BOLD	Modo negrita
A_DIM	Modo tenue
A_INVIS	Modo invisible o en blanco
A_ITALIC	Modo cursiva
A_NORMAL	Atributo normal
A_PROTECT	Modo protegido
A_REVERSE	Fondo inverso y colores de primer plano
A_STANDOUT	Modo destacado
A_UNDERLINE	Modo subrayado
A_HORIZONTAL	Resaltado horizontal
A_LEFT	Resaltado a la izquierda
A_LOW	Resaltado bajo
A_RIGHT	Resaltado a la derecha
A_TOP	Resaltado arriba
A_VERTICAL	Resaltado vertical
A_CHARTEXT	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
A_ATTRIBUTES	Mascara de bits para extraer atributos
A_CHARTEXT	Mascara de bits para extraer un caracter
A_COLOR	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
KEY_MIN	Valor mínimo de la clave

continué en la próxima página

Tabla 1 – proviene de la página anterior

Clave constante	Clave
KEY_BREAK	Clave rota (no confiable)
KEY_DOWN	Flecha hacia abajo
KEY_UP	Flecha hacia arriba
KEY_LEFT	Flecha hacia la izquierda
KEY_RIGHT	Flecha hacia la derecha
KEY_HOME	Tecla de inicio (flecha hacia arriba + flecha hacia la izquierda)
KEY_BACKSPACE	Retroceso (no confiable)
KEY_F0	Teclas de función. Más de 64 teclas de función son soportadas.
KEY_Fn	Valor de tecla de función *n*
KEY_DL	Borrar línea
KEY_IL	Inserte línea
KEY_DC	Borrar caracter
KEY_IC	Insertar carácter o ingresara al modo de inserción
KEY_EIC	Salir del modo de inserción de caracteres
KEY_CLEAR	Limpiar pantalla
KEY_EOS	Limpiar al final de pantalla
KEY_EOL	Limpiar al final de la línea
KEY_SF	Desplazar una línea hacia adelante
KEY_SR	Desplazar una línea hacia atrás (reversa)
KEY_NPAGE	Página siguiente
KEY_PPAGE	Página anterior
KEY_STAB	Establecer pestaña
KEY_CTAB	Limpiar pestaña
KEY_CATAB	Limpiar todas las pestañas
KEY_ENTER	Ingresar o enviar (no confiable)
KEY_SRESET	Reinicio suave (parcial) (no confiable)
KEY_RESET	Reinicio o reinicio fuerte (no confiable)
KEY_PRINT	Imprimir
KEY_LL	Inicio abajo o abajo (abajo a la izquierda)
KEY_A1	Superior izquierda del teclado
KEY_A3	Superior derecha de el teclado
KEY_B2	Centro del teclado
KEY_C1	Inferior izquierdo del teclado
KEY_C3	Inferior derecho del teclado
KEY_BTAB	Pestaña trasera
KEY_BEG	Mendigar (Comienzo)
KEY_CANCEL	Cancelar
KEY_CLOSE	Cerrar
KEY_COMMAND	Cmd (Comando)
KEY_COPY	Copiar
KEY_CREATE	Crear
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

continué en la próxima página

Tabla 1 – proviene de la página anterior

Clave constante	Clave
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, la siguientes funciones de teclado son estándar:

Tecla	Constante
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

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
ACS_BBSS	nombre alternativo para la esquina superior derecha
ACS_BLOCK	bloque cuadrado sólido
ACS_BOARD	tablero de cuadrados
ACS_BSBS	nombre alternativo para línea horizontal
ACS_BSSB	nombre alternativo para esquina superior izquierda
ACS_BSSS	nombre alternativo para el soporte superior
ACS_BTEE	soporte inferior
ACS_BULLET	bala
ACS_CKBOARD	tablero inspector (punteado)
ACS_DARROW	flecha punteada hacia abajo
ACS_DEGREE	símbolo de grado
ACS_DIAMOND	diamante
ACS_GEQUAL	mayor que o igual a
ACS_HLINE	línea horizontal
ACS_LANTERN	símbolo de la linterna
ACS_LARROW	flecha izquierda
ACS_LEQUAL	menor que o igual a
ACS_LLCORNER	esquina inferior izquierda
ACS_LRCORNER	esquina inferior derecha
ACS_LTEE	soporte izquierdo
ACS_NEQUAL	signo no igual
ACS_PI	letra pi
ACS_PLMINUS	signo más o menos
ACS_PLUS	gran signo más
ACS_RARROW	flecha hacia la derecha
ACS_RTEE	soporte derecho
ACS_S1	escanear línea 1
ACS_S3	escanear línea 3
ACS_S7	escanear línea 7
ACS_S9	escanear línea 9
ACS_SBBS	nombre alterno para la esquina inferior derecha
ACS_SBSB	nombre alterno para la línea vertical
ACS_SBSS	nombre alterno para el soporte derecho
ACS_SSBB	nombre alterno para la esquina inferior izquierda
ACS_SSBS	nombre alterno para el soporte inferior
ACS_SSSB	nombre alterno para el soporte izquierdo
ACS_SSSS	nombre alterno para *crossover* 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*, BBedit 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

Source code: Lib/curses/ascii.py

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

continué en la próxima página

Tabla 3 – proviene de la página anterior

Nombre	Significado
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 (*Carriage return* en inglés)
SO	*Shift-out*, comenzar un conjunto de caracteres alternativo
SI	*Shift-in*, 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 macOS (y quizás en otras plataformas), los archivos ejecutables pueden ser archivos universales que contienen múltiples 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**()
Returns the machine type, e.g. `'AMD64'`. An empty string is returned if the value cannot be determined.

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. `'amdk6'`.

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**()
Returns the system's release, e.g. `'2.2.0'` or `'NT'`. An empty string is returned if the value cannot be determined.

platform.**system**()
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: Result changed from a tuple to 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=''*)
Get additional version information from the Windows Registry and return a tuple (release, version, csd, ptype) referring to OS release, version number, CSD level (service pack) and OS type (multi/single processor). Values which cannot be determined are set to the defaults given as parameters (which all default to an empty string).

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**()
Returns a string representing the current Windows edition, or None if the value cannot be determined. Possible values include but are not limited to 'Enterprise', 'IoTUAP', 'ServerStandard', and '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 macOS

platform.**mac_ver**(*release=''*, *versioninfo=('', '', '')*, *machine=''*)
Obtenga información de la versión de macOS y devuélvala como tupla (release, versioninfo, machine) con *versioninfo* como tupla (version, dev_stage, non_release_version).

Cualquier registro que no puede ser determinado se establece como ''. Todas 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*.

16.14.6 Plataformas Linux

platform.**freedesktop_os_release**()
Obtiene la identificación del sistema operativo del archivo `os-release` y la retorna como dict. El archivo `os-release` es un freedesktop.org standard y está disponible en la mayoría de las distribuciones de Linux. Una excepción notable son las distribuciones de Android y basadas en Android.

Lanza *OSError* o subclase cuando no se pueden leer ni `/etc/os-release` ni `/usr/lib/os-release`.

En caso de éxito, la función retorna un diccionario donde las claves y los valores son cadenas de caracteres. Los valores tienen sus caracteres especiales como `"` y `$` sin comillas. Los campos `NAME`, `ID` y `PRETTY_NAME` siempre se definen de acuerdo con el estándar. Todos los demás campos son opcionales. Los proveedores pueden incluir campos adicionales.

Tenga en cuenta que campos como `NAME`, `VERSION` y `VARIANT` son cadenas de caracteres adecuadas para la presentación a los usuarios. Los programas deben usar campos como `ID`, `ID_LIKE`, `VERSION_ID` o `VARIANT_ID` para identificar distribuciones de Linux.

Ejemplo:

```
def get_like_distro():
    info = platform.freedesktop_os_release()
    ids = [info["ID"]]
    if "ID_LIKE" in info:
        # ids are space separated and ordered by precedence
        ids.extend(info["ID_LIKE"].split())
    return ids
```

Nuevo en la versión 3.10.

16.15 errno — Símbolos estándar del sistema errno

This module makes available standard `errno` system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from `linux/include/errno.h`, which should be all-inclusive.

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**
Operation not permitted. This error is mapped to the exception *PermissionError*.

errno.**ENOENT**
No such file or directory. This error is mapped to the exception *FileNotFoundError*.

errno.**ESRCH**
No such process. This error is mapped to the exception *ProcessLookupError*.

errno.**EINTR**
Interrupted system call. This error is mapped to the exception *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**
No child processes. This error is mapped to the exception *ChildProcessError*.

errno.**EAGAIN**
Try again. This error is mapped to the exception *BlockingIOError*.

errno.**ENOMEM**
Sin memoria

errno.**EACCES**
Permission denied. This error is mapped to the exception *PermissionError*.

errno.**EFAULT**
Dirección incorrecta

errno.**ENOTBLK**
Bloquear dispositivo requerido

errno.**EBUSY**
Dispositivo o recurso ocupado

errno.**EEXIST**
File exists. This error is mapped to the exception *FileExistsError*.

errno.**EXDEV**
Enlace entre dispositivos

errno.**ENODEV**
Hay tal dispositivo

errno.**ENOTDIR**
Not a directory. This error is mapped to the exception *NotADirectoryError*.

errno.**EISDIR**
Is a directory. This error is mapped to the exception *IsADirectoryError*.

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**
Broken pipe. This error is mapped to the exception *BrokenPipeError*.

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**
Operation would block. This error is mapped to the exception *BlockingIOError*.

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**
Software caused connection abort. This error is mapped to the exception *ConnectionAbortedError*.

errno.**ECONNRESET**
Connection reset by peer. This error is mapped to the exception *ConnectionResetError*.

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**
Cannot send after transport endpoint shutdown. This error is mapped to the exception *BrokenPipeError*.

errno.**ETOOMANYREFS**
Demasiadas referencias: no se puede empalmar

errno.**ETIMEDOUT**
Connection timed out. This error is mapped to the exception *TimeoutError*.

errno.**ECONNREFUSED**
Connection refused. This error is mapped to the exception *ConnectionRefusedError*.

errno.**EHOSTDOWN**
Anfitrión caído

errno.**EHOSTUNREACH**
Sin ruta al anfitrión

errno.**EALREADY**
Operation already in progress. This error is mapped to the exception *BlockingIOError*.

errno.**EINPROGRESS**
Operation now in progress. This error is mapped to the exception *BlockingIOError*.

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

Source code: Lib/ctypes

---

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 en este tutorial usan *doctest* para asegurarse de que realmente funcionen. Dado que algunos ejemplos de código se comportan de manera diferente en Linux, Windows o macOS, contienen directivas doctest 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).

None, integers, bytes objects and (unicode) strings are the only native Python objects that can directly be used as parameters in these function calls. None is passed as a C NULL pointer, bytes objects and strings are passed as pointer to the memory block that contains their data (char* or wchar_t*). Python integers are passed as the platforms default C int type, their value is masked to fit into the C type.

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
c_bool	_Bool	bool (1)
c_char	char	Un objeto bytes de 1-caracter
c_wchar	wchar_t	Una cadena de 1-caracter
c_byte	char	entero
c_ubyte	unsigned char	entero
c_short	short	entero
c_ushort	unsigned short	entero
c_int	int	entero
c_uint	unsigned int	entero
c_long	long	entero
c_ulong	unsigned long	entero
c_longlong	__int64 or long long	entero
c_ulonglong	unsigned __int64 or unsigned long long	entero
c_size_t	size_t	entero
c_ssize_t	ssize_t or Py_ssize_t	entero
c_float	float	flotante
c_double	double	flotante
c_longdouble	long double	flotante
c_char_p	char* (NUL terminated)	objeto de bytes o None
c_wchar_p	wchar_t* (NUL terminated)	cadena o None
c_void_p	void*	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)
>>> 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\x00'
>>>
```

The *create_string_buffer()* function replaces the `c_buffer()` function (which is still available as an alias), as well as the `c_string()` function from earlier ctypes releases. To create a mutable memory block containing unicode characters of the C type `wchar_t` use the *create_unicode_buffer()* function.

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 *PythonWin*:

```
>>> 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: 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
>>>
```

Calling variadic functions

On a lot of platforms calling variadic functions through ctypes is exactly the same as calling functions with a fixed number of parameters. On some platforms, and in particular ARM64 for Apple Platforms, the calling convention for variadic functions is different than that for regular functions.

On those platforms it is required to specify the *argtypes* attribute for the regular, non-variadic, function arguments:

```
libc.printf.argtypes = [ctypes.c_char_p]
```

Because specifying the attribute does inhibit portability it is advised to always specify `argtypes` for all variadic functions.

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: 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

By default functions are assumed to return the C `int` type. Other return types can be specified by setting the `restype` attribute of the function object.

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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: 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 cómo *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)]
...
>>> 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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()* qué 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=" ")
...     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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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])
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     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:

> This pointer is initialized to point to an array of `_frozen` records, terminated by one whose members are all `NULL` or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

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)]
...
>>>
```

We have defined the `_frozen` data type, so we can get the pointer to the table:

```
>>> 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 *descriptor*s 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 macOS, `find_library()` prueba varios esquemas de nombres y rutas predefinidos para ubicar la biblioteca, y retorna un nombre de ruta completo 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=None*)

Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return `int`.

En Windows, la creación de una instancia *CDLL* puede fallar incluso si existe el nombre de la DLL. Cuando no se encuentra una DLL dependiente de la DLL cargada, se lanza un error *OSError* con el mensaje *«[WinError 126] No se pudo encontrar el módulo especificado»*. Este mensaje de error no contiene el nombre de DLL que falta porque la API de Windows no devuelve esta información, lo que dificulta el diagnóstico de este error. Para resolver este error y determinar qué DLL no se encuentra, debe buscar la lista de DLL dependientes y determinar cuál no se encuentra utilizando las herramientas de depuración y seguimiento de Windows.

Ver también:

Herramienta Microsoft DUMPBIN – Una herramienta para encontrar dependientes de DLL.

class ctypes.**OleDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

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 lanza 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=None*)

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return `int` by default.

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 Python.

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)* del manual. 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.**oledll**
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**
An instance of *PyDLL* that exposes Python C API functions as attributes. Note that all these functions are assumed to return C `int`, which is of course not always the truth, so you have to assign the correct `restype` attribute to use these functions.

Lanza un *auditing event* `ctypes.dlopen` con argumento `name`.

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
Assign a ctypes type to specify the result type of the foreign function. Use `None` for `void`, a function not returning anything.

It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a C `int`, and the callable will be called with this integer, allowing further processing or error checking. Using this is deprecated, for more flexible post processing or error checking use a ctypes data type as *restype* and assign a callable to the *errcheck* attribute.

argtypes
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*)
Windows only: The returned function prototype creates functions that use the `stdcall` calling convention. The function will release the GIL during the call. *use_errno* and *use_last_error* have the same meaning as above.

ctypes.**PYFUNCTYPE**(*restype*, **argtypes*)
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 ex-

portada 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(
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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 *auditing event* `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 *auditing event* `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 *auditing event* `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*)
Intenta encontrar una biblioteca y retornar un nombre de ruta. *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.

ctypes.util.**find_msvcrt**()
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**()
Windows only: Returns the last error code set by Windows in the calling thread. This function calls the Windows `GetLastError()` function directly, it does not return the ctypes-private copy of the error code.

ctypes.**get_errno**()
Retorna el valor actual de la copia ctypes-private de la variable de sistema *errno* en el hilo de llamada.

Lanza un *auditing event* `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 *auditing event* `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* C: 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 *auditing event* `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 *auditing event* `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 *auditing event* `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 *auditing event* `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 *auditing event* `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 *auditing event* `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.

Lanza un *auditing event* `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**
Represents the C `signed char` datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_char**
Represents the C `char` datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.**c_char_p**
Represents the C `char*` datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, `POINTER(c_char)` must be used. The constructor accepts an integer address, or a bytes object.

class ctypes.**c_double**
Represents the C `double` datatype. The constructor accepts an optional float initializer.

class ctypes.**c_longdouble**
Represents the C `long double` datatype. The constructor accepts an optional float initializer. On platforms where `sizeof(long double) == sizeof(double)` it is an alias to *c_double*.

class ctypes.**c_float**
Represents the C `float` datatype. The constructor accepts an optional float initializer.

class ctypes.**c_int**
Represents the C `signed int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias to *c_long*.

class ctypes.**c_int8**
Represents the C 8-bit `signed int` datatype. Usually an alias for *c_byte*.

class ctypes.**c_int16**
Represents the C 16-bit `signed int` datatype. Usually an alias for *c_short*.

class ctypes.**c_int32**
Represents the C 32-bit `signed int` datatype. Usually an alias for *c_int*.

class ctypes.**c_int64**
Represents the C 64-bit `signed int` datatype. Usually an alias for *c_longlong*.

class ctypes.**c_long**
Represents the C `signed long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_longlong**
Represents the C `signed long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_short**
Represents the C `signed short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_size_t**
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**
Represents the C `unsigned char` datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_uint**
Represents the C `unsigned int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias for *c_ulong*.

class ctypes.**c_uint8**
Represents the C 8-bit `unsigned int` datatype. Usually an alias for *c_ubyte*.

class ctypes.**c_uint16**
Represents the C 16-bit `unsigned int` datatype. Usually an alias for *c_ushort*.

class ctypes.**c_uint32**
Represents the C 32-bit `unsigned int` datatype. Usually an alias for *c_uint*.

class ctypes.**c_uint64**
Represents the C 64-bit `unsigned int` datatype. Usually an alias for *c_ulonglong*.

class ctypes.**c_ulong**
Represents the C `unsigned long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ulonglong**
Represents the C `unsigned long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ushort**
Represents the C `unsigned short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_void_p**
Represents the C `void*` type. The value is represented as integer. The constructor accepts an optional integer initializer.

class ctypes.**c_wchar**
Represents the C `wchar_t` datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.**c_wchar_p**
Represents the C `wchar_t*` datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.

class ctypes.**c_bool**
Represent the C `bool` datatype (more accurately, `_Bool` from C99). Its value can be `True` or `False`, and the constructor accepts any object that has a truth value.

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**
Represents the C `PyObject*` datatype. Calling this without an argument creates a `NULL PyObject*` pointer.

El módulo `ctypes.wintypes` proporciona otros tipos de datos específicos de Windows, por ejemplo `HWND`, `WPARAM`, 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 *descriptor*s 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.

The recommended way to create concrete array types is by multiplying any *ctypes* data type with a non-negative integer. Alternatively, you can subclass this type and define *_length_* and *_type_* class variables. Array elements can be read and written using standard subscript and slice accesses; for slice reads, the resulting object is *not* itself an *Array*.

length
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

Retorna el objeto al que el puntero apunta. Asignando a este atributo cambia el puntero para que apunte al objeto asignado.

CAPÍTULO 17

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

This module constructs higher-level threading interfaces on top of the lower level *`_thread`* module.

Distinto en la versión 3.7: Este módulo solía ser opcional, ahora está siempre disponible.

Ver también:

`concurrent.futures.ThreadPoolExecutor` offers a higher level interface to push tasks to a background thread without blocking execution of the calling thread, while still being able to retrieve their results when needed.

`queue` provides a thread-safe interface for exchanging data between running threads.

`asyncio` offers an alternative approach to achieving task level concurrency without requiring the use of multiple operating system threads.

Nota: En la serie Python 2.x, este módulo contenía nombres `camelCase` para algunos métodos y funciones. Estos están obsoletos a partir de Python 3.10, pero aún son compatibles por compatibilidad con Python 2.5 y versiones anteriores.

Detalles de implementación de CPython: En CPython, debido al *Candado de intérprete global*, solo un hilo puede ejecutar código Python a la vez (aunque ciertas bibliotecas orientadas al rendimiento pueden superar esta limitación). Si desea que su aplicación haga un mejor uso de los recursos computacionales de las máquinas multinúcleo, se recomienda utilizar *`multiprocessing`* o *`concurrent.futures.ProcessPoolExecutor`*. Sin embargo, el subproceso sigue siendo un modelo apropiado si desea ejecutar varias tareas vinculadas a E/S simultáneamente.

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()*.

La función activeCount es un alias obsoleto para esta función.

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.

La función currentThread es un alias obsoleto para esta función.

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.**__excepthook__**
Mantiene el valor original de *threading.excepthook()*. Se guarda para que se pueda restaurar el valor original en caso de que se reemplacen con objetos rotos o alternativos.

Nuevo en la versión 3.10.

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 *Thread* actualmente activos. La lista incluye subprocesos demoníacos y objetos de subprocesos ficticios creados por *current_thread()*. Excluye los subprocesos terminados y los subprocesos que aún no se han iniciado. Sin embargo, el hilo principal siempre es parte del resultado, incluso cuando se termina.

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.**gettrace**()
Obtiene la función de rastreo según lo establecido por *settrace()*.

Nuevo en la versión 3.10.

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.**getprofile**()
Obtiene la función de generador de perfiles establecida por *setprofile()*.

Nuevo en la versión 3.10.

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 se 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. De forma predeterminada, se construye un nombre único con el formato «Hilo-*N*», donde *N* es un número decimal pequeño, o «Hilo-*N* (target)» donde «target» es `target.__name__` si se especifica el argumento *target*.

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 3.10: Utilice el nombre *target* si se omite el argumento *name*.

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 lanzaran 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()
API getter/setter obsoleta para *name*; utilícelo directamente como una propiedad en su lugar.

Obsoleto desde la versión 3.10.

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 incuso después de que el hilo ha abandonado.

native_id
El ID de subproceso (`TID`) de este subproceso, según lo asignado por el sistema operativo (kernel). Este es un número entero no negativo, o `None` si el hilo no se ha iniciado. Consulte la función

get_native_id(). Este valor se puede usar para identificar de forma única este hilo en particular en todo el sistema (hasta que el hilo termine, después de lo cual el sistema operativo puede reciclar el valor).

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 junto antes de que el método *run()* termine. La función *enumerate()* del módulo retorna una lista de todos los hilos vivos.

daemon
A boolean value indicating whether this thread is a daemon thread (`True`) or not (`False`). This must be set before *start()* is called, otherwise *RuntimeError* is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to *daemon* = `False`.

El programa de Python en su totalidad finaliza cuando no queda ningún hilo no-demonio vivo.

isDaemon()
setDaemon()
API getter/setter obsoleta para *daemon*; utilícelo directamente como una propiedad en su lugar.

Obsoleto desde la versión 3.10.

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 subsecuentes 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`.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. A *timeout* argument of `-1` specifies an unbounded wait. It is forbidden to specify a *timeout* when *blocking* is `False`.

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()
Return `True` if the lock is acquired.

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.

When invoked with the *blocking* argument set to `True`, do the same thing as when called without arguments, and return `True`.

When invoked with the *blocking* argument set to `False`, do not block. If a call without an argument would block, return `False` immediately; otherwise, do the same thing as when called without arguments, and return `True`.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. Return `True` if the lock has been acquired, `False` if the timeout has elapsed.

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 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*.

El método `notifyAll` es un alias obsoleto para este método.

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.

When invoked with *blocking* set to `False`, do not block. If a call without an argument would block, return `False` immediately; otherwise, do the same thing as when called without arguments, and return `True`.

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(*n=1*)
Suelta un semáforo, incrementando el contador interno por *n*. Cuando era cero en la entrada y otros subprocesos están esperando que vuelva a ser mayor que cero, active *n* de esos subprocesos.

Distinto en la versión 3.9: Se agregó el parámetro *n* para liberar varios subprocesos en espera a la vez.

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.

El método `isSet` es un alias obsoleto para este método.

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 es un paquete que permite crear procesos (*spawning*) utilizando una API similar al módulo *threading*. El paquete *multiprocessing* ofrece concurrencia tanto local como remota, esquivando el *Global Interpreter Lock* mediante el uso de subprocesos en lugar de hilos (*threads*). Debido a esto, el módulo *multiprocessing* le permite al programador aprovechar al máximo múltiples procesadores en una máquina determinada. Se ejecuta tanto en Unix como en 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]
```

Ver también:

concurrent.futures.ProcessPoolExecutor offers a higher level interface to push tasks to a background process without blocking execution of the calling process. Compared to using the *Pool* interface directly, the *concurrent.futures* API more readily allows the submission of work to the underlying process pool to be separated from waiting for the results.

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 subproceso. Consulte bpo-33725.

Distinto en la versión 3.4: *spawn* added on all Unix platforms, and *forkserver* added for some Unix platforms. Child processes no longer inherit all of the parents inheritable handles on 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'])

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,))
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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()
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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 subprocesos.

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:
        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:
            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:
```

(continué en la próxima página)

(proviene de la página anterior)

```
...   p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(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-enésimo* 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 qué 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
The child's exit code. This will be `None` if the process has not yet terminated.

If the child's *run()* method returned normally, the exit code will be 0. If it terminated via *sys.exit()* with an integer argument *N*, the exit code will be *N*.

If the child terminated due to an exception not caught within *run()*, the exit code will be 1. If it was terminated by signal *N*, the exit code will be the negative value *-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 autentificación*.

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 macOS 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 encolados, y es casi seguro que no necesitará usarlo. Realmente solo está allí si necesita que el proceso actual salga inmediatamente sin esperar a vaciar los datos en 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`.

close()
Cierra la cola: libera recursos internos.

Una cola no se debe usar más después de ser cerrada. Por ejemplo los métodos `get()`, `put()` y `empty()` no deben ser llamados.

Nuevo en la versión 3.9.

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))`

Cuando no se puede determinar el número de CPUs, se lanza un *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`.

El valor retornado puede ser `'fork'`, `'spawn'`, `'forkserver'` o `None`. En Unix `'fork'` es el valor predeterminado mientras que `'spawn'` lo es en Windows y 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 subproceso. Consulte bpo-33725.

Nuevo en la versión 3.4.

multiprocessing.**set_executable**(*executable*)
Set the path of the Python interpreter to use when starting a child process. (By default *sys.executable* is used). Embedders will probably need to do some thing like

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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 autentificación*.

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 macOS, 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 macOS, `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`.)

ctypes	*sharedctypes* usando *type*	*sharedctypes* usando *typecode*
c_double(2.4)	*RawValue(c_double, 2.4)*	*RawValue("d", 2.4)*
MyStruct(4, 6)	*RawValue(MyStruct, 4, 6)*	
*(c_short * 7)()*	*RawArray(c_short, 7)*	*RawArray('h', 7)*
*(c_int * 3)(9, 2, 8)*	*RawArray(c_int, (9, 2, 8))*	*RawArray("i", (9, 2, 8))*

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))
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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
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 lanzará 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 lanzará 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*]]]])
Una variante del método `apply()` que retorna un objeto `AsyncResult`.

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*]]])
Una variante del método `map()` que retorna un objeto `AsyncResult`.

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*])
Like *map()* except that the elements of the *iterable* are expected to be iterables that are unpacked as arguments.

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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 autentificación*.

`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 autentificación*.

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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
            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.
- Una dirección `'AF_UNIX'` es una cadena que representa un nombre de archivo en el sistema de archivos.
- 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.

Tenga en cuenta que cualquier cadena que comience con dos barras inclinadas invertidas se asume por defecto como una dirección `'AF_PIPE'` en lugar de una dirección `'AF_UNIX'`.

Llaves de autentificación

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`**`(`*level=None*`)`
This function performs a call to *get_logger()* but in addition to returning the logger created by get_logger, it adds a handler which sends output to *sys.stderr* using format `'[%(levelname)s/%(processName)s] %(message)s'`. You can modify `levelname` of the logger by passing a `level` argument.

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 /.../pymp-...
[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*.

En particular, la función `Pool` ofrecida por *multiprocessing.dummy* retorna un instancia de *ThreadPool*, la cual es un subclase de *Pool* que soporta todas las mismas llamadas, pero usa un *pool* de hebras trabajadoras en vez de procesos trabajadores.

class multiprocessing.pool.**ThreadPool**([*processes*[, *initializer*[, *initargs*]]])

Un objeto de *pool* de hebras que controla un *pool* de hebras trabajadoras a las cuales se pueden enviar trabajos. La interfaz de las instancias de *ThreadPool* son totalmente compatibles con las instancias de *Pool*, y sus recursos también deben ser debidamente administrador, ya sea usando el *pool* como un gestor de contexto, o llamando a *close()* y *terminate()* manualmente.

processes es el número de hebras 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.

A diferencia de *Pool*, *maxtasksperchild* and *context* no se pueden entregar.

> **Nota:** Un *ThreadPool* comparte la misma interfaz que *Pool*, la cual está diseñada alrededor de un *pool* de procesos, y precede la introducción del módulo *concurrent.futures*. Como tal, hereda algunas operaciones que no tienen sentido para un *pool* basado en hebras, y tiene su propio tipo para representar el estado de trabajos asíncronos, *AsyncResult*, el cual no es entendido por otras librerías.
>
> Los usuarios deberían por lo general preferir usar *concurrent.futures.ThreadPoolExecutor*, el cual tiene una interfaz más simple que fue diseñada alrededor de hebras desde un principio, y que retorna instancias de *concurrent.futures.Future*, las que son compatibles con muchas más librerías, incluyendo *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():
    ... 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 (*picklable*). 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)

##
```

(continué en la próxima página)

(proviene de la página anterior)

```
def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    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
        #
```

(continué en la próxima página)

(proviene de la página anterior)

```
        print('Testing error handling:')

        try:
            print(pool.apply(f, (5,)))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.apply()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(pool.map(f, 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:
```

(continué en la próxima página)

(proviene de la página anterior)

```
                sys.stdout.write('\n\t%s' % it.next(0.02))
            except StopIteration:
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()


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
```

(continué en la próxima página)

(proviene de la página anterior)

```
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print('Unordered results:')
    for i in range(len(TASKS1)):
        print('\t', done_queue.get())

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print('\t', done_queue.get())

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')


if __name__ == '__main__':
    freeze_support()
    test()
```

17.3 `multiprocessing.shared_memory` — Shared memory for direct access across processes

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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> shm_a.close()
>>> shm_a.unlink()  # Call unlink only once to release the shared memory
```

The following example demonstrates a practical use of the *SharedMemory* class with NumPy arrays, accessing the same `numpy.ndarray` from two distinct Python shells:

```
>>> # 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 sentencias 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()
```

El siguiente ejemplo demuestra que los objetos `ShareableList` (y de forma implícita `SharedMemory`) pueden ser serializados (*pickled*) y deserializados (*unpickled*) si es que se necesitan. Nota, Este va a seguir siendo el mismo objeto compartido. Esto sucede, porque el objeto deserializado tiene el mismo nombre único y simplemente se adjunta a un objeto existente con el mismo nombre (si el objeto todavía sigue vivo):

```
>>> import pickle
>>> from multiprocessing import shared_memory
>>> sl = shared_memory.ShareableList(range(10))
>>> list(sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> deserialized_sl = pickle.loads(pickle.dumps(sl))
>>> list(deserialized_sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sl[0] = -1
>>> deserialized_sl[1] = -2
>>> list(sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(deserialized_sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sl.shm.close()
>>> sl.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 subclases.

submit(*fn*, */*, **args*, ***kwargs*)

Schedules the callable, *fn*, to be executed as `fn(*args, **kwargs)` and returns a *`Future`* object representing the execution of the callable.

```
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.

Al usar `ProcessPoolExecutor`, este método divide los *iterables* en varios fragmentos que luego envía al grupo como tareas separadas. El tamaño (aproximado) de estos fragmentos puede especificarse estableciendo *chunksize* a un entero positivo. El uso de un valor grande para *chunksize* puede mejorar significativamente el rendimiento en comparación con el tamaño predeterminado de 1. Con `ThreadPoolExecutor`, el *chunksize* no tiene ningún efecto.

Distinto en la versión 3.5: Se agregó el argumento *chunksize*.

shutdown(*wait=True*, ***, *cancel_futures=False*)

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.

Si *cancel_futures* es `True`, este método cancelará todos los futuros pendientes que el ejecutor no ha comenzado a ejecutar. Los futuros que se completen o estén en ejecución no se cancelarán, independientemente del valor de *cancel_futures*.

Si tanto *cancel_futures* como *wait* son `True`, todos los futuros que el ejecutor ha comenzado a ejecutar se completarán antes de que regrese este método. Los futuros restantes se cancelan.

Se puede evitar tener que llamar este método explícitamente si se usa la sentencia `with`, la cual apagará el `Executor` (esperando como si `Executor.shutdown()` hubiera sido llamado con *wait* en `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')
```

Distinto en la versión 3.9: Agregado *cancel_futures*.

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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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*, *thread_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.

All threads enqueued to `ThreadPoolExecutor` will be joined before the interpreter can exit. Note that the exit handler which does this is executed *before* any exit handlers added using `atexit`. This means exceptions in the main thread must be caught and handled in order to signal threads to exit gracefully. For this reason, it is recommended that `ThreadPoolExecutor` not be used for long-running tasks.

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 numero 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

La clase *ProcessPoolExecutor* es una subclase *Executor* que usa un grupo de procesos para ejecutar llamadas de forma asíncrona. *ProcessPoolExecutor* usa el módulo *multiprocessing*, que le permite eludir el *Global Interpreter Lock* pero también significa que solo los objetos seleccionables pueden ser ejecutados y retornados.

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=()*)

Subclase de *Executor* que ejecuta llamadas asincrónicas mediante un grupo de, como máximo, *max_workers* procesos. Si *max_workers* es `None` o no fue especificado, el número predeterminado será la cantidad de procesadores de la máquina, Si *max_workers* es menor o igual a `0`, la excepción *ValueError* será lanzada. En Windows, *max_workers* debe ser menor o igual a `61`. Si no es así, la excepción *ValueError* será lanzada. Si *max_workers* es `None`, el número predeterminado será `61` como máximo, aún si existen más procesadores disponibles. *mp_context* puede ser un contexto de multiprocesamiento o `None` y será utilizado para iniciar los trabajadores. Si *mp_context* es `None` o no es especificado, se utilizará el contexto predeterminado de multiprocesamiento.

initializer es un invocable opcional que es llamado al comienzo de cada proceso trabajador; *initargs* es una tupla de argumentos pasados al inicializador. Si el *initializer* lanza una excepción, todos los trabajos actualmente pendientes lanzarán `BrokenProcessPool`, así como cualquier intento de enviar más trabajos al grupo.

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.

Los invocables agregados se llaman en el orden en que se agregaron y siempre se llaman en un hilo que pertenece al proceso que los agregó. Si el invocable genera una subclase `Exception`, se registrará y se ignorará. Si el invocable genera una subclase `BaseException`, el comportamiento no está definido.

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.

If the method returns `False` then the `Future` was cancelled, i.e. `Future.cancel()` was called and returned `True`. Any threads waiting on the `Future` completing (i.e. through `as_completed()` or `wait()`) will be woken up.

If the method returns `True` then the `Future` was not cancelled and has been put in the running state, i.e. calls to `Future.running()` will return `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*)
Wait for the *Future* instances (possibly created by different *Executor* instances) given by *fs* to complete. Duplicate futures given to *fs* are removed and will be returned only once. Returns a named 2-tuple of sets. The first set, named `done`, contains the futures that completed (finished or cancelled futures) before the wait completed. The second set, named `not_done`, contains the futures that did not complete (pending or running futures).

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
`FIRST_COMPLETED`	La función retornará cuando cualquier futuro finalice o sea cancelado.
`FIRST_EXCEPTION`	La función retornará cuando cualquier futuro finalice lanzando una excepción. Si ningún futuro lanza una excepción, esta opción es equivalente a `ALL_COMPLETED`.
`ALL_COMPLETED`	La función retornará cuando todos los futuros finalicen o sean cancelados.

concurrent.futures.**as_completed**(*fs*, *timeout=None*)
Retorna un iterador sobre las instancias de *Future* (posiblemente creadas por distintas instancias de *Executor*) dadas por *fs* que produce futuros a medida que van finalizando (normalmente o cancelados). Cualquier futuro dado por *fs* que esté duplicado será retornado una sola vez. Los futuros que hayan finalizado antes de la llamada a *as_completed()* serán entregados primero. El iterador retornado lanzará *concurrent.futures.TimeoutError* si *__next__()* es llamado y el resultado no está disponible luego de *timeout* segundos a partir de la llamada original a *as_completed()*. *timeout* puede ser un int o un float. Si *timeout* no es especificado o es `None`, no hay limite en el tiempo de espera.

Ver también:

PEP 3148 – futuros - ejecutar cómputos 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 subprocesos

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 subprocesos 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`*.

`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 subproceso. 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.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process" environment. It is passed directly to *Popen*. This mapping can be str to str on any platform or bytes to bytes on POSIX platforms much like *os.environ* or *os.environb*.

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 3.10.11: 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
Output of the child process if it was captured by *run()* or *check_output()*. Otherwise, None. This is always *bytes* when any output was captured regardless of the text=True setting. It may remain None instead of b'' when no output was observed.

stdout
Alias de *output*, por simetría con *stderr*.

stderr
Stderr output of the child process if it was captured by *run()*. Otherwise, None. This is always *bytes* when stderr output was captured regardless of the text=True setting. It may remain None instead of b'' when no stderr output was observed.

Nuevo en la versión 3.3.

Distinto en la versión 3.5: Se añadieron los atributos *stdout* y *stderr*

exception subprocess.**CalledProcessError**
Subclass of *SubprocessError*, raised when a process run by *check_call()*, *check_output()*, or *run()* (with check=True) returns a non-zero exit status.

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* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are *PIPE*, *DEVNULL*, an existing file descriptor (a positive integer), an existing file object with a valid file descriptor, and `None`. *PIPE* indicates that a new pipe to the child should be created. *DEVNULL* indicates that the special file *os.devnull* will be used. With the default settings of `None`, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be *STDOUT*, which indicates that the stderr data from the child process should be captured into the same file handle as for *stdout*.

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 del 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=()*, ***, *group=None*, *extra_groups=None*, *user=None*, *umask=-1*, *encoding=None*, *errors=None*, *text=None*, *pipesize=-1*)

Ejecuta un programa hijo en un proceso nuevo. En POSIX, la clase usa *os.execvp()* como comportamiento 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.

> **Advertencia:** For maximum reliability, use a fully qualified path for the executable. To search for an unqualified name on `PATH`, use *shutil.which()*. On all platforms, passing *sys.executable* is the recommended way to launch the current Python interpreter again, and use the `-m` command-line format to launch an installed module.
>
> La resolución de la ruta de *executable* (o el primer elemento de *args*) depende de la plataforma. Para POSIX, consulte *os.execvpe()*, y tenga en cuenta que al resolver o buscar la ruta ejecutable, *cwd* anula el directorio de trabajo actual y *env* puede anular la variable de entorno `PATH`. Para Windows, consulte la documentación de los parámetros `lpApplicationName` e `lpCommandLine` de WinAPI `CreateProcess`, y tenga en cuenta que al resolver o buscar la ruta ejecutable con `shell = False`, *cwd* no anula el directorio de trabajo actual y *env* no puede anular la variable de entorno `PATH`. El uso de una ruta completa evita todas estas variaciones.

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` means line buffered (only usable if `text=True` or `universal_newlines=True`)
- 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 3.10.11: 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* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are `PIPE`, `DEVNULL`, an existing file descriptor (a positive integer), an existing *file object* with a valid file descriptor, and `None`. `PIPE` indicates that a new pipe to the child should be created. `DEVNULL` indicates that the special file `os.devnull` will be used. With the default settings of `None`, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be `STDOUT`, which indicates that the stderr data from the applications should be captured into the same file handle as for 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 POSIX, 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 subproceso (solamente POSIX).

Distinto en la versión 3.2: Se añadió *start_new_session*.

Si *group* no es `None`, la llamada a sistema setregid() se hará en el proceso hijo antes de la ejecución del subproceso. Si el valor proveído es una cadena de caracteres, será buscado usando *grp.getgrnam()* y el valor en `gr_gid` será usado. Si el valor es un entero, será pasado literalmente. (solamente POSIX)

Disponibilidad: POSIX

Nuevo en la versión 3.9.

Si *extra_groups* no es `None`, la llamada a sistema setgroups() se hará en el proceso hijo antes de la ejecución del subproceso. Cadenas de caracteres proveídas en *extra_groups* serán buscadas usando *grp.getgrnam()* y los valores en `gr_gid` serán usados. Valor enteros serán pasados literalmente. (solamente POSIX)

Disponibilidad: POSIX

Nuevo en la versión 3.9.

Si *user* no es `None`, la llamada a sistema setreuid() se hará en el proceso hijo antes de la ejecución del subproceso. Si el valor proveído es una cadena de caracteres, será buscado usando *pwd.getpwnam()* y el valor en `pw_uid` será usado. Si el valor es un entero, será pasado literalmente. (solamente POSIX)

Disponibilidad: POSIX

Nuevo en la versión 3.9.

Si *umask* no es negativo, la llamada a sistema umask() se hará en el proceso hijo antes de la ejecución del subproceso.

Disponibilidad: POSIX

Nuevo en la versión 3.9.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process" environment. This mapping can be str to str on any platform or bytes to bytes on POSIX platforms much like *os.environ* or *os.environb*.

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*

pipeize se puede usar para cambiar el tamaño de la tubería cuando *PIPE* se usa para *stdin*, *stdout* o *stderr*. El tamaño de la tubería solo se cambia en plataformas que lo admiten (solo Linux en este momento de redacción). Otras plataformas ignorarán este parámetro.

Nuevo en la versión 3.10: Se añadió el parámetro `pipesize`.

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 común que se lanza es *OSError*. Esto ocurre, por ejemplo, al intentar ejecutar un archivo inexistente. Las aplicaciones deben prepararse para excepciones *OSError*. Tenga en cuenta que, cuando `shell = True`, *OSError* será lanzado por el hijo solo si no se encontró el shell seleccionado. Para determinar si el shell no pudo encontrar la aplicación solicitada, es necesario verificar el código de retorno o la salida del subproceso.

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

Al contrario que otras funciones popen, esta implementación nunca llamará implícitamente a la shell del sistema. Esto significa que todos los caracteres, incluidos los metacaracteres de la shell, se pueden pasar de manera segura a los procesos hijos. Si se invoca la shell explícitamente, mediante `shell=True`, es responsabilidad de la aplicación asegurar que todo el espaciado y metacaracteres se entrecomillan adecuadamente para evitar vulnerabilidades de inyección de código. En *algunas plataformas*, es posible usar *shlex.quote()* para este escape.

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*)
Interactúa con el proceso: Envía datos a stdin. Lee datos de stdout y stderr hasta encontrar un fin-de-fichero. Espera a que termine el proceso y escribe el atributo *returncode*. El argumento opcional *input* debe contener los datos que hay que enviar al proceso hijo o None si no hay que enviar datos al proceso hijo. Si se abrieron los flujos en modo texto, *input* ha de ser una cadena de caracteres. En caso contrario, debe contener 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.

No hace nada si el proceso ya ha terminado.

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 *CREATE_NEW_PROCESS_GROUP*.

Popen.**terminate**()
Detiene el proceso hijo. En sistemas operativos POSIX este método envía SIGTERM al proceso hijo. En Windows se llama a la función TerminateProcess() de la API de Win32 para detener el proceso hijo.

Popen.**kill**()
Mata el proceso hijo. En sistemas operativos POSIX la función envía SIGKILL al proceso hijo. En Windows *kill()* es un alias de *terminate()*.

The following attributes are also set by the class for you to access. Reassigning them to new values is unsupported:

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**
If the *stdin* argument was *PIPE*, this attribute is a writeable stream object as returned by *open()*. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was True, the stream is a text stream, otherwise it is a byte stream. If the *stdin* argument was not *PIPE*, this attribute is None.

Popen.**stdout**
If the *stdout* argument was *PIPE*, this attribute is a readable stream object as returned by *open()*. Reading from the stream provides output from the child process. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was True, the stream is a text stream, otherwise it is a byte stream. If the *stdout* argument was not *PIPE*, this attribute is None.

Popen.**stderr**
If the *stderr* argument was *PIPE*, this attribute is a readable stream object as returned by *open()*. Reading from the stream provides error output from the child process. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was True, the stream is a text stream, otherwise it is a byte stream. If the *stderr* argument was not *PIPE*, this attribute is 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 del 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**
Oculta 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 subproceso.

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 subprocesos. 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*, ***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 3.10.11: 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*, ***other_popen_kwargs*)
Ejecuta la instrucción con argumentos. Espera a que la instrucción se complete. Si el código de retorno es cero retorna; en caso contrario, lanza *CalledProcessError*. El objeto *CalledProcessError* tendrá el código de retorno en el atributo *returncode*.Si *check_call()* no pudo iniciar el proceso, propagará la excepción que fue lanzada.

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 3.10.11: 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
```

Los argumentos mostrados arriba son meramente algunos de los comunes. La signatura completa de la función es casi la misma que la de *run()* - la mayoría de los argumentos se pasa directamente a esa interfaz. Existe una diferencia con la interfaz de *run()* respecto al comportamiento: pasar `input=None` genera el mismo comportamiento que `input=b''` (o `input=''`, dependiendo de otros argumentos) en vez de usar el flujo entrada estándar del padre.

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 3.10.11: 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 remplazar 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
retcode = call("mycmd" + " myarg", shell=True)
```

Notas:

- No suele hacer falta llamar al programa a través de la shell.
- El valor de retorno de *call()* se codifica de forma diferente al de *os.system()*.
- La función *os.system()* ignora las señales SIGINT y SIGQUIT mientras se ejecuta el comando, pero el llamador debe hacerlo por separado cuando usa el módulo *subprocess*.

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)
==>
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,
```

(continué en la próxima página)

(proviene de la página anterior)

```
          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',))
...     # despite having higher priority, 'keyword' runs after 'positional' as␣
↪enter() is relative
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.enterabs(1_650_000_000, 10, print_time, argument=("first enterabs",))
...     s.enterabs(1_650_000_000, 5, print_time, argument=("second enterabs",))
...     s.run()
...     print(time.time())
...
>>> print_some_times()
1652342830.3640375
From print_time 1652342830.3642538 second enterabs
From print_time 1652342830.3643398 first enterabs
From print_time 1652342835.3694863 positional
From print_time 1652342835.3696074 keyword
From print_time 1652342840.369612 default
1652342840.3697174
```

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 lanzará 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.

The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a FIFO queue, the first tasks added are the first retrieved. In a LIFO queue, the most recently added entry is the first retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the `heapq` module) and the lowest valued entry is retrieved first.

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.

The lowest valued entries are retrieved first (the lowest valued entry is the one that would be returned by `min(entries)`). A typical pattern for entries is a tuple in the form: `(priority_number, data)`.

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*)
Put *item* into the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *Full* exception if no free slot was available within that time. Otherwise (*block* is false), put an item on the queue if a free slot is immediately available, else raise the *Full* exception (*timeout* is ignored in that case).

`Queue.put_nowait`(*item*)
Equivalent to `put(item, block=False)`.

`Queue.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 predeterminado), 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).

Prior to 3.0 on POSIX systems, and for all versions on Windows, if *block* is true and *timeout* is `None`, this operation goes into an uninterruptible wait on an underlying lock. This means that no exceptions can occur, and in particular a SIGINT will not trigger a *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.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls *task_done()* to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, *join()* unblocks.

Ejemplo de cómo esperar a que se completen las tareas en cola:

```
import threading
import 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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
    q.put(item)

# 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**()
Return `True` if the queue is empty, `False` otherwise. If empty() returns `False` it doesn't guarantee that a subsequent call to get() will not block.

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()*.

Detalles de implementación de CPython: 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*)
Equivalent to `put(item, block=False)`, provided for compatibility with *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 predeterminado), 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
A read-only property. Set to the value the variable had before the *ContextVar.set()* method call that created the token. It points to *Token.MISSING* if the variable was not set before the call.

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**
Mapeo 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()*.

Every thread will have a different top-level *Context* object. This means that a *ContextVar* object behaves in a similar fashion to *threading.local()* when values are assigned in different threads.

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'
```

(continué en la próxima página)

(proviene de la página anterior)

```
# 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()
Retorna una copia superficial (*shallow copy*) del objeto de contexto.

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 variables 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()
```

(continué en la próxima página)

(proviene de la página anterior)

```
        print(line)
        if not line.strip():
            break
        writer.write(line)

    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**(*signum=signal.SIGINT*, */*)
: Simular el efecto de una señal que llega al hilo principal. Un hilo puede usar esta función para interrumpir el hilo principal, aunque no hay garantías de que la interrupción ocurrirá inmediatamente.

Si se da, *signum* es el número de la señal a simular. Si *signum* no se da, `signal.SIGINT` es simulado.

Si la señal dada no es manejada por Python (se estableció en `signal.SIG_DFL` o `signal.SIG_IGN`), esta función no hace nada.

Distinto en la versión 3.10: Se agrega el argumento *signum* para personalizar el número de señal.

Nota: Esto no emite la señal correspondiente, sino que programa una llamada al controlador asociado (si existe). Si realmente se quiere emitir la señal, se utiliza `signal.raise_signal()`.

`_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`**`(`*blocking=True*, *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).

If the *blocking* argument is present, the action depends on its value: if it is False, the lock is only acquired if it can be acquired immediately without waiting, while if it is True, the lock is acquired unconditionally as above.

If the floating-point *timeout* argument is present and positive, it specifies the maximum wait time in seconds before returning. A negative *timeout* argument specifies an unbounded wait. You cannot specify a *timeout* if *blocking* is False.

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.

CAPÍTULO 18

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!')

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*.

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'
```

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*
- *Ejecutando en Hilos*
- *Planificación Desde Otros Hilos*
- *Introspección*
- *Objeto Task*
- *Corrutinas basadas en generadores*

Corrutinas

Source code: Lib/asyncio/coroutines.py

Coroutines declared with the async/await syntax is the preferred way of writing asyncio applications. For example, the following snippet of code prints «hello», waits 1 second, and then prints «world»:

```
>>> 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(
```

(continué en la próxima página)

(proviene de la página anterior)

```
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        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 **Futures**.

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())
```

Futures

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, *finalizing asynchronous generators* y cerrando el threadpool.

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.

Distinto en la versión 3.9: Actualizado para usar *loop.shutdown_default_executor()*.

Nota: El código fuente para `asyncio.run()` se puede encontrar en Lib/asyncio/runners.py.

Creando Tareas

Source code: Lib/asyncio/tasks.py

`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.

Importante: Save a reference to the result of this function, to avoid a task disappearing mid-execution. The event loop only keeps weak references to tasks. A task that isn't referenced elsewhere may get garbage collected at any time, even before it's done. For reliable «fire-and-forget» background tasks, gather them in a collection:

```
background_tasks = set()

for i in range(10):
    task = asyncio.create_task(some_coro(param=i))

    # Add task to the set. This creates a strong reference.
    background_tasks.add(task)

    # To prevent keeping references to finished tasks forever,
    # make each task remove its own reference from the set after
    # completion:
    task.add_done_callback(background_tasks.discard)
```

Nuevo en la versión 3.7.

Distinto en la versión 3.8: Added the *name* parameter.

Durmiendo

coroutine asyncio.**sleep**(*delay*, *result=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.

Establecer el retraso en 0 proporciona una ruta optimizada para permitir que se ejecuten otras tareas. Esto puede ser utilizado por funciones de ejecución prolongada para evitar bloquear el bucle de eventos durante toda la duración de la llamada a la función.

Deprecated since version 3.8, removed in version 3.10: El parámetro `loop`. Esta función ha estado obteniendo implícitamente el ciclo en ejecución actual desde 3.7. Consulte What's New in 3.10's Removed section para obtener más información. 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())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

asyncio.run(display_date())
```

Distinto en la versión 3.10: Removed the *loop* parameter.

Ejecutando Tareas Concurrentemente

awaitable asyncio.**gather**(**aws*, *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 Future 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/Future enviada para hacer que otras Tareas/Futures sean canceladas.

Distinto en la versión 3.10: Removed the *loop* parameter. Ejemplo:

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({number}), currently i={i}...")
```

(continué en la próxima página)

(proviene de la página anterior)

```
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")
    return f

async def main():
    # Schedule three calls *concurrently*:
    L = await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )
    print(L)

asyncio.run(main())

# Expected output:
#
#     Task A: Compute factorial(2), currently i=2...
#     Task B: Compute factorial(3), currently i=2...
#     Task C: Compute factorial(4), currently i=2...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3), currently i=3...
#     Task C: Compute factorial(4), currently i=3...
#     Task B: factorial(3) = 6
#     Task C: Compute factorial(4), currently i=4...
#     Task C: factorial(4) = 24
#     [2, 6, 24]
```

Nota: Si *return_exceptions* es False, cancelar gather() después de que se haya marcado como hecho no cancelará ninguna espera enviada. Por ejemplo, la recopilación se puede marcar como hecha después de propagar una excepción a la persona que llama, por lo tanto, llamar a `gather.cancel()` después de detectar una excepción (generada por uno de los elementos pendientes) de recopilación no cancelará ningún otro elemento pendiente.

Distinto en la versión 3.7: Si se cancela el propio *gather*, la cancelación se propaga independientemente de *return_exceptions*.

Distinto en la versión 3.10: Removed the *loop* parameter.

Obsoleto desde la versión 3.10: Se emite una advertencia de obsolescencia si no se proporcionan argumentos posicionales o no todos los argumentos posicionales son objetos de tipo Future y no hay un bucle de eventos en ejecución.

Protección contra Cancelación

awaitable asyncio.**shield**(*aw*)

Protege un *objeto esperable* de ser *cancelado*.

Si *aw* es una corrutina, se programa automáticamente como una Tarea.

La declaración:

```
task = asyncio.create_task(something())
res = await shield(task)
```

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:

```
task = asyncio.create_task(something())
try:
    res = await shield(task)
except CancelledError:
    res = None
```

Importante: Save a reference to tasks passed to this function, to avoid a task disappearing mid-execution. The event loop only keeps weak references to tasks. A task that isn't referenced elsewhere may get garbage collected at any time, even before it's done.

Distinto en la versión 3.10: Removed the *loop* parameter.

Obsoleto desde la versión 3.10: Se emite una advertencia de obsolescencia si *aw* no es un objeto similares a Futures y no hay un bucle de eventos en ejecución.

Tiempo agotado

coroutine asyncio.**wait_for**(*aw*, *timeout*)

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 Future se completa.

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 se cancele el Future, por lo que el tiempo de espera total puede exceder el *timeout*. Si ocurre una excepción durante la cancelación, se propaga.

Si se cancela la espera, el Future *aw* también se cancela.

Distinto en la versión 3.10: Removed the *loop* parameter. 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())
```

(continué en la próxima página)

(proviene de la página anterior)

```
# 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*.

Distinto en la versión 3.10: Removed the *loop* parameter.

Esperando Primitivas

coroutine asyncio.**wait**(*aws*, ***, *timeout=None*, *return_when=ALL_COMPLETED*)

Ejecuta *objetos en espera* en el *aws* iterable simultáneamente y bloquee hasta la condición especificada por *return_when*.

El iterable *aws* no debe estar vacío.

Retorna dos conjuntos de Tareas/Futures: `(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 Futures 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
`FIRST_COMPLETED`	La función retornará cuando cualquier Future termine o se cancele.
`FIRST_EXCEPTION`	La función retornará cuando cualquier Future finalice lanzando una excepción. Si ningún Future lanza una excepción, entonces es equivalente a `ALL_COMPLETED`.
`ALL_COMPLETED`	La función retornará cuando todos los Futures terminen o se cancelen.

A diferencia de *wait_for()*, `wait()` no cancela los Futures 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*.

Distinto en la versión 3.10: Removed the *loop* parameter.

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, se eliminará en la versión 3.11: El paso de objetos corrutina a `wait()` directamente está en desuso.

Distinto en la versión 3.10: Removed the *loop* parameter.

`asyncio.`**`as_completed`**(*aws*, *, *timeout=None*)

Ejecuta *objetos en espera* en el *aws* iterable al mismo tiempo. Devuelve un iterador de corrutinas. Se puede esperar a cada corrutina devuelta para obtener el siguiente resultado más temprano del iterable de los esperables restantes.

Lanza *asyncio.TimeoutError* si el agotamiento de tiempo ocurre antes que todos los Futures terminen.

Ejemplo:

```
for coro in as_completed(aws):
    earliest_result = await coro
    # ...
```

Distinto en la versión 3.10: Removed the *loop* parameter.

Obsoleto desde la versión 3.10: Se emite una advertencia de obsolescencia si no todos los objetos en espera en el iterable *aws* son objetos de tipo Future y no hay un bucle de eventos en ejecución.

Ejecutando en Hilos

coroutine `asyncio.`**`to_thread`**(*func*, */*, **args*, ***kwargs*)

Ejecutar asincrónicamente la función *func* en un hilo separado.

Cualquier *args y **kwargs suministrados para esta función se pasan directamente a *func*. Además, el current *contextvars.Context* se propaga, lo que permite acceder a las variables de contexto del subproceso del bucle de eventos en el subproceso separado.

Retorna una corrutina que se puede esperar para obtener el resultado final de *func*.

Esta función de corrutina está destinada principalmente a ser utilizada para ejecutar funciones/métodos vinculados a IO que de otro modo bloquearían el bucle de eventos si se ejecutaran en el hilo principal. Por ejemplo:

```
def blocking_io():
    print(f"start blocking_io at {time.strftime('%X')}")
    # Note that time.sleep() can be replaced with any blocking
    # IO-bound operation, such as file operations.
    time.sleep(1)
    print(f"blocking_io complete at {time.strftime('%X')}")

async def main():
    print(f"started main at {time.strftime('%X')}")

    await asyncio.gather(
        asyncio.to_thread(blocking_io),
        asyncio.sleep(1))

    print(f"finished main at {time.strftime('%X')}")
```

(continué en la próxima página)

(proviene de la página anterior)

```
asyncio.run(main())

# Expected output:
#
# started main at 19:50:53
# start blocking_io at 19:50:53
# blocking_io complete at 19:50:54
# finished main at 19:50:54
```

Directly calling `blocking_io()` in any coroutine would block the event loop for its duration, resulting in an additional 1 second of run time. Instead, by using `asyncio.to_thread()`, we can run it in a separate thread without blocking the event loop.

Nota: Due to the *GIL*, `asyncio.to_thread()` can typically only be used to make IO-bound functions non-blocking. However, for extension modules that release the GIL or alternative Python implementations that don't have one, `asyncio.to_thread()` can also be used for CPU-bound functions.

Nuevo en la versión 3.9.

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 Future retornado será notificado. También se puede utilizar para cancelar la tarea en el bucle de eventos:

```
try:
    result = future.result(timeout)
except concurrent.futures.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 Future* 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 Future, la Tarea suspende la ejecución de la corrutina y espera la finalización del Future. Cuando el Future *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 Future, 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 Future durante la cancelación, se cancelará el objeto Future.

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: Added the *name* parameter.

Obsoleto desde la versión 3.10: Se emite una advertencia de obsolescencia si no se especifica *loop* y no hay un bucle de eventos en ejecución.

cancel(*msg=None*)
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.

Distinto en la versión 3.9: Added the *msg* parameter. 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 predeterminada 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.)ss

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()`.

The *file* argument is an I/O stream to which the output is written; by default output is written to `sys.stdout`.

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.

Corrutinas basadas en generadores

Nota: Support for generator-based coroutines is **deprecated** and is removed in Python 3.11.

Las corrutinas basadas en generadores son anteriores a la sintaxis async/await. Son generadores de Python que utilizan expresiones `yield from` para esperar en Futures 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`.

Obsoleto desde la versión 3.8, se eliminará en la versión 3.11: 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}')
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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*, ***, *limit=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*)

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`.

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()`.

Nota: The *sock* argument transfers ownership of the socket to the `StreamWriter` created. To close the socket, call its `close()` method.

Distinto en la versión 3.7: Added the *ssl_handshake_timeout* parameter.

Nuevo en la versión 3.8: Added *happy_eyeballs_delay* and *interleave* parameters.

Distinto en la versión 3.10: Removed the *loop* parameter.

coroutine asyncio.**start_server**(*client_connected_cb*, *host=None*, *port=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`.

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()`.

Nota: The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server's `close()` method.

Distinto en la versión 3.7: Added the *ssl_handshake_timeout* and *start_serving* parameters.

Distinto en la versión 3.10: Removed the *loop* parameter.

Sockets Unix

coroutine asyncio.**open_unix_connection**(*path=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()*.

Nota: The *sock* argument transfers ownership of the socket to the *StreamWriter* created. To close the socket, call its *close()* method.

Disponibilidad: Unix.

Distinto en la versión 3.7: Added the *ssl_handshake_timeout* parameter. The *path* parameter can now be a *path-like object*

Distinto en la versión 3.10: Removed the *loop* parameter.

coroutine asyncio.**start_unix_server**(*client_connected_cb*, *path=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()*.

Nota: The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server's *close()* method.

Disponibilidad: Unix.

Distinto en la versión 3.7: Added the *ssl_handshake_timeout* and *start_serving* parameters. The *path* parameter can now be a *path-like object*.

Distinto en la versión 3.10: Removed the *loop* parameter.

StreamReader

class asyncio.**StreamReader**

Represents a reader object that provides APIs to read data from the IO stream. As an *asynchronous iterable*, the object supports the `async for` statement.

No se recomienda crear instancias de objetos *StreamReader* directamente; utilice *open_connection()* y *start_server()* en su lugar.

coroutine read(*n=-1*)

Read up to *n* bytes from the stream.

If *n* is not provided or set to `-1`, read until EOF, then return all read *bytes*. If EOF was received and the internal buffer is empty, return an empty `bytes` object.

If *n* is `0`, return an empty `bytes` object immediately.

If *n* is positive, return at most *n* available `bytes` as soon as at least 1 byte is available in the internal buffer. If EOF is received before any byte is read, return an empty `bytes` object.

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.

The method should be used, though not mandatory, along with the `wait_closed()` method:

```
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.

Should be called after *close()* to wait until the underlying connection is closed, ensuring that all data has been flushed before e.g. exiting the program.

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()
    await writer.wait_closed()

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()
    await writer.wait_closed()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addrs = ', '.join(str(sock.getsockname()) for sock in server.sockets)
    print(f'Serving on {addrs}')

    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'))
```

(continué en la próxima página)

(proviene de la página anterior)

```
    while True:
        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()
    await writer.wait_closed()

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()
    await writer.wait_closed()

    # 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**

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()
```

Distinto en la versión 3.10: Removed the *loop* parameter.

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**
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.

Un objeto Event administra una bandera interna que se puede establecer en *true* con el método *set()* y se restablece en *false* con el método *clear()*. El método *wait()* se bloquea hasta que la bandera se establece en *true*. El flag se establece en *false* inicialmente.

Distinto en la versión 3.10: Removed the *loop* parameter. 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.

Si el evento está configurado, retorna `True` inmediatamente. De lo contrario, bloquea hasta que otra tarea llame a *set()*.

set()
Establece el evento.

Todas las tareas esperando a que el evento se establezca serán activadas inmediatamente.

clear()
Borra (restablece) el evento.

Las tareas que esperan en *wait()* ahora se bloquearán hasta que se vuelva a llamar al método *set()*.

is_set()
Retorna `True` si el evento está establecido.

Condition

class asyncio.**Condition**(*lock=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.

Distinto en la versión 3.10: Removed the *loop* parameter.

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*)
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*.

Distinto en la versión 3.10: Removed the *loop* parameter.

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*)
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.

Distinto en la versión 3.10: Removed the *loop* parameter.

Distinto en la versión 3.9: Adquirir un bloqueo usando `await lock` o `yield from lock` o una declaración `with` (`with await lock`, `with (yield from lock)`) se eliminó . En su lugar, use `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 todos 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*, *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.

Distinto en la versión 3.10: Removed the *loop* parameter.

coroutine asyncio.**create_subprocess_shell**(*cmd*, *stdin=None*, *stdout=None*, *stderr=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*.

Distinto en la versión 3.10: Removed the *loop* parameter.

Nota: Los subprocesos están disponibles para Windows si se utiliza un `ProactorEventLoop`. Consulte *Soporte de subprocesos en Windows* para obtener más detalles.

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 *stding*, *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 subprocesos 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()*;
- the *communicate()* and *wait()* methods don't have a *timeout* parameter: use the *wait_for()* function;
- 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 subprocesos (*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 lanza 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.SIGNTERM` al proceso hijo.

En Windows, la función de la API de Win32 `TerminateProcess()` es llamado para parar a los procesos hijos.

kill()
Mata el proceso hijo.

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 (`SreamReader`) 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: Use the *communicate()* method rather than *process.stdin.write()*, *await process.stdout.read()* or *await process.stderr.read()*. This avoids deadlocks due to streams pausing reading or writing and blocking the child process.

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 subproceso y la clase `StreamReader` para leer de su salida estándar.

El subproceso 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*)

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 subprocesos de la biblioteca estándar *queue*, el tamaño de la cola siempre es conocido y puede ser retornado llamando el método *qsize()*.

Distinto en la versión 3.10: Removed the *loop* parameter.

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 `fill()` 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()
```

(continué en la próxima página)

(proviene de la página anterior)

```
    # 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

    # 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 esta 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.

Raise a *RuntimeError* if there is no running event loop.

This function can only be called from a coroutine or a callback.

Nuevo en la versión 3.7.

asyncio.**get_event_loop**()
Obtiene bucle de eventos actual.

When called from a coroutine or a callback (e.g. scheduled with call_soon or similar API), this function will always return the running event loop.

If there is no running event loop set, the function will return the result of the `get_event_loop_policy().get_event_loop()` call.

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.

As noted above, consider using the higher-level *asyncio.run()* function, instead of using these lower level functions to manually create and close an event loop.

Nota: In Python versions 3.10.0–3.10.8 and 3.11.0 this function (and other functions which use it implicitly) emitted a *DeprecationWarning* if there was no running event loop, even if the current loop was set on the policy. In Python versions 3.10.9, 3.11.1 and 3.12 they emit a *DeprecationWarning* if there is no running event loop and no current loop is set. In some future Python release this will become an error.

---

asyncio.**set_event_loop**(*loop*)
Set *loop* as the current event loop for the current OS thread.

asyncio.**new_event_loop**()
Create and return a new event loop object.

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.

coroutine loop.**shutdown_default_executor**()
Schedule the closure of the default executor and wait for it to join all of the threads in the *ThreadPoolExecutor*. Once this method has been called, using the default executor with *loop.run_in_executor()* will raise a *RuntimeError*.

Nota: Do not call this method when using *asyncio.run()*, as the latter handles default executor shutdown automatically.

Nuevo en la versión 3.9.

Programación de llamadas de retorno

loop.**call_soon**(*callback*, **args*, *context=None*)
Programa el *callback* (retrollamada) *callback* para que se llame con argumentos *args* en la próxima iteración del ciclo de eventos.

Return an instance of *asyncio.Handle*, which can be used later to cancel the callback.

Llamadas que son ejecutadas en el orden en el que fueron registradas. Cada llamada será ejecutada exactamente una sola vez.

The optional keyword-only *context* argument specifies a custom *contextvars.Context* for the *callback* to run in. Callbacks use the current context when no *context* is provided.

Unlike *call_soon_threadsafe()*, this method is not thread-safe.

loop.**call_soon_threadsafe**(*callback*, **args*, *context=None*)
A thread-safe variant of *call_soon()*. When scheduling callbacks from another thread, this function *must* be used, since *call_soon()* is not thread-safe.

Lanza *RuntimeError* si se llama en un bucle que ha sido cerrado. Esto puede suceder en un hilo secundario cuando la aplicación principal se está apagando.

Vea sección *concurrencia 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*)
Schedule the execution of *coroutine* *coro*. Return a *Task* object.

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: Added the *name* parameter.

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 *transporte* 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*, si se proporciona, habilita Happy Eyeballs para esta conexión. Debe ser un número de punto flotante que represente la cantidad de tiempo en segundos para esperar a que se complete un intento de conexión, antes de comenzar el siguiente intento en paralelo. Este es el «Retraso de intento de conexión» como se define en **RFC 8305**. Un valor predeterminado sensato recomendado por el RFC es `0.25` (250 milisegundos).

- *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.

  Nota: The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport's `close()` method.

- *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 *port*.

- *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` (predefinido).

Distinto en la versión 3.5: Agregado el soporte para SSL/TLS en `ProactorEventLoop`.

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 3.7: Added the *ssl_handshake_timeout* parameter.

Distinto en la versión 3.8: Agregados los parámetros *happy_eyeballs_delay* y *interleave*.

Happy Eyeballs Algorithm: Success with Dual-Stack Hosts. When a server's IPv4 path and protocol are working, but the server's IPv6 path and protocol are not working, a dual-stack client application experiences significant connection delay compared to an IPv4-only client. This is undesirable because it causes the dual-stack client to have a worse user experience. This document specifies requirements for algorithms that reduce this user-visible delay and provides an algorithm.

Para mas información: https://tools.ietf.org/html/rfc6555

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*).

Nota: The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport's *close()* method.

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.

Distinto en la versión 3.7: Added the *ssl_handshake_timeout* parameter. The *path* parameter can now be a *path-like object*.

Creando servidores de red

coroutine loop.**create_server**(*protocol_factory*, *host=None*, *port=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*)

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).
- The *port* parameter can be set to specify which port the server should listen on. If `0` or `None` (the default), a random unused port will be selected (note that if *host* resolves to multiple network interfaces, a different random port will be selected for each interface).
- *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.

Nota: The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server's *close()* method.

- *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.

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.

Distinto en la versión 3.6: Added *ssl_handshake_timeout* and *start_serving* parameters. The socket option `TCP_NODELAY` is set by default for all TCP connections.

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.

Distinto en la versión 3.7: Added the *ssl_handshake_timeout* and *start_serving* parameters. The *path* parameter can now be a `Path` object.

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`.

Nota: The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport's `close()` method.

- *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 3.5.3.

Distinto en la versión 3.7: Added the *ssl_handshake_timeout* parameter.

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.

Create a TLS coder/decoder instance and insert it between the *transport* and the *protocol*. The coder/decoder implements both *transport*-facing protocol and *protocol*-facing transport.

Return the created two-interface instance. After *await*, the *protocol* must stop using the original *transport* and communicate with the returned object only because the coder caches *protocol*-side data and sporadically exchanges extra TLS session packets with *transport*.

In some situations (e.g. when the passed transport is already closing) this may return `None`.

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` (predefinido).

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*)
Stop monitoring the *fd* file descriptor for read availability. Returns `True` if *fd* was previously being monitored for reads.

`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*)
Stop monitoring the *fd* file descriptor for write availability. Returns `True` if *fd* was previously being monitored for writes.

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 con 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()* ara 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*)
Hace arreglos para que *func* sea llamado en el ejecutor especificado.

El argumento *executor* debe ser una instancia de *concurrent.futures.Executor*. El ejecutor predeterminado 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)

if __name__ == '__main__':
    asyncio.run(main())
```

Note that the entry point guard (`if __name__ == '__main__'`) is required for option 3 due to the peculiarities of *multiprocessing*, which is used by *ProcessPoolExecutor*. See *Safe importing of main module*.

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()* ya no configura el max_workers del ejecutor del grupo de subprocesos que crea, sino que lo deja en manos del ejecutor del grupo de subprocesos (*ThreadPoolExecutor*) para establecer el valor por defecto.

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*;
- "task" (opcional): instancia de *asyncio.Task*;
- "handle" (opcional): instancia de *asyncio.Handle*;
- "protocol" (opcional): instancia de *Protocol*;
- "transport" (opcional): instancia de *Transport*;
- "socket" (opcional): instancia de *socket.socket*;
- **"asyncgen" (opcional): Generador asíncrono que causó** la excepción.

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: El nuevo *Python Modo de Desarrollo* ahora también se puede usar para habilitar el modo de 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: On Windows, the default event loop *ProactorEventLoop* supports subprocesses, whereas *SelectorEventLoop* does not. See *Subprocess Support on Windows* for details.

---

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 subproceso 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 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
  - 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 subproceso 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 subproceso 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

class MyPolicy(asyncio.DefaultEventLoopPolicy):
   def new_event_loop(self):
      selector = selectors.SelectSelector()
      return asyncio.SelectorEventLoop(selector)

asyncio.set_event_loop_policy(MyPolicy())
```

Disponibilidad: Unix, Windows.

class asyncio.**ProactorEventLoop**
Un bucle de eventos para Windows que usa «E/S Puertos de Finalización» (IOCP).

Disponibilidad: Windows.

Ver también:

Documentaci´on de MSDN sobre E/S Puertos de Finalización.

class asyncio.**AbstractEventLoop**
Clase base abstracta para bucles de evento compatibles con asyncio.

The *Métodos del bucle de eventos* section lists all methods that an alternative implementation of `AbstractEventLoop` should have defined.

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()
```

(continué en la próxima página)

(proviene de la página anterior)

```
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)
    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*.

Save a reference to the result of this function, to avoid a task disappearing mid-execution.

Distinto en la versión 3.5.1: La función acepta cualquier objeto *awaitable*.

Obsoleto desde la versión 3.10: Se emite una advertencia de desaprobación si *obj* no es un objeto tipo Future y no se especifica *loop* y no hay un bucle de eventos en ejecución.

asyncio.**wrap_future**(*future*, *, *loop=None*)
Envuelve un objeto `concurrent.futures.Future` en un objeto `asyncio.Future`.

Obsoleto desde la versión 3.10: Se emite una advertencia de desaprobación si *future* no es un objeto tipo Future y *loop* no se especifica y no hay un bucle de eventos en ejecución.

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 is an *awaitable* object. Coroutines can await on Future objects until they either have a result or an exception set, or until they are cancelled. A Future can be awaited multiple times and the result is same.

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`.

Obsoleto desde la versión 3.10: Se emite una advertencia de desaprobación si no se especifica *loop* y no hay un bucle de eventos en ejecución.

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(*msg=None*)
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.

Distinto en la versión 3.9: Added the *msg* parameter.

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 ...')

    # 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()`.
- `asyncio.Future.cancel()` acepta un argumento opcional `msg`, pero `concurrent.futures.cancel()` no.

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/transports.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.

If the transport has a buffer for outgoing data, buffered data will be flushed asynchronously. No more data will be received. After all buffered data is flushed, the protocol's *protocol.connection_lost()* method will be called with *None* as its argument. The transport should not be used once it is closed.

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
- subproceso:
  - `'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 semicerradas (*half-closed*).

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 subprocesos

SubprocessTransport.**get_pid**()
Retorna la id del subproceso 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 subproceso no se creó con `stdin = PIPE`
- `1`: transporte de *streaming* para escritura de la salida estándar (*stdout*) o *None* si el subproceso no se creó con `stdout = PIPE`
- *2*‘: transporte de *streaming* para escritura del error estándar (*stderr*) o *None* si el subproceso no se creó con `stderr = PIPE`
- otro *fd*: *None*

SubprocessTransport.**get_returncode**()
Retorna el código de retorno del subproceso como un entero o *None* si no ha retornado aún, lo que es similar al atributo *subprocess.Popen.returncode*.

SubprocessTransport.**kill**()
Mata al subproceso.

En los sistemas POSIX, la función envía SIGKILL al subproceso. En Windows, este método es un alias para *terminate()*.

Ver también *subprocess.Popen.kill()*.

SubprocessTransport.**send_signal**(*signal*)
Envía el número de *señal* al subproceso, como en *subprocess.Popen.send_signal()*.

SubprocessTransport.**terminate**()
Detiene el subproceso.

En los sistemas POSIX, este método envía SIGTERM al subproceso. En Windows, se llama a la función de la API de Windows *TerminateProcess()* para detener el subproceso.

Ver también *subprocess.Popen.terminate()*.

SubprocessTransport.**close**()
Mata al subproceso llamando al método *kill()*.

Si el subproceso 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.

However, *protocol.eof_received()* is called at most once. Once `eof_received()` is called, `data_received()` is not called anymore.

`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 semicerradas (*half-closed*), 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.

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 subprocesos

Las instancias de protocolo de subproceso 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))
        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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
        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.
    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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
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()


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()
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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.
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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 subproceso que se utiliza para obtener la salida de un subproceso y esperar su terminación.

El subproceso es creado por el método `loop.subprocess_exec()`

```
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()
```

(continué en la próxima página)

(proviene de la página anterior)

```
    # 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

An event loop policy is a global object used to get and set the current *event loop*, as well as create new event loops. The default policy can be *replaced* with *built-in alternatives* to use different event loop implementations, or substituted by a *custom policy* that can override these behaviors.

The *policy object* gets and sets a separate event loop per *context*. This is per-thread by default, though custom policies could define *context* differently.

Custom event loop policies can control the behavior of `get_event_loop()`, `set_event_loop()`, and `new_event_loop()`.

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.

Nota: In Python versions 3.10.9, 3.11.1 and 3.12 the *get_event_loop()* method of the default asyncio policy emits a *DeprecationWarning* if there is no running event loop and no current loop is set. In some future Python release this will become an error.

---

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 `créate_subprocess_exec()` y *loop.subprocess_exec()*.

asyncio define la clase base abstracta *AbstractChildWatcher*, qué observadores de subprocesos 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.

No hay sobrecarga notable cuando se gestiona un número grande de procesos secundarios (*O(1)* cada vez que un proceso secundario termina), pero iniciar un hilo por proceso requiere memoria extra.

Este observador es usado por defecto.

Nuevo en la versión 3.8.

class asyncio.**MultiLoopChildWatcher**
Esta implementación registra un gestor de señal en instanciación `SIGCHLD`. Eso puede romper código de terceras partes que instalen un gestor personalizado para la señal `SIGCHLD`.

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.waitpad(-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*.

class asyncio.**PidfdChildWatcher**
Esta implementación sondea los descriptores de archivos de proceso (pidfds) para esperar la terminación del proceso hijo. En ciertos sentidos *PidfdChildWatcher* es una implementación de niño vigilante «Ricitos de oro». No requiere señales o hilos, no interfiere con ningún proceso lanzado fuera del bucle de eventos y escala linealmente con el número de subprocesos lanzados por el bucle de eventos. La principal desventaja es que los pidfds son específicos de Linux y solo funcionan en kernels recientes (5.3+).

Nuevo en la versión 3.9.

Personalizar Políticas

Para implementar una nueva política de bucle de eventos, se recomienda heredar *DefaultEventLoopPolicy* y sobreescribir 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 en múltiples cosas con tiempos de espera.

run()	Crea un loop de eventos, ejecuta una corrutina, cierra el loop.
create_task()	Lanza una Tarea asyncio.
await *sleep()*	Duerme por un número de segundos.
await *gather()*	Programa y espera por cosas concurrentemente.
await *wait_for()*	Ejecuta con un tiempo de expiración.
await *shield()*	Protege de la cancelación.
await *wait()*	Monitorea la completitud.
current_task()	Retorna la Tarea actual.
all_tasks()	Retorna todas las tareas para un loop de eventos.
Task	Objeto Tarea.
to_thread()	Ejecute de forma asincrónica una función en un subproceso del sistema operativo independiente.
run_coroutine_threadsafe()	Programa una corrutina de desde otro hilo del sistema operativo.
for in *as_completed()*	Monitorea por completitud con un loop `for`.

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.

Queue	Una cola FIFO.
PriorityQueue	Una cola de prioridad.
LifoQueue	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 subprocesos y ejecutar comandos de consola.

await *create_subprocess_exec()*	Crea un subproceso.
await *create_subprocess_shell()*	Ejecuta un comando de consola.

Ejemplos

- *Ejecutando un comando de consola*.
- Ver también la documentación de las *APIs de subprocesos*.

Flujos

APIs de alto nivel para trabajar con IO de red.

await *open_connection()*	Establece una conexión TCP.
await *open_unix_connection()*	Establece una conexión de un socket Unix.
await *start_server()*	Lanza un servidor TCP.
await *start_unix_server()*	Lanza un servidor de sockets Unix.
StreamReader	Objeto de alto nivel async/await para recibir datos de red.
StreamWriter	Objeto de alto nivel async/await 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.

`Lock`	Un bloqueo mutex.
`Event`	Un objeto de evento.
`Condition`	Un objeto de condición.
`Semaphore`	Un semáforo.
`BoundedSemaphore`	Un semáforo acotado.

Ejemplos

- *Usando asyncio.Event*.
- Ver también la documentación de las *primitivas de sincronización* de asyncio.

Excepciones

`asyncio.TimeoutError`	Lanzado en tiempos de expiración por funciones como `wait_for()`. Ten en mente que *asyncio.TimeoutError*‘ **no está relacionada** con la excepción predefinida `TimeoutError`.
`asyncio.CancelledError`	Lanzada cuando una Tarea es cancelada. Ver también `Task.cancel()`.

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

`asyncio.get_running_loop()`	La función **preferida** para obtener el bucle de eventos en ejecución.
`asyncio.get_event_loop()`	Get an event loop instance (running or current via the current policy).
`asyncio.set_event_loop()`	Establece el bucle de eventos como actual a través de la política del bucle.
`asyncio.new_event_loop()`	Crea un nuevo bucle de eventos.

Ejemplos

- *Usando asyncio.get_running_loop()*.

Métodos del bucle de eventos

See also the main documentation section about the *Métodos del bucle de eventos*.

Ciclo de vida

loop.run_until_complete()	Ejecuta un Future/Tarea/aguardable (*awaitable*) hasta que se complete.
loop.run_forever()	Ejecuta el bucle de eventos para siempre.
loop.stop()	Detiene el bucle de eventos.
loop.close()	Cierra el bucle de eventos.
loop.is_running()	Retorna `True` si el bucle de eventos se está ejecutando.
loop.is_closed()	Retorna `True` si el bucle de eventos está cerrado.
await *loop.shutdown_asyncgens()*	Cierra generadores asincrónicos.

Depuración

loop.set_debug()	Habilita o deshabilita el modo de depuración.
loop.get_debug()	Obtiene el modo de depuración actual.

Programación de devoluciones de llamada

loop.call_soon()	Invoca una devolución de llamada *soon*.
loop.call_soon_threadsafe()	Una variante segura para subprocesos de *loop.call_soon()*.
loop.call_later()	Invoca una devolución de llamada *después* del tiempo especificado.
loop.call_at()	Invoca una devolución de llamada *en* el tiempo especificado.

Hilo/Grupo de procesos

await *loop.run_in_executor()*	Ejecuta una función de bloqueo vinculada a la CPU o de otro tipo en un ejecutor *concurrent.futures*.
loop.set_default_executor()	Establece el ejecutor predeterminado para *loop.run_in_executor()*.

Tareas y Futures

loop.create_future()	Crea un objeto *Future*.
loop.create_task()	Programa una corrutina como *Task*.
loop.set_task_factory()	Establece una fábrica utilizada por *loop.create_task()* para crear *Tareas*.
loop.get_task_factory()	Obtiene la fábrica *loop.create_task()* que se usa para crear *Tareas*.

DNS

await *loop.getaddrinfo()*	Versión asincrónica de *socket.getaddrinfo()*.
await *loop.getnameinfo()*	Versión asincrónica de *socket.getnameinfo()*.

Redes e IPC

await *loop.create_connection()*	Abre una conexión TCP.
await *loop.create_server()*	Crea un servidor TCP.
await *loop.create_unix_connection()*	Abre una conexión de socket Unix.
await *loop.create_unix_server()*	Crea un servidor socket de Unix.
await *loop.connect_accepted_socket()*	Envuelve un *socket* en un par `(transport, protocol)`.
await *loop.create_datagram_endpoint()*	Abre una conexión de datagramas (UDP).
await *loop.sendfile()*	Envía un archivo a través del transporte.
await *loop.start_tls()*	Actualiza una conexión existente a TLS.
await *loop.connect_read_pipe()*	Envuelve el fin de lectura de *pipe* en un par `(transport, protocol)`.
await *loop.connect_write_pipe()*	Envuelve el fin de escritura de *pipe* en un par `(transport, protocol)`.

Sockets

await *loop.sock_recv()*	Recibe datos de *socket*.
await *loop.sock_recv_into()*	Recibe datos de *socket* en un buffer.
await *loop.sock_sendall()*	Envía datos a *socket*.
await *loop.sock_connect()*	Conecta con *socket*.
await *loop.sock_accept()*	Acepta una conexión *socket*.
await *loop.sock_sendfile()*	Envía un archivo a través de *socket*.
loop.add_reader()	Comienza a monitorear un descriptor de archivo para ver la disponibilidad de lectura.
loop.remove_reader()	Detiene el monitoreo del descriptor de archivo para ver la disponibilidad de lectura.
loop.add_writer()	Comienza a monitorear un descriptor de archivo para ver la disponibilidad de escritura.
loop.remove_writer()	Detiene el monitoreo del descriptor de archivo para ver la disponibilidad de escritura.

Señales Unix

loop.add_signal_handler()	Añade un gestor para *signal*.
loop.remove_signal_handler()	Elimina un gestor para *signal*.

Subprocesos

loop.subprocess_exec()	Genera un subproceso.
loop.subprocess_shell()	Genera un subproceso desde un comando de shell.

Gestor de errores

loop.call_exception_handler()	Invoca al gestor de excepciones.
loop.set_exception_handler()	Establece un nuevo gestor de excepciones.
loop.get_exception_handler()	Obtiene el gestor de excepciones actual.
loop.default_exception_handler()	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:

transport.close()	Cierra el transporte.
transport.is_closing()	Retorna `True` si el transporte está cerrado o se está cerrando.
transport.get_extra_info()	Solicita información sobre el transporte.
transport.set_protocol()	Establece un nuevo protocolo.
transport.get_protocol()	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

transport.is_reading()	Retorna `True` si el transporte está recibiendo.
transport.pause_reading()	Pausa la recepción.
transport.resume_reading()	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

transport.write()	Escribe datos en el transporte.
transport.writelines()	Escribe búferes en el transporte.
transport.can_write_eof()	Retorna *True* si el transporte admite el envío de EOF.
transport.write_eof()	Cierra y envía EOF después de vaciar los datos almacenados en búfer.
transport.abort()	Cierra el transporte inmediatamente.
transport.get_write_buffer_size()	Return the current size of the output buffer.
transport.get_write_buffer_limits()	Retorna los límites superior e inferior para controlar el flujo de escritura.
transport.set_write_buffer_limits()	Establece nuevos límites superior e inferior para el control del flujo de escritura.

Transportes retornados por *loop.create_datagram_endpoint()*:

Transportes de datagramas

transport.sendto()	Envía datos al par remoto.
transport.abort()	Cierra el transporte inmediatamente.

Abstracción de transporte de bajo nivel sobre subprocesos. Retornado por *loop.subprocess_exec()* y *loop.subprocess_shell()*:

Transportes de subprocesos

transport.get_pid()	Retorna el id de proceso del subproceso.
transport.get_pipe_transport()	Retorna el transporte para la *pipe* de comunicación solicitada (*stdin*, *stdout* o *stderr*).
transport.get_returncode()	Retorna el código de retorno del subproceso.
transport.kill()	Mata el subproceso.
transport.send_signal()	Envía una señal al subproceso.
transport.terminate()	Detiene el subproceso.
transport.close()	Mata el subproceso y cierra todas las *pipes*.

Protocolos

Las clases de protocolo pueden implementar los siguientes **métodos de devolución de llamada**:

callback *connection_made()*	Se llama cuando se establece una conexión.
callback *connection_lost()*	Se llama cuando la conexión se pierde o cierra.
callback *pause_writing()*	Se llama cuando el búfer del transporte excede el límite superior.
callback *resume_writing()*	Se llama cuando el búfer del transporte se vacía por debajo del límite inferior.

Protocolos de streaming (TCP, Unix Sockets, Pipes)

callback *data_received()*	Se llama cuando se reciben algunos datos.
callback *eof_received()*	Se llama cuando se recibe un EOF.

Protocolos de streaming en búfer

callback *get_buffer()*	Se llama para asignar un nuevo búfer de recepción.
callback *buffer_updated()*	Se llama cuando el búfer se actualizó con los datos recibidos.
callback *eof_received()*	Se llama cuando se recibe un EOF.

Protocolos de datagramas

callback *datagram_received()*	Se llama cuando se recibe un datagrama.
callback *error_received()*	Se llama cuando una operación de envío o recepción anterior genera un *OSError*.

Protocolos de subprocesos

callback *pipe_data_received()*	Se llama cuando el proceso hijo escribe datos en su *pipe* *stdout* o *stderr*.
callback *pipe_connection_lost()*	Se llama cuando se cierra un *pipe* que se comunica con el proceso hijo.
callback *process_exited()*	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

asyncio.get_event_loop_policy()	Retorna la política actual en todo el proceso.
asyncio.set_event_loop_policy()	Establece una nueva política para todo el proceso.
AbstractEventLoopPolicy	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 *Modo de Desarrollo 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.

Para programar un *callback* desde otro hilo del SO, se debe usar el método *loop.call_soon_threadsafe()*. Ejemplo:

```
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 subprocesos, 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 *Métodos del bucle de eventos* 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)
```

Network logging can block the event loop. It is recommended to use a separate thread for handling logs or use non-blocking IO. For example, see blocking-handlers.

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` — interfaz de red de bajo nivel

Código fuente: Lib/socket.py

Este módulo proporciona acceso a la interfaz BSD *socket*. Está disponible en todos los sistemas Unix modernos, Windows, MacOS y probablemente plataformas adicionales.

Nota: Algunos comportamientos pueden depender de la plataforma, ya que las llamadas se realizan a las API de socket del sistema operativo.

La interfaz de Python es una transcripción sencilla de la llamada al sistema Unix y la interfaz de la biblioteca para sockets al estilo orientado a objetos de Python: la función `socket()` devuelve a *socket object* cuyos métodos implementan las diversas llamadas al sistema de socket. Los tipos de parámetros tienen un nivel algo más alto que en la interfaz C: como con `read()` y `write()` en las operaciones en los archivos Python, la asignación del buffer en las operaciones de recepción es automática y la longitud del buffer está implícita en las operaciones de envío.

Ver también:

Módulo *`socketserver`* Clases que simplifican la escritura de servidores de red.

Módulo *`ssl`* Un contenedor TLS/SSL para objetos de socket.

18.2.1 Familias Socket

Dependiendo del sistema y de las opciones de compilación, este módulo admite varias familias de sockets.

El formato de dirección requerido por un objeto de socket determinado se selecciona automáticamente en función de la familia de direcciones especificada cuando se creó el objeto de socket. Las direcciones de socket se representan de la siguiente manera:

- La dirección de un socket *AF_UNIX* enlazado a un nodo del sistema de archivos es representado como una cadena de caracteres, utilizando la codificación del sistema de archivos y el controlador de errores `'surrogateescape'` (Observar **PEP 383**). Una dirección en el espacio de nombre abstracto de Linux es devuelvo como un *bytes-like object* con un byte inicial nulo; tenga en cuenta que los sockets en este nombre de espacio puede comunicarse con sockets normales del sistema de archivos, así que los programas destinados a correr en Linux podrían necesitar tratar con ambos tipos de direcciones. Se puede pasar un objeto similar a una cadena de caracteres o bytes para cualquier tipo de dirección al pasarlo como argumento.

  Distinto en la versión 3.3: Anteriormente, se suponía que las rutas de socket *AF_UNIX* utilizaban codificación UTF-8.

  Distinto en la versión 3.5: Ahora se acepta la grabación *bytes-like object*.

- Se utiliza un par `(host, port)` para la familia de direcciones *AF_INET*, donde *host* es una cadena que representa un nombre de host en notación de dominio de Internet como `'daring.cwi.nl'` o una dirección IPv4 como `'100.50.200.5'`, y *port* es un número entero.

  – Para direcciones IPv4, se aceptan dos formas especiales en lugar de una dirección de host: `''` representa `INADDR_ANY`, que se utiliza para enlazar a todas las interfaces, y la cadena de caracteres `'<broadcast>'` representa `INADDR_BROADCAST`. Este comportamiento no es compatible con IPv6, por lo tanto, es posible que desee evitarlos sí tiene la intención de admitir IPv6 con sus programas Python.

- Para la familia de direcciones *AF_INET6*, se utiliza una `(host, port, flowinfo, scope_id)` de cuatro tuplas, donde *flowinfo* y *scope_id* representan los miembros `sin6_flowinfo` y `sin6_scope_id` en `struct sockaddr_in6` en C. Para los métodos de los módulos *socket*, *flowinfo* y *scope_id* pueden ser omitidos solo por compatibilidad con versiones anteriores. Sin embargo la omisión de *scope_id* puede causar problemas en la manipulación de direcciones IPv6 con ámbito.

  Distinto en la versión 3.7: Para direcciones de multidifusión (con *scopeid* significativo) *address* puede no contener la parte `%scope` (o `zone id`). Esta información es superflua y puede omitirse de forma segura (recomendado).

- `AF_NETLINK` sockets se representan como pares `(pid, groups)`.

- La compatibilidad con LINUX solo para TIPC está disponible mediante la familia de direcciones `AF_TIPC`. TIPC es un protocolo en red abierto y no basado en IP diseñado para su uso en entornos informáticos agrupados. Las direcciones se representan mediante una tupla y los campos dependen del tipo de dirección. El formulario de tupla general es `(addr_type, v1, v2, v3 [, scope])`, donde:

  – *addr_type* es uno de `TIPC_ADDR_NAMESEQ`, `TIPC_ADDR_NAME`, o `TIPC_ADDR_ID`.

  – *scope* es una de `TIPC_ZONE_SCOPE`, `TIPC_CLUSTER_SCOPE`, y `TIPC_NODE_SCOPE`.

  – Si *addr_type* es `TIPC_ADDR_NAME`, entonces *v1* es el tipo de servidor, *v2* es el identificador de puerto, y *v3* debe ser 0.

    Si *addr_type* es `TIPC_ADDR_NAMESEQ`, entonces *v1* es el tipo de servidor, *v2* es el numero de puerto inferior, y *v3* es el numero de puerto superior.

    Si *addr_type* es `TIPC_ADDR_ID`, *v1* es el nodo, *v2* es la referencia y *v3* debe establecerse en 0.

- Una tupla `(interface, )` es usada para la dirección de familia *AF_CAN*, donde *interface* es una cadena de caracteres representando a un nombre de interfaz de red como `'can0'`. La interfaz de red llamada `''` puede ser usada para recibir paquetes de todas las interfaces de red de esta familia.

  - Protocolo *CAN_ISOTP* requiere una tupla `(interface, rx_addr, tx_addr)` donde ambos tiene parámetros adicionales son enteres largos sin símbolos que representan una identificador CAN (estándar o extendido).
  - Protocolo *CAN_J1939* requiere una tupla `(interface, name, pgn, addr)` donde los parámetros adicionales son números enteros sin signo de 64 bits representando el nombre ECU, los enteros sin signo de 32-bits representan el numero de grupo de parámetros(PGN), y los enteros de 8-bit representan la dirección.

- 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` admite los siguientes protocolos y formatos de dirección:
  - `BTPROTO_L2CAP` acepta `(bdaddr, psm)` donde `bdaddr` es la dirección Bluetooth como una cadena de caracteres y `psm` es un entero.
  - `BTPROTO_RFCOMM` acepta `(bdaddr, channel)` donde `bdaddr` es la dirección Bluetooth como una cadena de caracteres y `channel` es un entero.
  - `BTPROTO_HCI` acepta `(device_id,)` donde `device_id` es un numero entero o una cadena de caracteres con la dirección Bluetooth de la interfaz. (Esto depende de tu OS; NetBSD y DragonFlyBSD supone una dirección Bluetooth mientras todo lo demás espera un entero.)

    Distinto en la versión 3.2: Se ha añadido compatibilidad con NetBSD y DragonFlyBSD.
  - `BTPROTO_SCO` acepta `bdaddr` donde `bdaddr` es un objeto *bytes* que contiene la dirección Bluetooth en un formato cadena. (ex. `b'12:23:34:45:56:67'`) este protocolo no es admitido bajo FreeBSD.

- *AF_ALG* es una interfaz basada en socket sólo Linux para la criptografía del núcleo. Un socket de algoritmo se configura con una tupla de dos a cuatro elementos `(type, name [, feat [, mask]])`, donde:
  - *type* es el tipo de algoritmos como cadenas de caracteres, e.g. `aead`, `hash`, `skcipher` o `rng`.
  - *name* es el nombre del algoritmo y el modo de operación como cadena de caracteres, e.g. `sha256`, `hmac(sha256)`, `cbc(aes)` o `drbg_nopr_ctr_aes256`.
  - *feat* y *mask* son enteros de 32 bits sin signo.

  Availability: Linux 2.6.38, some algorithm types require more recent Kernels.

  Nuevo en la versión 3.6.

- *AF_VSOCK* permite comunicación entre maquinas virtuales y sus hosts. Los sockets están representando como una tupla `(CID, port)` donde el contexto del ID o CID y el puerto son enteros.

  Availability: Linux >= 4.8 QEMU >= 2.8 ESX >= 4.0 ESX Workstation >= 6.5.

  Nuevo en la versión 3.7.

- *AF_PACKET* es una interfaz de bajo nivel directa con los dispositivos de red. Los paquetes están representados por la tupla `(ifname, proto[, pkttype[, hatype[, addr]]])` donde:
  - *ifname* - Cadena que especifica el nombre del dispositivo.
  - *proto* - Un entero en orden de byte de red que especifica el número de protocolo Ethernet.
  - *pkttype* - Entero opcional especificando el tipo de paquete:
    * `PACKET_HOST` (por defecto) - Paquetes diseccionado al local host.
    * `PACKET_BROADCAST` - Paquete de transmisión de la capa física.
    * `PACKET_MULTIHOST` - Paquete enviado a una dirección de multidifusión de capa física.
    * `PACKET_OTHERHOST` - Paquete a otro host que haya sido capturado por un controlador de dispositivo en modo promiscuo.

    * PACKET_OUTGOING - Paquete originalmente desde el local host que se enlaza de nuevo a un conector de paquetes.

  - *hatype* - Entero opcional que especifica el tipo de dirección de hardware ARP.
  - *addr* - Objeto opcional en forma de bytes que especifica la dirección física del hardware, cuya interpretación depende del dispositivo.

    Availability: Linux >= 2.2.

- *AF_QIPCRTR* es una interfaz basada en sockets solo para Linux para comunicarse con servicios que se ejecutan en co-procesadores en plataformas Qualcomm. La familia de direcciones se representa como una tupla `(node, port)` donde el *node* y *port* son enteros no negativos.

  Availability: Linux >= 4.7.

  Nuevo en la versión 3.8.

- `IPPROTO_UDPLITE` es una variante de UDO que te permite especificar que porción del paquete es cubierta con la suma de comprobación. Esto agrega dos opciones al socket que pueden cambiar. `self.setsockopt(IPPROTO_UDPLITE, UDPLITE_SEND_CSCOV, length)` cambiara que parte de los paquetes salientes están cubierta por la suma de comprobación y `self.setsockopt(IPPROTO_UDPLITE, UDPLITE_RECV_CSCOV, length)` filtrara los paquetes que permitirá cubrir una pequeña parte de tu datos. En ambos casos `length` deben estar en `range(8, 2**16, 8)`.

  Tal socket debe construirse como `socket(AF_INET, SOCK_DGRAM, IPPROTO_UDPLITE)` para IPV4 o *socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDPLITE)*‘ para IPV6.

  Availability: Linux >= 2.6.20, FreeBSD >= 10.1-RELEASE

  Nuevo en la versión 3.9.

Si utiliza un nombre de host en la parte *host* de la dirección de socket IPv4/v6, el programa puede mostrar un comportamiento no determinista, ya que Python utiliza la primera dirección devuelta por la resolución DNS. La dirección del socket se resolverá de manera diferente en una dirección IPv4/v6 real, dependiendo de los resultados de la resolución DNS y/o la configuración del host. Para un comportamiento determinista, utilice una dirección numérica en la parte *host*.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised. Errors related to socket or address semantics raise *OSError* or one of its subclasses.

El modo de no bloqueo es compatible a través de *setblocking()*. Se admite una generalización de esto basada en los tiempos de espera a través de *settimeout()*.

18.2.2 Contenido del módulo

El módulo *socket* exporta los siguientes elementos.

Excepciones

exception socket.**error**
Un alias en desuso de *OSError*.

Distinto en la versión 3.3: Siguiendo **PEP 3151**, es clase fue creada como un alias de *OSError*.

exception socket.**herror**
Una subclase de *OSError*, esta excepción se produce para los errores relacionados con la dirección, es decir, para las funciones que utilizan *h_errno* en la API de POSIX C, incluidas *gethostbyname_ex()* y *gethostbyaddr()*. El valor adjunto es un par `(h_errno, string)` que representa un error devuelto por una llamada a la biblioteca. *h_errno* es un valor numérico, mientras que *string* representa la descripción de *h_errno*, devuelta por la función `hstrerror()` C.

Distinto en la versión 3.3: Esta clase fue creada como una subclase de *OSError*.

exception socket.**gaierror**
Una subclase de *OSError*, esta excepción se genera por errores relacionados a la dirección por *getaddrinfo()* y *getnameinfo()*. El valor de acompañamiento es un par `(error, string)` representado un error retornado por una llamada de librería. *string* representa la descripción del *error*, tal como es retornado por la función C `gai_strerror()`. El valor numérico *error* coincide con una de las constantes `EAI_*` definidas en este modulo.

Distinto en la versión 3.3: Esta clase fue creada como una subclase de *OSError*.

exception socket.**timeout**
Un alias obsoleto de *TimeoutError*.

Una subclase de *OSError*, esta excepción se genera cuando ocurre un *timeout* en un socket que ha tenido tiempos de espera habilitados a través de una llamada previa a *settimeout()* (o implícitamente mediante *setdefaulttimeout()*). El valor del acompañamiento es una cadena de caracteres cuyo valor es actualmente siempre "tiempo de espera".

Distinto en la versión 3.3: Esta clase fue creada como una subclase de *OSError*.

Distinto en la versión 3.10: Esta clase se convirtió en un alias de *TimeoutError*.

Constantes

Las constantes AF_* y SOCK_* ahora son colecciones: `AddressFamily` y `SocketKind` *IntEnum*.

Nuevo en la versión 3.4.

socket.**AF_UNIX**
socket.**AF_INET**
socket.**AF_INET6**
Estas constantes representan la familias de direcciones (y protocolos), usados para el primer argumento para `socket()`. Si la constante *AF_UNIX* no esta definida entonces este protocolo no es compatible. Mas constantes podrían estar disponibles dependiendo del sistema.

socket.**SOCK_STREAM**
socket.**SOCK_DGRAM**
socket.**SOCK_RAW**
socket.**SOCK_RDM**
socket.**SOCK_SEQPACKET**
Estas constantes representan los tipos de socket, usadas por el segundo argumento para `socket()`. Mas constante podrían estar disponibles dependiendo del sistema. (Solamente *SOCK_STREAM* y *SOCK_DGRAM* parecen ser útiles en general.)

socket.**SOCK_CLOEXEC**
socket.**SOCK_NONBLOCK**
Estas dos constantes, si se definen, se pueden combinar con los tipos de socket y le permiten establecer algunas banderas atómicamente (evitando así posibles condiciones de carrera y la necesidad de llamadas separadas).

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_*

Muchas constantes de estos formularios, documentadas en la documentación de Unix en sockets y/o el protocolo IP, también se definen en el módulo de socket. Generalmente se utilizan en argumentos de los métodos `setsockopt()` y `getsockopt()` de objetos socket. En la mayoría de los casos, solo se definen los símbolos definidos en los archivos de encabezado Unix; para algunos símbolos, se proporcionan valores predeterminados.

Distinto en la versión 3.6: `SO_DOMAIN`, `SO_PROTOCOL`, `SO_PEERSEC`, `SO_PASSSEC`, `TCP_USER_TIMEOUT`, `TCP_CONGESTION` han sido agregados.

Distinto en la versión 3.6.5: En Windows, `TCP_FASTOPEN`, `TCP_KEEPCNT` aparecen si el tiempo de ejecución de Windows lo admite.

Distinto en la versión 3.7: `TCP_NOTSENT_LOWAT` ha sido agregada.

En Windows, `TCP_KEEPIDLE`, `TCP_KEEPINTVL` aparecen si el tiempo de ejecución de Windows lo admite.

Distinto en la versión 3.10: Se agregó `IP_RECVTOS`. Se agregó `TCP_KEEPALIVE`. En MacOS, esta constante se puede utilizar de la misma forma que `TCP_KEEPIDLE` en Linux.

`socket.`**AF_CAN**
`socket.`**PF_CAN**
SOL_CAN_*
CAN_*

Muchas constantes de estos formularios, documentadas en la documentación de Linux, también se definen en el módulo de socket.

Availability: Linux >= 2.6.25.

Nuevo en la versión 3.3.

`socket.`**CAN_BCM**
CAN_BCM_*

CAN_BCM, en la familia de protocolo CAN, es el protocolo del administrador de difusión (BCM. Las constantes del administrador de difusión, documentada en la documentación de Linux, también esta definidos en el modulo socket.

Availability: Linux >= 2.6.25.

Nota: El indicador `CAN_BCM_CAN_FD_FRAME` esta solamente disponible en Linux >= 4.8.

Nuevo en la versión 3.4.

`socket.`**CAN_RAW_FD_FRAMES**

Habilita la compatibilidad con CAN FD en un socket CAN_RAW. Esta opción está deshabilitada de forma predeterminada. Esto permite que la aplicación envíe tramas CAN y CAN FD; sin embargo, debe aceptar las tramas CAN y CAN FD al leer desde el socket.

Esta constante se documenta en la documentación de Linux.

Availability: Linux >= 3.6.

Nuevo en la versión 3.5.

`socket.`**CAN_RAW_JOIN_FILTERS**

Se une a los filtros CAN aplicados de modo que solo las tramas CAN que coinciden con todos los filtros CAN dados se pasan al espacio del usuario.

Esta constante se documenta en la documentación de Linux.

Availability: Linux >= 4.1.

Nuevo en la versión 3.9.

socket.**CAN_ISOTP**
CAN_ISOTP, en el protocolo de familia CAN, es el protocolo ISO-TP (ISO 15765-2). Constantes ISO-TP, documentadas en la documentación Linux.

Availability: Linux >= 2.6.25.

Nuevo en la versión 3.7.

socket.**CAN_J1939**
CAN_J1939, en el protocolo de familias CAN, es el protocolo SAE J1939. Constantes J1939, documentadas en el documentación Linux.

Availability: Linux >= 5.4.

Nuevo en la versión 3.9.

socket.**AF_PACKET**
socket.**PF_PACKET**
PACKET_*
Muchas constantes de estos formularios, documentadas en la documentación de Linux, también se definen en el módulo de socket.

Availability: Linux >= 2.2.

socket.**AF_RDS**
socket.**PF_RDS**
socket.**SOL_RDS**
RDS_*
Muchas constantes de estos formularios, documentadas en la documentación de Linux, también se definen en el módulo de socket.

Availability: Linux >= 2.6.30.

Nuevo en la versión 3.3.

socket.**SIO_RCVALL**
socket.**SIO_KEEPALIVE_VALS**
socket.**SIO_LOOPBACK_FAST_PATH**
RCVALL_*
Constantes para Windows' WSAIoctl(). Las constantes se utiliza como argumentos al método *ioctl()* de objetos de sockets.

Distinto en la versión 3.6: SIO_LOOPBACK_FAST_PATH ha sido agregado.

TIPC_*
LAS constantes relacionadas con TIPC, que coinciden con las exportadas por la API de socket de C. Consulte la documentación de TIPC para obtener más información.

socket.**AF_ALG**
socket.**SOL_ALG**
ALG_*
Constantes para la criptográfica del Kernel de Linux.

Availability: Linux >= 2.6.38.

Nuevo en la versión 3.6.

socket.**AF_VSOCK**
socket.**IOCTL_VM_SOCKETS_GET_LOCAL_CID**
VMADDR*

SO_VM*
Constantes para la comunicación host/invitado de Linux.

Availability: Linux >= 4.8.

Nuevo en la versión 3.7.

socket.**AF_LINK**
Availability: BSD, macOS.

Nuevo en la versión 3.4.

socket.**has_ipv6**
Esta constante contiene un valor booleano que indica si IPv6 se admite en esta plataforma.

socket.**BDADDR_ANY**
socket.**BDADDR_LOCAL**
Estas son constantes de cadenas que contienen direcciones Bluetooth con significados especiales. Por ejemplo *BDADDR_ANY* son usados para indicar cualquier dirección al especificar el socket vinculante con BTPROTO_RFCOMM.

socket.**HCI_FILTER**
socket.**HCI_TIME_STAMP**
socket.**HCI_DATA_DIR**
Para usar con BTPROTO_HCI. *HCI_FILTER* no esta disponible para NetBSD o DragonFlyBSD. *HCI_TIME_STAMP* y *HCI_DATA_DIR* no esta disponible para FreeBSD, NetBSD, o DragonFlyBSD.

socket.**AF_QIPCRTR**
Constante para el protocolo de router IPC de Qualcomm, que se utiliza para comunicarse con procesadores remotos que brindan servicios.

Availability: Linux >= 4.7.

Funciones

Creación de sockets

Todas las siguientes funciones crean *socket objects*.

class socket.**socket**(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)
Crear un nuevo socket usando la dirección de familia dada, tipo de socket y el numero de protocolo. La dirección de familia debería ser *AF_INET* (por defecto), *AF_INET6*, *AF_UNIX*, *AF_CAN*, *AF_PACKET*, o *AF_RDS*. El tipo de socket debería ser *SOCK_STREAM* (por defecto), *SOCK_DGRAM*, *SOCK_RAW* o quizás una de las otras constantes SOCK_. El numero de protocolo es usualmente cero u omitirse o en el caso donde la familia de dirección es *AF_CAN* el protocolo debería ser uno de CAN_RAW, *CAN_BCM*, *CAN_ISOTP* o *CAN_J1939*.

Si *fileno* esta especificado, el valor de *family*, *type*, y *proto* son detectados automáticamente por el descriptor especificado de archivo. La detección automática se puede anular llamado la función con los argumentos explícitos *family*, *type*, o *proto*. Esto solamente afecta como Python representa e.g. el valor de retorno de *socket.getpeername()* pero no del recurso actual del OS. Diferente a *socket.fromfd()*, *fileno* retornara el mismo socket y no un duplicado. Esto puede ayudar a cerrar un socket desconectado usando *socket.close()*.

El socket recién creado es *non-inheritable*.

Genera un *auditing event* socket.__new__ con los argumentos self, family, type, protocol.

Distinto en la versión 3.3: Se añadió la familia AF_CAN. Se añadió la familia AF_RDS.

Distinto en la versión 3.4: El protocolo CAN_BCM ha sido agregado.

Distinto en la versión 3.4: Los sockets devueltos ahora no son heredables.

Distinto en la versión 3.7: El protocolo CAN_ISOTP ha sido agregado.

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)
```

seguirá creando un socket sin bloqueo en los sistemas operativos que admiten `SOCK_NONBLOCK`, pero `sock.type` se establecerá en `socket.SOCK_STREAM`.

Distinto en la versión 3.9: El protocolo CAN_J1939 ha sido agregado.

Distinto en la versión 3.10: Se agregó el protocolo IPPROTO_MPTCP.

`socket.`**`socketpair`**(*[family[, type[, proto]]]*)

Cree un par de objetos de socket conectados utilizando la familia de direcciones, el tipo de socket y el número de protocolo especificados. La familia de direcciones, el tipo de socket y el número de protocolo son los siguientes para la función `socket()` anterior. La familia predeterminada es *AF_UNIX* si se define en la plataforma; de lo contrario, el valor predeterminado es *AF_INET*.

Los sockets creados recientemente son *non-inheritable*.

Distinto en la versión 3.2: Los objetos de socket devueltos ahora admiten toda la API de socket, en lugar de un subconjunto.

Distinto en la versión 3.4: Los sockets devueltos ahora no son heredables.

Distinto en la versión 3.5: Se ha agregado compatibilidad con Windows.

`socket.`**`create_connection`**(*address[, timeout[, source_address]]*)

Se conecta a un servicio TCP que esté escuchando en Internet *address* (un `(host, port)` de 2 tuplas) y retorna el objeto de socket. Esta es una función de nivel superior que *socket.connect()*: si *host* es un nombre de host no numérico, intentará resolverlo para *AF_INET* y *AF_INET6*, y luego intentará conectarse a todas las direcciones posibles sucesivamente hasta que la conexión se realice correctamente. Esto facilita la escritura de clientes que sean compatibles con IPv4 e IPv6.

Pasando el parámetro opcional *timeout* establece el tiempo de espera dentro de la instancia del socket. Si no es proporcionado *timeout*, la configuración global de tiempo de espera predeterminada retornada por *getdefaulttimeout()* es usada.

Si se suministra, *source_address* debe ser una “”(host, puerto)”” de 2 tuplas para que el socket se enlace como su dirección de origen antes de conectarse. Si el host o el puerto son “” o 0 respectivamente, se utilizará el comportamiento predeterminado del sistema operativo.

Distinto en la versión 3.2: *source_address* ha sido agregado.

`socket.`**`create_server`**(*address*, *, *family=AF_INET*, *backlog=None*, *reuse_port=False*, *dualstack_ipv6=False*)

Función de conveniencia que crea un socket TCP enlazado a *address* (una tupla de 2 `(host, puerto)`) y devuelve el objeto de socket.

family should be either *AF_INET* or *AF_INET6*. *backlog* is the queue size passed to *socket.listen()*; if not specified , a default reasonable value is chosen. *reuse_port* dictates whether to set the `SO_REUSEPORT` socket option.

Si *dualstack_ipv6* es true y la plataforma lo admite el socket podrá aceptar conexiones IPv4 e IPv6, de lo contrario lanzará *ValueError*. Se supone que la mayoría de las plataformas POSIX y Windows admiten esta funcionalidad. Cuando esta funcionalidad está habilitada, la dirección devuelta por *socket.getpeername()* cuando se produce una conexión IPv4 será una dirección IPv6 representada como una dirección IPv4 asignada6. Si *dualstack_ipv6* es false, deshabilitará explícitamente esta funcionalidad en las plataformas que la habilitan de forma predeterminada (por ejemplo, Linux). Este parámetro se puede utilizar junto con *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: En plataformas POSIX la opción del socket `SO_REUSEADDR` está configurado para inmediatamente rehusar los sockets previos que estaban vinculados la misma *address* y permanecer en estado TIME_WAIT.

Nuevo en la versión 3.8.

`socket.`**`has_dualstack_ipv6`**`()`
Retorna `True` si la plataforma admite la creación de un socket TCP que pueda manejar conexiones IPv4 e IPv6.

Nuevo en la versión 3.8.

`socket.`**`fromfd`**(*fd*, *family*, *type*, *proto=0*)
Duplica el descriptor de archivo *fd* (un entero retornado por el método `fileno()` de un objeto archivo) y crea un objeto socket a partir del resultado. La familia de direcciones, el tipo de socket y el número de protocolo son como para la función `socket()` anterior. El descriptor de archivo debe hacer referencia a un socket, pero esto no se comprueba — las operaciones posteriores en el objeto pueden fallar si el descriptor de archivo no es válido. Esta función rara vez se necesita, pero se puede usar para obtener o establecer opciones de socket en un socket pasado a un programa como entrada o salida estándar (como un servidor iniciado por el demonio inet de Unix). Se supone que el socket está en modo de bloqueo.

El socket recién creado es *non-inheritable*.

Distinto en la versión 3.4: Los sockets devueltos ahora no son heredables.

`socket.`**`fromshare`**(*data*)
Cree una instancia de un socket a partir de los datos obtenidos del método *`socket.share()`*. Se supone que el socket está en modo de bloqueo.

Availability: Windows.

Nuevo en la versión 3.3.

`socket.`**`SocketType`**
Este es un tipo de objeto Python que representa el tipo de objeto del socket. Es lo mismo que decir `type(socket(…))`.

Otras funciones

El modulo *`socket`* también ofrece varios servicios de red relacionados:

`socket.`**`close`**(*fd*)
Cierre un descriptor de archivo de socket. Esto es como *`os.close()`*, pero para sockets. En algunas plataformas (la mayoría notable de Windows) *`os.close()`* no funciona para descriptores de archivos de socket.

Nuevo en la versión 3.7.

`socket.`**`getaddrinfo`**(*host*, *port*, *family=0*, *type=0*, *proto=0*, *flags=0*)
Traduce el argumento *host*/*port* dentro de una secuencia de 5 tuplas que contiene todo los argumentos necesarios para crear un socket conectado a ese servicio. *host* es un nombre de dominio, una cadena en representación de una dirección IPV4/IPV6 o `None`. *port* es una nombre de una cadena de servicio como `'http'`, un numero de puerto numérico o `None`. Pasando `None` como el valor del *host* y *port*, pasando `NULL` a la API C subyacente.

Los argumentos *family*, *type* y *proto* se puede especificar opcionalmente para reducir la lista de direcciones retornadas. Pasando cero como un valor para cada uno de estos argumentos se selecciona la gama completa

de resultados. El argumento *flags* puede ser uno o varios de los argumentos `AI_*`, y pueden influenciar como los resultados son computados y devueltos. Por ejemplo, `AI_NUMERICHOST` desactivará la resolución de nombres de dominio y lanzará un error sí *host* es un nombre de dominio.

La función devuelve una lista de 5 tuplas con la siguiente estructura:

```
(family, type, proto, canonname, sockaddr)
```

En estas tuplas, *family*, *type*, *proto* son todos enteros y están destinados a ser pasados por la función `socket()`. *canonname* debe ser una cadena que representa el nombre canónico del *host* si `AI_CANONNAME` es parte de el argumento *flags*; de lo contrario *canonname* estará vacía. *sockaddr* es un tupla describiendo una dirección de socket, cuyo formato depende del devuelto *family* (una `(address, port)` tupla de 2 para *AF_INET*, una `(address, port, flowinfo, scope_id)` una tupla de 4 para una *AF_INET6*), y está destinado a ser pasado a el método *socket.connect()*.

Genera un *auditing event* `socket.getaddrinfo` con los argumentos `host`, `port`, `family`, `type`, `protocol`.

En el ejemplo siguiente se obtiene información de dirección para una conexión TCP hipotética a `example.org` en el puerto 80 (los resultados pueden diferir en el sistema si IPv6 no está habilitado):

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(<AddressFamily.AF_INET6: 10>, <AddressFamily.SOCK_STREAM: 1>,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (<AddressFamily.AF_INET: 2>, <AddressFamily.SOCK_STREAM: 1>,
 6, '', ('93.184.216.34', 80))]
```

Distinto en la versión 3.2: los parámetros ahora se pueden pasar mediante argumentos de palabra clave.

Distinto en la versión 3.7: para direcciones de multidifusión IPv6, la cadena que representa una dirección no contendrá partes `%scope_id`.

`socket.`**`getfqdn`**`(`[*name*]`)`

Retorna un nombre de dominio completo para *name*. Si *name* se omite o está vacío, se interpreta como el host local. Para encontrar el nombre completo, se comprueba el nombre de host retornado por *gethostbyaddr()*, seguido de los alias del host, si están disponibles. Se selecciona el primer nombre que incluye un punto. En caso de que no haya disponible un nombre de dominio completo y se haya proporcionado *name*, se retorna sin cambios. Si *name* estaba vacío o era igual a `'0.0.0.0'`, se retorna el nombre de host de *gethostname()*.

`socket.`**`gethostbyname`**`(`*hostname*`)`

Traduce un nombre de host a un formato de dirección IPV4. La dirección IPV4 es retornada como una cadena, como `'100.50.200.5'`. Si el nombre de host es una dirección IPV4 en sí, se devuelve sin cambios. Observa *gethostbyname_ex()* para una interfaz mas completa. *gethostbyname()* no soporta la resolución de nombres IPV6, y *getaddrinfo()* debe utilizarse en su lugar para compatibilidad con doble pila IPv4/v6.

Genera un evento *auditing* `socket.gethostbyname` con el argumento `hostname`.

`socket.`**`gethostbyname_ex`**`(`*hostname*`)`

Traduce un nombre de host al formato de dirección IPv4, interfaz ampliada. Retorna un triple `(hostname, aliaslist, ipaddrlist)` donde *hostname* es el nombre de host principal del host, *aliaslist* es una lista (posiblemente vacía) de nombres de host alternativos para la misma dirección y *ipaddrlist* es una lista de direcciones IPv4 para la misma interfaz en el mismo host (a menudo, pero no siempre una sola dirección). *gethostbyname_ex()* no es compatible con la resolución de nombres IPv6 y, en su lugar, se debe utilizar *getaddrinfo()* para el soporte de pila dual IPv4 / v6.

Genera un evento *auditing* `socket.gethostbyname` con el argumento `hostname`.

`socket.`**`gethostname`**`()`

Retorna una cadena que contenga el nombre de host de la máquina donde se está ejecutando actualmente el intérprete de Python.

Genera un *auditing event* `socket.gethostname` sin argumentos.

Nota: *gethostname()* no siempre retorna el nombre de dominio completo, usa *getfqdn()* para eso.

socket.**gethostbyaddr**(*ip_address*)
Retorna un triple `(hostname, aliaslist, ipaddrlist)` donde *hostname* es el nombre del host primario respondiendo de la misma *ip_address*, *aliaslist* es una (posiblemente vacía) lista de nombres de hosts alternativa, y *ipaddrlist* es una lista de direcciones IPV4/IPV6 para la misma interfaz y en el mismo host (lo más probable es que contenga solo una dirección). Para encontrar el nombre de dominio completo, use la función *getfqdn()*. *gethostbyaddr()* admite tanto IPv4 como IPv6.

Generar un *auditing event* `socket.gethostbyaddr` con los argumentos `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.

Para las direcciones IPv6, `%scope` se anexa a la parte del host si *sockaddr* contiene *scopeid* significativo. Generalmente esto sucede para las direcciones de multidifusión.

Para mas información sobre *flags* pueden consultar *getnameinfo(3)*.

Plantea un *auditing event* `socket.getnameinfo` con el argumento `sockaddr`.

socket.**getprotobyname**(*protocolname*)
Traduzca un nombre de protocolo de Internet (por ejemplo, `'icmp'`) a una constante adecuada para pasar como tercer argumento (opcional) a la función `socket()`. Esto normalmente sólo es necesario para sockets abiertos en modo «raw» (*SOCK_RAW*); para los modos de conexión normales, el protocolo correcto se elige automáticamente si el protocolo se omite o se pone a cero.

socket.**getservbyname**(*servicename*[, *protocolname*])
Traduzca un nombre de servicio de Internet y un nombre de protocolo a un número de puerto para ese servicio. El nombre de protocolo opcional, si se proporciona, debe ser `'tcp'` o `'udp'`; de lo contrario, cualquier protocolo coincidirá.

Genera un *auditing event* `socket.getservbyname` con los argumentos `servicename`, `protocolname`.

socket.**getservbyport**(*port*[, *protocolname*])
Traduzca un número de puerto de Internet y un nombre de protocolo a un nombre de servicio para ese servicio. El nombre de protocolo opcional, si se proporciona, debe ser `'tcp'` o `'udp'`; de lo contrario, cualquier protocolo coincidirá.

Genera un *auditing event* `socket.getservbyport` con los argumentos `port`, `protocolname`.

socket.**ntohl**(*x*)
Convierta enteros positivos de 32 bits de red a orden de bytes de host. En equipos donde el orden de bytes de host es el mismo que el orden de bytes de red, se trata de un no-op; de lo contrario, realiza una operación de intercambio de 4 bytes.

socket.**ntohs**(*x*)
Convierta enteros positivos de 16 bits de red a orden de bytes de host. En equipos donde el orden de bytes de host es el mismo que el orden de bytes de red, se trata de un no-op; de lo contrario, realiza una operación de intercambio de 2 bytes.

Distinto en la versión 3.10: Lanza *OverflowError* si *x* no cabe en un entero sin signo de 16 bits.

socket.**htonl**(*x*)
Convierta enteros positivos de 32 bits del host al orden de bytes de red. En equipos donde el orden de bytes de host es el mismo que el orden de bytes de red, se trata de un no-op; de lo contrario, realiza una operación de intercambio de 4 bytes.

socket.**htons**(*x*)
Convierta enteros positivos de 16 bits del host al orden de bytes de red. En equipos donde el orden de bytes de host es el mismo que el orden de bytes de red, se trata de un no-op; de lo contrario, realiza una operación de intercambio de 2 bytes.

Distinto en la versión 3.10: Lanza *OverflowError* si *x* no cabe en un entero sin signo de 16 bits.

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 `in_addr`, which is the C type for the 32-bit packed binary this function returns.

Ademas `inet_aton`acepta cadena de caracteres con menos de tres puntos, observar la pagina del manual Unix :manpage:`inet(3)` para mas detalles.

Si la cadena de dirección IPv4 es pasada a esta función es invalido, *OSError* se lanzará. Tenga en cuenta que exactamente lo que es valido depende de la implementación C de `inet_aton()`.

inet_aton() no admite IPV6, y *inet_pton()* deberían utilizarse en su lugar para compatibilidad con doble pilas IPV4/v6.

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 `in_addr`, which is the C type for the 32-bit packed binary data this function takes as an argument.

Si la secuencia de byte pasada a esta función no es exactamente 4 bytes de longitud *OSError* podría generarse *inet_ntoa()* no soporta IPV6, y *inet_ntop()* debe utilizarse en su lugar para compatibilidad con doble pila IPv4 / v6.

Distinto en la versión 3.5: Ahora se acepta la grabación *bytes-like object*.

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 `in_addr` (similar to *inet_aton()*) or `in6_addr`.

Los Valores soportados para *address_family* son actualmente *AF_INET* y *AF_INET6*. Si la cadena de dirección IP *ip_string* no es válida, *OSError* se genera. Tenga en cuenta que exactamente lo que es válido depende tanto del valor de *address_family* como de la implementación subyacente de `inet_pton()`.

Disponibilidad: Unix(Tal vez no todas las plataformas), Windows.

Distinto en la versión 3.4: Se ha añadido compatibilidad con Windows

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 `in_addr` (similar to *inet_ntoa()*) or `in6_addr`.

Los valores soportados para *address_family* actualmente son *AF_INET* y *AF_INET6*. Si los objetos de bytes *packed_ip* no tienen la longitud correcta para la familia de direcciones especificas, *ValueError* podría generarse. *OSError* se genera para errores desde la llamada a *inet_ntop()*.

Disponibilidad: Unix(Tal vez no todas las plataformas), Windows.

Distinto en la versión 3.4: Se ha añadido compatibilidad con Windows

Distinto en la versión 3.5: Ahora se acepta la grabación *bytes-like object*.

socket.**CMSG_LEN**(*length*)
Retorna la longitud total, sin relleno de arrastre, de un elemento de datos auxiliares con datos asociados del *length*. Este valor se puede utilizar a menudo como tamaño de búfer para *recvmsg()* para recibir un solo valor de datos auxiliares, pero **RFC 3542** requiere aplicaciones portables para usar *CMSG_SPACE()* y así incluir espacio para el relleno, incluso cuando el elemento será el último en el búfer. Genera *OverflowError* si *length* está fuera del rango de valores permitido.

Availability: la mayoría de plataformas Unix, posiblemente otras.

Nuevo en la versión 3.3.

socket.**CMSG_SPACE**(*length*)
Retorna el tamaño del buffer necesario por *recvmsg()* para recibir un elemento de datos auxiliares con

datos asociados del *length* dado, junto con cualquier relleno final. El espacio de buffer necesario para recibir múltiples elementos es la suma de los valores de `CMSG_SPACE()` para los datos asociados con la longitudes. Genera `OverflowError` si *length* está fuera del rango de valores permitido.

Tenga en cuenta que algunos sistemas pueden admitir datos auxiliares sin proporcionar esta función. Tenga en cuenta también que establecer el tamaño del búfer utilizando los resultados de esta función puede no limitar con precisión la cantidad de datos auxiliares que se pueden recibir, ya que los datos adicionales pueden caber en el área de relleno.

Availability: la mayoría de plataformas Unix, posiblemente otras.

Nuevo en la versión 3.3.

`socket.`**`getdefaulttimeout`**`()`
Retorna el tiempo de espera por defecto en segundos (flotante) para los objetos de un nuevo socket. Un valor de `None` indicada que los objetos del nuevo socket no tiene un tiempo de espera. Cuando el modulo socket es importado primero, por defecto es `None`.

`socket.`**`setdefaulttimeout`**`(`*timeout*`)`
Selecciona el tiempo de espera por defecto en segundos (flotante) para los objetos nuevos del socket. Cuando el modulo socket es importado primero, el valor por defecto es `None`. Observar `settimeout()` para posible valores y sus respectivos significados.

`socket.`**`sethostname`**`(`*name*`)`
Establece el nombre de host de la maquina en *name*. Esto genera un `OSError` si no tiene suficientes derechos.

Plantea un *auditing event* `socket.sethostname` con el argumento `name`.

Availability: Unix.

Nuevo en la versión 3.3.

`socket.`**`if_nameindex`**`()`
Retorna una lista de tuplas de información de interfaz de red (índice int, cadena de nombre). `OSError` si se produce un error en la llamada del sistema.

Availability: Unix, Windows.

Nuevo en la versión 3.3.

Distinto en la versión 3.8: Se ha agregado compatibilidad con Windows.

Nota: En Windows las interfaces de redes tienen diferentes nombres en diferentes contextos (todos los nombres son ejemplos):

- UUID: `{FB605B73-AAC2-49A6-9A2F-25416AEA0573}`
- nombre: `ethernet_32770`
- nombre amigable: `vEthernet (nat)`
- descripción: `Hyper-V Virtual Ethernet Adapter`

Esta función retorna los nombres del segundo formulario de la lista, en este caso de ejemplo `ethernet_32770`.

`socket.`**`if_nametoindex`**`(`*if_name*`)`
Retorna un número de índice de interfaz de red correspondiente a un nombre de interfaz. `OSError` si no existe ninguna interfaz con el nombre especificado.

Availability: Unix, Windows.

Nuevo en la versión 3.3.

Distinto en la versión 3.8: Se ha agregado compatibilidad con Windows.

Ver también:

“”Interface name” es un nombre como se documenta en `if_nameindex()`.

socket.**if_indextoname**(*if_index*)
Retorna un nombre de interfaz de red correspondiente a un número de índice de interfaz. *OSError* si no existe ninguna interfaz con el índice dado.

Availability: Unix, Windows.

Nuevo en la versión 3.3.

Distinto en la versión 3.8: Se ha agregado compatibilidad con Windows.

Ver también:

""Interface name" es un nombre como se documenta en *if_nameindex()*.

socket.**send_fds**(*sock*, *buffers*, *fds*[, *flags*[, *address*]])
Envíe la lista de descriptores de archivo *fds* a través de un conector *AF_UNIX* *sock*. El parámetro *fds* es una secuencia de descriptores de archivo. Consulte `sendmsg()` para obtener la documentación de estos parámetros.

Availability: Soporte para Unix *sendmsg()* con el mecanismo `SCM_RIGHTS`.

Nuevo en la versión 3.9.

socket.**recv_fds**(*sock*, *bufsize*, *maxfds*[, *flags*])
Reciba descriptores de archivo hasta *maxfds* desde un socket *AF_UNIX* *sock*. Retorna `(msg, list(fds), flags, addr)`. Consulte `recvmsg()` para obtener la documentación de estos parámetros.

Availability: Soporte para Unix *recvmsg()* y el mecanismo `SCM_RIGHTS`.

Nuevo en la versión 3.9.

Nota: Cualquier número entero truncado al final de la lista de descriptores de archivo.

18.2.3 Objetos Socket

Los objetos socket tienen los siguientes métodos. Excepto para *makefile()*, esto corresponde al sistema de llamadas Unix para sockets.

Distinto en la versión 3.2: El soporte para el protocolo *context manager* ha sido agregado. Salir del gestor de contexto es equivalente para el llamado *close()*.

socket.**accept**()
Acepta una conexión. El socket debe estar vinculado a una dirección y estar escuchando las conexiones. El valor de retorno es el par `(conn, address)` cuando *conn* es un *new* objeto socket usado para enviar y recibir información en la conexión, y *address* es la dirección vinculada al socket en el extremo de la conexión.

El socket recién creado es *non-inheritable*.

Distinto en la versión 3.4: El socket ahora no es heredable.

Distinto en la versión 3.5: Si se interrumpe la llamada del sistema y el controlador de señal no genera una excepción, el método ahora vuelve a intentar la llamada del sistema en lugar de generar una excepción *InterruptedError* (consulte **PEP 475** para la lógica).

socket.**bind**(*address*)
Enlaza el socket a *address*. El socket no debe estar ya unido. (El formato de *address* depende de la familia de direcciones, consulte más arriba).

Genera un *auditing event* `socket.getaddrinfo` con los argumentos `host`, `port`, `family`, `type`, `protocol`.

socket.**close**()
Marca el socket cerrado. El recurso del sistema subyacente (ej. un descriptor de archivo) también se cierra cuando todos los objetos de archivo de *makefile()* están cerrados. Una vez que eso suceda, todas las

operaciones futuras en el objeto socket fallarán. El extremo remoto no recibirá más datos (después de que se vacíen los datos en cola).

Los sockets se cierran automáticamente cuando se recogen basura, pero se recomienda `cerrarlos()` explícitamente, o usar una instrucción `with` alrededor de ellos.

Distinto en la versión 3.6: *OSError* ahora se lanza si se produce un error cuando se realiza la llamada `close()` subyacente.

Nota: *close()* libera el recurso asociado a una conexión, pero no necesariamente cierra la conexión inmediatamente. Si desea cerrar la conexión a tiempo, llame a *shutdown()* antes de *close()*.

`socket.`**`connect`**`(`*address*`)`
Conectar a un socket remoto en *address*. (El formato de *address* depende de la familia de direcciones — ver arriba.)

Si la conexión es interrumpida por una señal, el método espera hasta que se complete la conexión, o lanza un *TimeoutError* en el tiempo de espera, si el manejador de señales no lanza una excepción y el socket se bloquea o tiene un tiempo de espera. Para sockets sin bloqueo, el método lanza una excepción *InterruptedError* si la conexión es interrumpida por una señal (o la excepción lanzada por el manejador de señales).

Genera un *auditing event* `socket.connect` con los argumentos `self`, `address`.

Distinto en la versión 3.5: El método ahora espera hasta que se completa la conexión en lugar de generar una excepción *InterruptedError* si la conexión se interrumpe por una señal, el controlador de señal no genera una excepción y el socket está bloqueando o tiene un tiempo de espera (consulte el **PEP 475** para la razón de ser).

`socket.`**`connect_ex`**`(`*address*`)`
Similar a *connect(address)*', pero retorna un indicador de error en lugar de generar una excepción para los errores retornados por la llamada de nivel C `connect()` (otros problemas, como "host no encontrado", aún pueden generar excepciones). El indicador de error es `0` si la operación tuvo éxito, caso contrario es el valor de la variable `errno`. Esto es útil para admitir, por ejemplo, conexiones asincrónicas.

Genera un *auditing event* `socket.connect` con los argumentos `self`, `address`.

`socket.`**`detach`**`()`
Coloque el objeto de socket en estado cerrado sin cerrar realmente el descriptor de archivo subyacente. Se devuelve el descriptor de archivo y se puede reutilizar para otros fines.

Nuevo en la versión 3.2.

`socket.`**`dup`**`()`
Duplica el socket.

El socket recién creado es *non-inheritable*.

Distinto en la versión 3.4: El socket ahora no es heredable.

`socket.`**`fileno`**`()`
Retorna un archivo descriptor del socket (un entero pequeño), o -1 si falla. Esto es útil con *select.select()*.

En Windows el pequeño entero retornado por este método no puede ser usado donde un descriptor de un archivo pueda ser usado (como una *os.fdopen()*). Unix no tiene esta limitación.

`socket.`**`get_inheritable`**`()`
Obtiene el *inheritable flag* del descriptor del archivo del socket o el controlador del socket: `True` si el socket puede ser heredada en procesos secundarios, `False` si falla.

Nuevo en la versión 3.4.

`socket.`**`getpeername`**`()`
Retorna la dirección remota a la que esta conectado el socket. Esto es útil para averiguar el número de puerto

de un socket IPv4/v6 remoto, por ejemplo. (El formato de la dirección devuelta depende de la familia de direcciones, consulte más arriba). En algunos sistemas, esta función no es compatible.

socket.**getsockname**()
Retorna la dirección del propio socket. Esto es útil para descubrir el numero de puerto de un socket IPv4/IPv6, por ejemplo. (El formato de la dirección devuelta depende de la familia de direcciones, consulte más arriba).

socket.**getsockopt**(*level*, *optname*[, *buflen*])
Retorna el valor de la opción de socket dada (consulte la página de comando man de Unix *getsockopt(2)*). Las constantes simbólicas necesarias (SO_* etc.) se definen en este módulo. Si *buflen* está ausente, se asume una opción de entero y la función retorna su valor entero. Si *buflen* está presente, especifica la longitud máxima del búfer utilizado para recibir la opción y este búfer se devuelve como un objeto bytes. Depende del autor de la llamada descodificar el contenido del búfer (consulte el módulo integrado opcional *struct* para obtener una manera de decodificar las estructuras C codificadas como cadenas de bytes).

socket.**getblocking**()
Retorna `True` si el socket está en modo de bloqueo, `False` si está en sin bloqueo.

Esto es equivalente a comprobar `socket.gettimeout() == 0`.

Nuevo en la versión 3.7.

socket.**gettimeout**()
Retorna el tiempo de espera en segundos (flotante) asociado con las operaciones del socket, o `None` si el tiempo de espera no es seleccionado. Esto refleja la ultima llamada al *setblocking()* o *settimeout()*.

socket.**ioctl**(*control*, *option*)

plataforma Windows

El método *ioctl()* es una interfaz limitada para el sistema de interfaces WSAIoctl. Por favor refiérase a Win32 documentation para mas información.

En otras plataformas, las funciones genéricas *fcntl.fcntl()* y *fcntl.ioctl()* podrían ser usadas; ellas aceptan un objeto socket como su primer argumento.

Actualmente solo el siguiente control de códigos está soportados: SIO_RCVALL, SIO_KEEPALIVE_VALS, y SIO_LOOPBACK_FAST_PATH.

Distinto en la versión 3.6: SIO_LOOPBACK_FAST_PATH ha sido agregado.

socket.**listen**([*backlog*])
Habilita un servidor para aceptar conexiones. Si *backlog* es especifico, debe ser al menos 0 (si es menor, se establece en 0); especifica el número de conexiones no aceptadas que permitirá el sistema antes de rechazar nuevas conexiones. Si no se especifica, se elige un valor razonable predeterminado.

Distinto en la versión 3.5: El parámetro *backlog* ahora es opcional.

socket.**makefile**(*mode='r'*, *buffering=None*, ***, *encoding=None*, *errors=None*, *newline=None*)
Retorna un *file object* asociado con el socket. El tipo exacto retornado depende de los argumentos dados a *makefile()*. Estos argumentos se interpretan de la misma forma que la función *open()*, excepto que los únicos valores de *mode* admitidos son `'r'` (default), `'w'` and `'b'`.

El socket debe estar en modo de bloqueo; puede tener un tiempo de espera, pero el búfer interno del objeto de archivo puede terminar en un estado incoherente si se produce un tiempo de espera.

Cerrar el objeto de archivo devuelto por *makefile()* no cerrará el socket original a menos que se hayan cerrado todos los demás objetos de archivo y *socket.close()* se haya llamado al objeto socket.

Nota: En Windows, el objeto similar a un archivo creado por *makefile()* no se puede utilizar cuando se espera un objeto de archivo con un descriptor de archivo, como los argumentos de secuencia de *subprocess.Popen()*.

---

socket.**recv**(*bufsize*[, *flags*])
Recibir datos del socket. El valor devuelto es un objeto bytes que representa los datos recibidos. *bufsize* espe-

cifica la cantidad máxima de datos que se recibirán a la vez. Consulte la página del manual de Unix *recv(2)* para conocer el significado del argumento opcional *flags*; por defecto es cero.

Nota: Para una mejor coincidencia con las realidades de hardware y red, el valor de *bufsize* debe ser una potencia relativamente pequeña de 2, por ejemplo, 4096.

Distinto en la versión 3.5: Si se interrumpe la llamada del sistema y el controlador de señal no genera una excepción, el método ahora vuelve a intentar la llamada del sistema en lugar de generar una excepción *InterruptedError* (consulte **PEP 475** para la lógica).

socket.**recvfrom**(*bufsize*[, *flags*])

Recibe datos desde el socket. El valor de retorno es un par `(bytes, address)` donde *bytes* es un objeto de bytes que representa los datos recibidos y *address* es la dirección de el socket enviando los datos. Observar la pagina del manual Unix *recv(2)* para el significado del argumento opcional *flags*; por defecto es cero. (El formato de *address* depende de la familia de direcciones, consulte más arriba).

Distinto en la versión 3.5: Si se interrumpe la llamada del sistema y el controlador de señal no genera una excepción, el método ahora vuelve a intentar la llamada del sistema en lugar de generar una excepción *InterruptedError* (consulte **PEP 475** para la lógica).

Distinto en la versión 3.7: Para direcciones IPv6 de multidifusión, el primer elemento de *address* ya no contiene la parte `%scope_id`. Para obtener la dirección IPV6 completa, use *getnameinfo()*.

socket.**recvmsg**(*bufsize*[, *ancbufsize*[, *flags*]])

Reciba datos normales (hasta *bufsize* bytes) y datos auxiliares del socket. El argumento *ancbufsize* establece el tamaño en bytes del búfer interno utilizado para recibir los datos auxiliares; el valor predeterminado es 0, lo que significa que no se recibirán datos auxiliares. Los tamaños de búfer adecuados para los datos auxiliares se pueden calcular mediante *CMSG_SPACE()* o *CMSG_LEN()*, y los elementos que no caben en el búfer pueden truncarse o descartarse. El valor predeterminado del argumento *flags* es 0 y tiene el mismo significado que para *recv()*.

El valor de retorno es una tupla de 4: `(data, ancdata, msg_flags, address)`. El valor *data* es un objeto *bytes* que contiene los datos no auxiliares recibidos. El valor *ancdata* es una lista de cero o mas tuplas `(cmsg_level, cmsg_type, cmsg_data)` representado los datos auxiliares (control de mensajes) recibidos: *cmsg_level* y *cmsg_type* son enteros especificando el nivel de protocolo y tipo específico de protocolo respectivamente, y *cmsg_data* es un objeto *bytes* sosteniendo los datos asociados. El valor *msg_flags* es el OR bit a bit de varios indicadores que indican condiciones en el mensaje recibido; consulte la documentación de su sistema para obtener más detalles. Si la toma de recepción no está conectada, *address* es la dirección de el socket enviado, si está disponible; de lo contrario, su valor no se especifica.

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.

Algunos sistemas no indican la longitud truncada de los elementos de datos auxiliares que solo se han recibido parcialmente. Si un elemento parece extenderse más allá del final del búfer, *recvmsg()* emitirá un *RuntimeWarning*, y devolverá la parte de él que está dentro del búfer siempre que no se haya truncado antes del inicio de sus datos asociados.

En sistemas donde soporta el mecanismo `SCM_RIGHTS`, la siguiente función recibirá un descriptor de archivos *maxfds*, devolviendo el mensaje de datos y un lista que contiene los descriptores (mientras se ignoran las condiciones inesperadas, como la recepción de mensajes de control no relacionados). Ver también *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))
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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: la mayoría de plataformas Unix, posiblemente otras.

Nuevo en la versión 3.3.

Distinto en la versión 3.5: Si se interrumpe la llamada del sistema y el controlador de señal no genera una excepción, el método ahora vuelve a intentar la llamada del sistema en lugar de generar una excepción *InterruptedError* (consulte **PEP 475** para la lógica).

socket.**recvmsg_into**(*buffers*[, *ancbufsize*[, *flags*]])

Recibir datos normales y datos auxiliares desde el socket, comportándose como *recvmsg()* lo haría, pero dispersar los datos no auxiliares en una serie de buffers en lugar de devolver un nuevo objeto bytes. El argumento *buffers* debe ser un iterable de objetos que exportan buffers de escritura (por ejemplo, objetos *bytearray*); estos se llenarán con fragmentos sucesivos de los datos no auxiliares hasta que se hayan escrito todos o no haya más buffers. El sistema operativo puede establecer un límite (*sysconf()* valor SC_IOV_MAX) en el número de buffers que se pueden utilizar. Los argumentos *ancbufsize* y *flags* tienen el mismo significado que para *recvmsg()*.

El valor de retorno es tupla de 4: `(nbytes, ancdata, msg_flags, address)`, donde *nbytes* es el numero total de bytes de datos no auxiliares escrito dentro de los bufetes, y *ancdata*, *msg_flags* y *address* son lo mismo que para *recvmsg()*.

Ejemplo:

```
>>> 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: la mayoría de plataformas Unix, posiblemente otras.

Nuevo en la versión 3.3.

socket.**recvfrom_into**(*buffer*[, *nbytes*[, *flags*]])

Reciba datos del socket, escribiéndolo en *buffer* en lugar de crear una nueva cadena de bytes. El valor devuelto es un par `(nbytes, address)` donde *nbytes* es el número de bytes recibidos y *address* es la dirección del socket que envía los datos. Consulte la página del manual de Unix *recv(2)* para conocer el significado del argumento opcional *flags*; por defecto es cero. (El formato de *address* depende de la familia de direcciones — ver arriba.)

socket.**recv_into**(*buffer*[, *nbytes*[, *flags*]])

Recibe hasta *nbytes* bytes desde el socket, almacenado los datos en un búfer en lugar de crear una nueva cadena de bytes. Si *nbytes* no esta especificado (o 0), recibir hasta el tamaño disponible en el búfer dado. Retorna el número de bytes recibidos. Ver la página del manual de Unix *recv(2)* para el significado del argumento opcional *flags*; por defecto es cero.

socket.**send**(*bytes*[, *flags*])

Enviar datos al socket. El socket debe estar conectado a un socket remoto. El argumento opcional *flags* tiene el mismo significado que para *recv()* arriba. Retorna el número de bytes enviados. Las aplicaciones son responsables de comprobar que se han enviado todos los datos; si sólo se transmitieron algunos de los datos,

la aplicación debe intentar la entrega de los datos restantes. Para obtener más información sobre este tema, consulte socket-howto.

Distinto en la versión 3.5: Si se interrumpe la llamada del sistema y el controlador de señal no genera una excepción, el método ahora vuelve a intentar la llamada del sistema en lugar de generar una excepción `InterruptedError` (consulte **PEP 475** para la lógica).

`socket.`**`sendall`**(*bytes*[, *flags*])
Enviar datos al socket. El socket debe estar conectado a un socket remoto. El argumento opcional *flags* tiene el mismo significado que para `recv()` arriba. A diferencia de `send()`, este método continúa enviando datos desde *bytes* hasta que se han enviado todos los datos o se produce un error. `Ninguno` se devuelve en caso de éxito. Por error, se genera una excepción y no hay forma de determinar cuántos datos, si los hay, se enviaron correctamente.

Distinto en la versión 3.5: The socket timeout is no longer 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: Si se interrumpe la llamada del sistema y el controlador de señal no genera una excepción, el método ahora vuelve a intentar la llamada del sistema en lugar de generar una excepción `InterruptedError` (consulte **PEP 475** para la lógica).

`socket.`**`sendto`**(*bytes*, *address*)
`socket.`**`sendto`**(*bytes*, *flags*, *address*)
Enviar datos al socket. El socket no debe estar conectado a un socket remoto, ya que el socket de destino se especifica mediante *address*. El argumento opcional *flags* tiene el mismo significado que para `recv()` arriba. Devolver el número de bytes enviados. (El formato de *address* depende de la familia de direcciones — ver arriba.)

Genera un *auditing event* `socket.sendto` con los argumentos `self`, `address`.

Distinto en la versión 3.5: Si se interrumpe la llamada del sistema y el controlador de señal no genera una excepción, el método ahora vuelve a intentar la llamada del sistema en lugar de generar una excepción `InterruptedError` (consulte **PEP 475** para la lógica).

`socket.`**`sendmsg`**(*buffers*[, *ancdata*[, *flags*[, *address*]]])
Enviar datos normales y auxiliares al socket, recopilar los datos no auxiliares de una serie de buffers y concatenando en un único mensaje. El argumento *buffers* especifica los datos no auxiliares como un iterable de *bytes-como objetos* (por ejemplo: objetos `bytes`); el sistema operativo puede establecer un límite (`os.sysconf()` valor `SC_IOV_MAX`) en el número de buffers que se pueden utilizar. El argumento *ancdata* especifica los datos auxiliares (mensajes de control) como un iterable de cero o más tuplas `(cmsg_level, cmsg_type, cmsg_data)`, donde *cmsg_level* y *cmsg_type* son enteros que especifican el nivel de protocolo y el tipo específico del protocolo respectivamente, y *cmsg_data* es un objeto similar a bytes que contiene los datos asociados. Tenga en cuenta que algunos sistemas (en particular, sistemas sin `CMSG_SPACE()`) pueden admitir el envío de solo un mensaje de control por llamada. El valor predeterminado del argumento *flags* es 0 y tiene el mismo significado que para `send()`. Si se proporciona *address* y no `None`, establece una dirección de destino para el mensaje. El valor devuelto es el número de bytes de datos no auxiliares enviados.

La siguiente función envía la lista de descriptores de archivos *fds* sobre un socket `AF_UNIX`, estos sistemas pueden soportar la mecánica `SCM_RIGHTS`. Observar también `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: la mayoría de plataformas Unix, posiblemente otras.

Genera un *auditing event* `socket.sendmsg` con los argumentos `self`, `address`.

Nuevo en la versión 3.3.

Distinto en la versión 3.5: Si se interrumpe la llamada del sistema y el controlador de señal no genera una excepción, el método ahora vuelve a intentar la llamada del sistema en lugar de generar una excepción `InterruptedError` (consulte **PEP 475** para la lógica).

socket.**sendmsg_afalg**([*msg*], *, *op*[, *iv*[, *assoclen*[, *flags*]]])
Versión especializada de *sendmsg()* para el socket *AF_ALG*. Modo de ajuste, IV, longitud de datos asociados a AEAD y banderas para el socket *AF_ALG*.

Availability: Linux >= 2.6.38.

Nuevo en la versión 3.6.

socket.**sendfile**(*file*, *offset=0*, *count=None*)
Enviar un archivo hasta que se alcance EOF mediante el uso de alto rendimiento *os.sendfile* y devolver el número total de bytes que se enviaron. *file* debe ser un objeto de archivo normal abierto en modo binario. Si *os.sendfile* no está disponible (por ejemplo, Windows) o *file* no es un archivo normal *send()* se utilizará en su lugar. *offset* indica desde dónde empezar a leer el archivo. Si se especifica, *count* es el número total de bytes para transmitir en lugar de enviar el archivo hasta que se alcance EOF. La posición del archivo se actualiza a la vuelta o también en caso de error en cuyo caso *file.tell()* se puede utilizar para averiguar el número de bytes que se enviaron. El socket debe ser de tipo *SOCK_STREAM* No se admiten sockets sin bloqueo.

Nuevo en la versión 3.5.

socket.**set_inheritable**(*inheritable*)
Selecciona el *inheritable flag* descriptor del archivo del socket o el controlador del socket.

Nuevo en la versión 3.4.

socket.**setblocking**(*flag*)
Establecer el modo de bloqueo o no bloqueo del socket: si *flag* es false, el socket se establece en modo sin bloqueo, de lo contrario en modo de bloqueo.

El método es una abreviatura para ciertas llamadas *settimeout()*:

- `sock.setblocking(True)` es equivalente a `sock.settimeout(None)`
- `sock.setblocking(False)` es equivalente a `sock.settimeout(0.0)`

Distinto en la versión 3.7: El método ya no aplica la bandera *SOCK_NONBLOCK* en *socket.type*.

socket.**settimeout**(*value*)
Establece un tiempo de espera para bloquear las operaciones de socket. El argumento *value* puede ser un número de punto flotante no negativo que exprese segundos, o `None`. Si se da un valor distinto de cero, las operaciones subsiguientes de socket lanzarán una excepción *timeout* si el período de tiempo de espera *value* ha transcurrido antes de que se complete la operación. Si se da cero, el socket se pone en modo sin bloqueo. Si se da `None`, el enchufe se pone en modo de bloqueo.

Para obtener más información, consulte las notas *notas sobre los tiempos de espera del socket*.

Distinto en la versión 3.7: El método ya no cambia la bandera *SOCK_NONBLOCK* en *socket.type*.

socket.**setsockopt**(*level*, *optname*, *value:* int)

socket.**setsockopt**(*level*, *optname*, *value: buffer*)

socket.**setsockopt**(*level*, *optname*, *None*, *optlen: int*)
Establece el valor de la opción de socket dada (consulte la página de manual de Unix *setsockopt(2)*). Las constantes simbólicas necesarias se definen en el modulo *socket* (`SO_*` etc.). El valor puede ser un entero, `None` o un *bytes-like object* representan un buffer. En el último caso, depende de la persona que llama asegurarse de que la cadena de bytes contenga los bits adecuados (consulte el módulo integrado opcional *struct* para una forma de codificar estructuras C como cadenas de bytes). Cuando *value* se establece en `None`, el argumento *optlen* es requerido. Esto es equivalente a llamar a una función C `setsockopt()` con `optval=NULL` y `optlen=optlen`.

Distinto en la versión 3.5: Ahora se acepta la grabación *bytes-like object*.

Distinto en la versión 3.6: setsockopt(level, optname, None, optlen: int) form added.

socket.**shutdown**(*how*)
Apague una o ambas mitades de la conexión. Si *how* es `SHUT_RD`, más recibe no se permiten. Si *how* es `SHUT_WR`, mas recibe no se permiten. Si *how* es `SHUT_RDWR`, más recibe no se permiten.

socket.**share**(*process_id*)
Duplica un socket y lo prepara para compartirlo con el proceso de destino. El proceso de destino debe estar provisto de *process_id*. el objeto de bytes resultante luego se puede pasar al proceso de destino usando alguna forma de comunicación entre procesos y el socket se puede recrear allí usando *fromshare()*. Una vez que se ha llamado a este método, es seguro cerrar el socket ya que el sistema operativo ya lo ha duplicado para el proceso de destino.

Availability: Windows.

Nuevo en la versión 3.3.

Tenga en cuenta que no hay métodos `read()` o `write()`; use *recv()* y *send()* sin el argumento *flags* en su lugar.

Los objetos de socket también tienen estos atributos (de solo lectura) que corresponden a los valores dados al constructor *socket*.

socket.**family**
La familia Socket.

socket.**type**
El tipo de socket.

socket.**proto**
The socket protocol.

18.2.4 Notas sobre los tiempos de espera del socket

Un objeto de socket puede estar en uno de los tres modos: bloqueo, no bloqueo o tiempo de espera. Los sockets se crean de forma predeterminada siempre en modo de bloqueo, pero esto se puede cambiar llamando a *setdefaulttimeout()*.

- En el modo *bloqueo*, las operaciones se bloquean hasta que se completan o el sistema devuelve un error (como tiempo de espera de conexión agotado).
- In *non-blocking mode*, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately: functions from the *select* module can be used to know when and whether a socket is available for reading or writing.
- En *timeout mode*, las operaciones fallan si no se puede completan el tiempo de espera especifico para el socket (ellos lanzan una excepción *timeout*) o si el sistema devuelve un error.

Nota: En el nivel del sistema operativo, los sockets en el *timeout mode* se establecen internamente en modo sin bloqueo. Además, los modos de bloqueo y tiempo de espera se comparten entre descriptores de archivo y objetos de socket que hacen referencia al mismo punto de conexión de red. Este detalle de implementación puede tener consecuencias visibles si, por ejemplo, decide utilizar el *fileno()* de un socket.

Tiempos de espera y el método `connect`

La operación *connect()* también está sujeta a la configuración de tiempo de espera, y en general se recomienda llamar a *settimeout()* antes de llamar a *connect()* o pasar un parámetro de tiempo de espera a *create_connection()*. Sin embargo, la pila de red del sistema también puede devolver un error de tiempo de espera de conexión propio independientemente de cualquier configuración de tiempo de espera del socket de Python.

Tiempos de espera y el método `accept`

Si *`getdefaulttimeout()`* no es una *`None`*, los sockets devuelto por el método *`accept()`* heredan ese tiempo de espera. De lo contrario, el comportamiento depende de la configuración de la toma de escucha:

- si los sockets están escuchando en *blocking mode* o en el *timeout mode*, el socket devuelve el *`accept()`* en un *blocking mode*;
- si los sockets están escuchando en *non-blocking mode*, ya sea el socket devuelto por *`accept()`* es un modo de bloqueo o no bloque depende del sistema operativo. Si desea garantizar un comportamiento multiplataforma, se recomienda que se anule manualmente esta configuración.

18.2.5 Ejemplo

Aquí están cuatro programas mínimos usando el protocolo TCP/IP: un servidor que hace eco de todos los datos que reciban de vuelta (Servicio a un solo cliente), y un cliente usando esto. Recuerde que un servidor debe llevar a cabo la secuencia `socket()`, *`bind()`*, *`listen()`*, *`accept()`* (posiblemente repitiendo la *`accept()`* para un servicio mas que un cliente), mientras un cliente solamente necesita la secuencia `socket()`, *`connect()`*. También recuerde que el servidor no *`sendall()`*/*`recv()`* en el socket esta escuchando pero en el nuevo socket devuelto por *`accept()`*.

Los dos primeros ejemplos solo admiten IPv4.

```
# 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))
```

Los dos ejemplos siguientes son idénticos a los dos anteriores, pero admiten IPv4 e IPv6. El lado del servidor escuchará la primera familia de direcciones disponible (debe escuchar ambas en su lugar). En la mayoría de los sistemas listos para IPv6, IPv6 tendrá prioridad y es posible que el servidor no acepte tráfico IPv4. El lado del cliente intentará conectarse a todas las direcciones devueltas como resultado de la resolución de nombres y enviará el tráfico al primero conectado correctamente.

```
# Echo server program
import socket
import sys

```

(continué en la próxima página)

(proviene de la página anterior)

```
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()
        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))
```

El siguiente ejemplo muestra cómo escribir un rastreador de red muy simple con sockets sin procesar en Windows. El ejemplo requiere privilegios de administrador para modificar la interfaz:

```
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 packets
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a packet
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

En el ejemplo siguiente se muestra cómo utilizar la interfaz de socket para comunicarse con una red CAN mediante el protocolo de socket sin procesar. Para utilizar CAN con el protocolo de gestor de difusión en su lugar, abra un socket con:

```
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 *socket.recv()* operations (and their counterparts) on the socket object as usual.

Este último ejemplo puede requerir privilegios especiales:

```
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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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')
```

Ejecutar un ejemplo varias veces con un retraso demasiado pequeño entre ejecuciones, podría dar lugar a este error:

```
OSError: [Errno 98] Address already in use
```

Esto se debe a que la ejecución anterior ha dejado el socket en un estado `TIME_WAIT` y no se puede reutilizar inmediatamente.

Este es una bandera *socket* para establecer, en orden para prevenir esto, `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))
```

el indicador `SO_REUSEADDR` indica al kernel que reutilice un socket local en estado `TIME_WAIT`, sin esperar a que expire su tiempo de espera natural.

Ver también:

Para obtener una introducción a la programación de sockets (en C), consulte los siguientes documentos:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*, por Stuart Sechrest
- *An Advanced 4.3BSD Interprocess Communication Tutorial*, por Samuel J. Leffler et al,

ambos en el manual del programador de Unix, documentos suplementarios 1 (secciones PS1:7 y PS1:8). La plataforma especifica material de referencia para las diversas llamadas al sistema también son una valiosa fuente de información en los detalles de la semántica del socket. Para Unix, referencia a las paginas del manual, para Windows, observa la especificación WinSock (o WinSock 2). Para APIS listas IPV6, los lectores pueden querer referirse al titulado Extensiones básicas de interfaz de socket para IPv6 **RFC 3493** .

18.3 `ssl` —Empaquetador o wrapper TLS/SSL para objetos de tipo socket

Código fuente: Lib/ssl.py

Este módulo proporciona acceso a las funciones de cifrado de Seguridad de la capa de transporte (a menudo conocida como «Capa de sockets seguros») y de autenticación de pares para sockets de red, tanto del lado del cliente como del lado del servidor. Este módulo utiliza la biblioteca OpenSSL. Está disponible en todos los sistemas Unix modernos, Windows, Mac OS X y probablemente en plataformas adicionales, siempre que OpenSSL esté instalado en esa plataforma.

Nota: Algunos comportamientos pueden depender de la plataforma, ya que se realizan llamadas a las API de socket del sistema operativo. La versión instalada de OpenSSL también puede provocar variaciones en el comportamiento. Por ejemplo, TLSv1.3 con OpenSSL versión 1.1.1.

Advertencia: No use este módulo sin leer *Consideraciones de seguridad*. Hacerlo puede generar una falsa sensación de seguridad, ya que la configuración predeterminada del módulo ssl no es necesariamente la adecuada para su aplicación.

Esta sección documenta los objetos y funciones en el módulo `ssl`; para más información general sobre TLS,SSL, y certificados, se recomienda que el lector acuda a la sección «Ver también» al final de la página.

Este módulo proporciona una clase, *`ssl.SSLSocket`*, que se deriva del tipo *`socket.socket`*, y proporciona una envoltura similar a un socket que también encripta y desencripta los datos que pasan por el socket con SSL . Admite métodos adicionales como `getpeercert()`, que recupera el certificado del otro lado de la conexión, y `cipher()`, que recupera el cifrado que se utiliza para la conexión segura.

Para aplicaciones más sofisticadas, la clase *`ssl.SSLContext`* ayuda a administrar la configuración y los certificados, que luego pueden ser heredados por sockets SSL creados a través del método *`SSLContext.wrap_socket()`*.

Distinto en la versión 3.5.3: Actualizado para admitir la vinculación con OpenSSL 1.1.0

Distinto en la versión 3.6: OpenSSL 0.9.8, 1.0.0 y 1.0.1 son obsoletos y no son compatibles. En el futuro, el módulo ssl requerirá al menos OpenSSL 1.0.2 o 1.1.0.

Distinto en la versión 3.10: Se ha implementado **PEP 644**. El módulo ssl requiere OpenSSL 1.1.1 o versiones más recientes.

El uso de constantes y funciones obsoletas genera advertencias de obsolescencia.

18.3.1 Funciones, constantes y excepciones

Creación de sockets

Desde Python 3.2 y 2.7.9, se recomienda utilizar *`SSLContext.wrap_socket()`* de una instancia de *`SSLContext`* para envolver sockets como objetos *`SSLSocket`*. La función utilitaria *`create_default_context()`* retorna un nuevo contexto con ajustes por defecto seguros. La vieja función *`wrap_socket()`* es obsoleta debido a que es ineficiente y que no tiene soporte para la indicación de nombre de servidor (SNI) ni hostname matching.

Ejemplo de socket cliente con contexto por defecto y doble pila IPv4/IPv6:

```
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())
```

Ejemplo de socket cliente con contexto personalizado y 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())
```

Ejemplo de socket servidor escuchando en 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()
        ...
```

Creación de contexto

Una función conveniente ayuda a crear objetos *SSLContext* para propósitos comunes.

ssl.**create_default_context**(*purpose=Purpose.SERVER_AUTH*, *cafile=None*, *capath=None*, *cadata=None*)

Retorna un nuevo objeto *SSLContext* con ajustes por defecto para el *purpose* dado. Los ajustes son elegidos por el módulo *ssl* y generalmente representan un nivel de seguridad mas alto que invocando directamente el constructor de *SSLContext*.

cafile, *capath*, *cadata* representan certificados CA opcionales para confiar en la verificación de certificados, como en *SSLContext.load_verify_locations()*. Si los tres son *None* al mismo tiempo, esta función puede optar por confiar en su lugar en los certificados CA por defecto del sistema.

Las configuraciones son: *PROTOCOL_TLS_CLIENT* o *PROTOCOL_TLS_SERVER*, *OP_NO_SSLv2* y *OP_NO_SSLv3* con conjuntos de cifrado de alto cifrado sin RC4 y sin conjuntos de cifrado no autenticados. Pasando *SERVER_AUTH* como *purpose* establece *verify_mode* a *CERT_REQUIRED* y carga certificados CA (cuando al menos uno de los *cafile*, *capath*, o *cadata*) o usa *SSLContext.load_default_certs()* para cargar certificados de CA predeterminados.

Cuando *keylog_filename* es soportado y la variable de entorno SSLKEYLOGFILE está establecida, *create_default_context()* activa el registro de claves.

Nota: El protocolo, las opciones, el cifrado y otros ajustes pueden cambiar a valores mas restrictivos en cualquier momento sin previa obsolescencia. Los valores representan un equilibrio justo entre compatibilidad y seguridad.

Si su aplicación necesita ajustes específicos, debe crear un *SSLContext* y aplicar los ajustes usted mismo.

Nota: Si encuentra que cuando ciertos clientes o servidores antiguos intentan conectarse con un *SSLContext* creado con esta función obtienen un error indicando *Protocol or cipher suite mismatch*, puede ser que estos sólo soportan SSL3.0 el cual esta función excluye utilizando *OP_NO_SSLv3*. SSL3.0 está ampliamente considerado como completamente roto. Si todavía desea seguir utilizando esta función pero permitir conexiones SSL 3.0, puede volver a activarlas mediante:

```
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 ha sido abandonado de la cadena de cifrado por defecto.

Distinto en la versión 3.6: ChaCha20/Poly1305 ha sido agregado a la cadena de caracteres de cifrado por defecto.

3DES ha sido abandonado de la cadena de caracteres de cifrado por defecto.

Distinto en la versión 3.8: Soporte del registro de claves en SSLKEYLOGFILE ha sido agregado.

Distinto en la versión 3.10: El contexto ahora usa el protocolo *PROTOCOL_TLS_CLIENT* o *PROTOCOL_TLS_SERVER* en lugar del *PROTOCOL_TLS* genérico.

Excepciones

exception ssl.**SSLError**
Se lanza para señalar un error de la implementación de SSL subyacente (actualmente proporcionada por la biblioteca OpenSSL). Esto indica algún problema en la capa de cifrado y autenticación de alto nivel que se superpone a la conexión de red subyacente. Este error es un subtipo de *OSError*. El código de error y el mensaje de las instancias de *SSLError* son proporcionados por la biblioteca OpenSSL.

Distinto en la versión 3.3: *SSLError* era un subtipo de *socket.error*.

library
Una cadena de caracteres mnemotécnica que designa el submódulo de OpenSSL en el que se ha producido el error, como SSL, PEM o X509. El rango de valores posibles depende de la versión de OpenSSL.

Nuevo en la versión 3.3.

reason
Una cadena de caracteres mnemotécnica que designa la razón por la que se produjo el error, por ejemplo CERTIFICATE_VERIFY_FAILED. El rango de valores posibles depende de la versión de OpenSSL.

Nuevo en la versión 3.3.

exception ssl.**SSLZeroReturnError**
Una subclase de *SSLError* lanzada cuando se intenta leer o escribir y la conexión SSL ha sido cerrada limpiamente. Tenga en cuenta que esto no significa que el transporte subyacente (lectura TCP) haya sido cerrado.

Nuevo en la versión 3.3.

exception ssl.**SSLWantReadError**
Una subclase de *SSLError* lanzada por un *socket SSL no bloqueante* cuando se intenta leer o escribir datos, pero mas datos necesitan ser recibidos en el transporte TCP subyacente antes de que la solicitud pueda ser completada.

Nuevo en la versión 3.3.

exception ssl.**SSLWantWriteError**
Una subclase de *SSLError* lanzada por un *socket SSL no bloqueante* cuando se intenta leer o escribir datos, pero mas datos necesitan ser enviados en el transporte TCP subyacente antes de que la solicitud pueda ser completada.

Nuevo en la versión 3.3.

exception ssl.**SSLSyscallError**
Una subclase de *SSLError* lanzada cuando se encuentra un error del sistema mientras se intenta completar una operación en un socket SSL. Por desgracia, no hay una manera fácil de inspeccionar el número errno original.

Nuevo en la versión 3.3.

exception ssl.**SSLEOFError**
Una subclase de *SSLError* lanzada cuando la conexión SSL ha sido cancelada abruptamente. Generalmente, no debería intentar reutilizar el transporte subyacente cuando este error se produce.

Nuevo en la versión 3.3.

exception ssl.**SSLCertVerificationError**
Una subclase de *SSLError* lanzada cuando la validación del certificado ha fallado.

Nuevo en la versión 3.7.

verify_code
Un número de error numérico que indica el error de verificación.

verify_message
Una cadena de caracteres legible del error de verificación.

exception ssl.**CertificateError**
Un alias para *SSLCertVerificationError*.

Distinto en la versión 3.7: La excepción es ahora un alias para *SSLCertVerificationError*.

Generación aleatoria

ssl.**RAND_bytes**(*num*)
Retorna *num* bytes pseudoaleatorios criptográficamente fuertes. Lanza un *SSLError* si el PRNG no a sido sembrado con suficiente datos o si la operación no es soportada por el método RAND actual. *RAND_status()* puede ser usada para verificar el estado de PRNG y *RAND_add()* puede ser usada para sembrar el PRNG.

Para casi todas las aplicaciones *os.urandom()* es preferible.

Léase el artículo Wikipedia, Generador de números pseudoaleatorios criptográficamente seguro (CSPRNG), para obtener los requisitos para un generador criptográficamente seguro.

Nuevo en la versión 3.3.

ssl.**RAND_pseudo_bytes**(*num*)
Retorna (bytes, is_cryptographic): bytes es *num* bytes pseudoaleatorios, is_cryptographic es `True` si los bytes generados son criptográficamente fuertes. Lanza un *SSLError* si la operación no es soportada por el método RAND actual.

Las secuencias de bytes pseudoaleatorios generadas serán únicas si tienen una longitud suficiente, pero no son necesariamente impredecibles. Pueden utilizarse para fines no criptográficos y para ciertos fines en protocolos criptográficos, pero normalmente no para la generación de claves, etc.

Para casi todas las aplicaciones *os.urandom()* es preferible.

Nuevo en la versión 3.3.

Obsoleto desde la versión 3.6: OpenSSL a dejado obsoleta *ssl.RAND_pseudo_bytes()*, utilice *ssl.RAND_bytes()* en su lugar.

ssl.**RAND_status**()
Retorna `True` si el generador de números pseudoaleatorios SSL a sido sembrado con "suficiente" aleatoriedad, y `False` de lo contrario. Puede utilizarse `ssl.RAND_egd()` y *ssl.RAND_add()* para aumentar la aleatoriedad del generador de números pseudoaleatorios.

ssl.**RAND_add**(*bytes*, *entropy*)
Mezcla los *bytes* dados en el generador de números pseudoaleatorios de SSL. El parámetro *entropy* (un flotante) es un límite inferior de la entropía contenida en la cadena de caracteres (por lo que siempre se puede utilizar `0.0`). Véase **RFC 1750** para mas información sobre las fuentes de entropía.

Distinto en la versión 3.5: Ahora se acepta *bytes-like object* modificable.

Gestión de certificados

ssl.**match_hostname**(*cert*, *hostname*)
Verifica que *cert* (en formato decodificado tal y como es retornado por *SSLSocket.getpeercert()*) coincide con el *hostname* dado. Las reglas aplicadas son las de comprobación de la identidad de los servidores HTTPS, como se indica en **RFC 2818**, **RFC 5280** y **RFC 6125**. Además de HTTPS, esta función debería ser adecuada para comprobar la identidad de servidores en varios protocolos basados en SSL como FTPS, IMAPS, POPS y otros.

CertificateError es lanzado en caso de error. En caso de éxito, la función no retorna nada:

```
>>> 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: La función ahora sigue **RFC 6125** sección 6.4.3 y no soporta múltiples caracteres comodín (por ejemplo `*.*.com` o `*a*.example.org`) ni tampoco un carácter comodín dentro de un fragmento de un nombre de dominio internacionalizado (IDN). Etiquetas A de IDN tales como `www*.xn--pthon-kva.org` son todavía soportadas, pero `x*.python.org` ya no corresponde con `xn--tda.python.org`.

Distinto en la versión 3.5: Ahora se admite la coincidencia de direcciones IP cuando están presentes en el campo subjectAltName del certificado.

Distinto en la versión 3.7: La función ya no se utiliza para las conexiones TLS. La coincidencia de hostname es ahora realizada por OpenSSL.

Se permite el carácter comodín cuando es el carácter más a la izquierda y el único en ese segmento. Ya no se admiten comodines parciales como `www*.example.com`.

Obsoleto desde la versión 3.7.

`ssl.`**`cert_time_to_seconds`**(*cert_time*)

Retorna el tiempo en segundos desde la Época, dada la cadena de caracteres `cert_time` que representa la fecha *notBefore* o *notAfter* de un certificado en formato strptime `"%b %d %H:%M:%S %Y %Z"` (C locale).

He aquí un ejemplo:

```
>>> 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
```

Las fechas *notBefore* o *notAfter* deben utilizar GMT (**RFC 5280**).

Distinto en la versión 3.5: Interpreta la hora de entrada como una hora en UTC según lo especificado por la zona horaria "GMT" en la cadena de caracteres de entrada. Anteriormente se utilizaba la zona horaria local. Devuelve un número entero (sin fracciones de segundo en el formato de entrada)

`ssl.`**`get_server_certificate`**(*addr*, *ssl_version=PROTOCOL_TLS_CLIENT*, *ca_certs=None*[, *timeout*])

Dada la dirección `addr` de un servidor protegido con SSL, como un par (*hostname*, *port-number*), obtiene el certificado del servidor, y lo retorna como una cadena de caracteres codificada en PEM. Si se especifica `ssl_version`, utiliza esta versión del protocolo SSL para intentar conectarse al servidor. Si se especifica `ca_certs`, debe ser un archivo que contenga una lista de certificados raíz, con el mismo formato que se utiliza para el mismo parámetro en *`SSLContext.wrap_socket()`*. La llamada intentará validar el certificado del servidor contra ese conjunto de certificados raíz, y fallará si el intento de validación falla.

Distinto en la versión 3.3: Esta función es ahora compatible IPv6.

Distinto en la versión 3.5: La *ssl_version* por defecto se cambia de *`PROTOCOL_SSLv3`* a *`PROTOCOL_TLS`* para una máxima compatibilidad con los servidores modernos.

Distinto en la versión 3.10: Se agregó el argumento *session*.

`ssl.`**`DER_cert_to_PEM_cert`**(*DER_cert_bytes*)

Dado un certificado como blob de bytes codificado en DER, devuelve una versión de cadena de caracteres codificada en PEM del mismo certificado.

ssl.**PEM_cert_to_DER_cert**(*PEM_cert_string*)
Dado un certificado como cadena de caracteres ASCII PEM, devuelve una secuencia de bytes codificada con DER para ese mismo certificado.

ssl.**get_default_verify_paths**()
Retorna una tupla con nombre con las rutas por defecto de cafile y capath de OpenSSL. Las rutas son las mismas que las usadas por *SSLContext.set_default_verify_paths()*. El valor de retorno es una *named tuple* `DefaultVerifyPaths`:

- `cafile` - ruta resuelta a cafile o `None` si el archivo no existe,
- `capath` - ruta resuelta a capath o `None` si el directorio no existe,
- `openssl_cafile_env` - clave de entorno de OpenSSL que apunta a un cafile,
- `openssl_cafile` - camino codificado de forma rígida a un cafile,
- `openssl_capath_env` - clave de entorno de OpenSSL que apunta a un capath,
- `openssl_capath` - camino codificado de forma rígida a un directorio capath

Disponibilidad: LibreSSL ignora las variables de entorno `openssl_cafile_env` y `openssl_capath_env`.

Nuevo en la versión 3.4.

ssl.**enum_certificates**(*store_name*)
Recupera los certificados del almacén de certificados del sistema de Windows. *store_name* puede ser uno de los siguientes: `CA`, `ROOT` o `MY`. Windows también puede proporcionar almacenes de certificados adicionales.

La función devuelve una lista de tuplas (cert_bytes, encoding_type, trust). El encoding_type especifica la codificación de cert_bytes. Es `x509_asn` para datos X.509 ASN.1 o `pkcs_7_asn` para datos PKCS#7 ASN.1. Trust especifica el propósito del certificado como un conjunto de OIDS o exactamente `True` si el certificado es de confianza para todos los propósitos.

Ejemplo:

```
>>> 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)]
```

Disponibilidad: Windows.

Nuevo en la versión 3.4.

ssl.**enum_crls**(*store_name*)
Obtiene CRLs del almacén de certificados del sistema de Windows. *store_name* puede ser uno de los siguientes: `CA`, `ROOT` o `MY`. Windows también puede proporcionar almacenes de certificados adicionales.

La función devuelve una lista de tuplas (cert_bytes, encoding_type, trust). El encoding_type especifica la codificación de cert_bytes. Es `x509_asn` para datos X.509 ASN.1 o `pkcs_7_asn` para datos PKCS#7 ASN.1.

Disponibilidad: 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*)
Toma una instancia `sock` de *socket.socket*, y devuelve una instancia de *ssl.SSLSocket*, un subtipo de *socket.socket*, que envuelve el socket de base en un contexto SSL. `sock` debe ser un socket *SOCK_STREAM*; otros tipos de socket no son compatibles.

Internamente, la función crea un *SSLContext* con un protocolo *ssl_version* y *SSLContext.options* establecido a *cert_reqs*. Si los parámetros *keyfile*, *certfile*, *ca_certs* o *ciphers* son establecidos, entonces los valores son pasados a *SSLContext.load_cert_chain()*, *SSLContext.load_verify_locations()*, y *SSLContext.set_ciphers()*.

Los argumentos *server_side*, *do_handshake_on_connect*, y *supress_ragged_eofs* tienen el mismo significado que *SSLContext.wrap_socket()*.

Obsoleto desde la versión 3.7: Desde Python 3.2 y 2.7.9, se recomienda usar *SSLContext.wrap_socket()* en lugar de *wrap_socket()*. La función de alto nivel tiene limitaciones y crea un socket cliente no seguro sin indicación de nombre de servidor ni hostname matching.

Constantes

Todas las constantes son ahora colecciones *enum.IntEnum* o *enum.IntFlag*.

Nuevo en la versión 3.6.

ssl.**CERT_NONE**
Valor posible para *SSLContext.verify_mode*, o el parámetro `cert_reqs` de *wrap_socket()*. A excepción de *PROTOCOL_TLS_CLIENT*, es el modo por defecto. Con sockets del lado del cliente, se acepta casi cualquier certificado. Errores de validación, como certificado no confiable o caducado, son ignorados y no abortan el handshake TLS/SSL.

En modo servidor, no se solicita ningún certificado al cliente, por lo que el cliente no envía ninguno para la autenticación del certificado del cliente.

Vea la discusión sobre *Consideraciones de seguridad* más abajo.

ssl.**CERT_OPTIONAL**
Valor posible para *SSLContext.verify_mode*, o el parámetro `cert_reqs` de *wrap_socket()*. En modo cliente, *CERT_OPTIONAL* tiene el mismo significado que *CERT_REQUIRED*. Se recomienda usar en su lugar *CERT_REQUIRED* para sockets del lado del cliente.

En el modo servidor, se envía una solicitud de certificado de cliente al cliente. El cliente puede ignorar la solicitud o enviar un certificado para realizar la autenticación de certificado de cliente TLS. Si el cliente opta por enviar un certificado, éste se verifica. Cualquier error de verificación aborta inmediatamente el handshake TLS.

El uso de esta configuración requiere que se pase un conjunto válido de certificados de CA, ya sea a *SSLContext.load_verify_locations()* o como valor del parámetro `ca_certs` de *wrap_socket()*.

ssl.**CERT_REQUIRED**
Valor posible para *SSLContext.verify_mode*, o el parámetro `cert_reqs` de *wrap_socket()*. En este modo, se requieren certificados del otro lado de la conexión del socket; se lanzará un *SSLError* si no se proporciona ningún certificado, o si su validación falla. Este modo **no** es suficiente para verificar un certificado en modo cliente, ya que no coincide con los hostnames. *check_hostname* debe estar activado también para verificar la autenticidad de un certificado. *PROTOCOL_TLS_CLIENT* utiliza *CERT_REQUIRED* y activa *check_hostname* por defecto.

Con socket servidor, este modo proporciona una autenticación obligatoria de certificado de cliente TLS. Se envía una solicitud de certificado de cliente al cliente y el cliente debe proporcionar un certificado válido y de confianza.

El uso de esta configuración requiere que se pase un conjunto válido de certificados de CA, ya sea a *SSLContext.load_verify_locations()* o como valor del parámetro `ca_certs` de *wrap_socket()*.

class ssl.**VerifyMode**
Colección *enum.IntEnum* de constantes CERT_*.

Nuevo en la versión 3.6.

ssl.**VERIFY_DEFAULT**
Valor posible para *SSLContext.verify_flags*. En este modo, las listas de revocación de certificado (CRLs) no son verificadas. Por defecto OpenSSL no requiere ni verifica CRLs.

Nuevo en la versión 3.4.

ssl.**VERIFY_CRL_CHECK_LEAF**
Valor posible para *SSLContext.verify_flags*. En este modo, sólo el certificado de pares es verificado pero ninguno de los certificados CA intermedios. El modo requiere una CRL válida que esté firmada por el emisor del certificado de pares (su CA antecesora directa). Si no se ha cargado una CRL adecuada con *SSLContext.load_verify_locations*, la validación fallará.

Nuevo en la versión 3.4.

ssl.**VERIFY_CRL_CHECK_CHAIN**
Valor posible para *SSLContext.verify_flags*. En este modo, las CRLs de todos los certificados en la cadena de certificado de pares son verificadas.

Nuevo en la versión 3.4.

ssl.**VERIFY_X509_STRICT**
Valor posible para *SSLContext.verify_flags* para desactivar soluciones alternativas para certificados X.509 rotos.

Nuevo en la versión 3.4.

ssl.**VERIFY_ALLOW_PROXY_CERTS**
Valor posible para *SSLContext.verify_flags* para desactivar soluciones alternativas para certificados X.509 rotos.

Nuevo en la versión 3.10.

ssl.**VERIFY_X509_TRUSTED_FIRST**
Valor posible para *SSLContext.verify_flags*. Indica a OpenSSL de preferir certificados de confianza al construir la cadena de confianza para validar un certificado. Esta opción está activada por defecto.

Nuevo en la versión 3.4.4.

ssl.**VERIFY_X509_PARTIAL_CHAIN**
Valor posible para *SSLContext.verify_flags*. Indica a OpenSSL que acepte CA intermedias en el almacén de confianza para que se traten como anclajes de confianza, de la misma manera que los certificados de CA raíz autofirmados. Esto hace posible confiar en los certificados emitidos por una CA intermedia sin tener que confiar en su CA raíz antecesora.

Nuevo en la versión 3.10.

class ssl.**VerifyFlags**
Colección *enum.IntFlag* de constantes VERIFY_*.

Nuevo en la versión 3.6.

ssl.**PROTOCOL_TLS**
Selecciona la versión mas alta del protocolo soportada tanto por el cliente como por el servidor. A pesar de su nombre, esta opción puede seleccionar ambos protocolos «SSL» y «TLS».

Nuevo en la versión 3.6.

Obsoleto desde la versión 3.10: Los clientes y servidores TLS requieren diferentes configuraciones predeterminadas para una comunicación segura. La constante del protocolo TLS genérico está en desuso en favor de *PROTOCOL_TLS_CLIENT* y *PROTOCOL_TLS_SERVER*.

ssl.**PROTOCOL_TLS_CLIENT**
Negocie automáticamente la versión de protocolo más alta que admiten tanto el cliente como el servidor, y configure las conexiones del lado del cliente de contexto. El protocolo habilita *CERT_REQUIRED* y *check_hostname* de forma predeterminada.

Nuevo en la versión 3.6.

ssl.**PROTOCOL_TLS_SERVER**
Selecciona la versión mas alta del protocolo soportada tanto por el cliente como por el servidor. A pesar de su nombre, esta opción puede seleccionar ambos protocolos «SSL» y «TLS».

Nuevo en la versión 3.6.

ssl.**PROTOCOL_SSLv23**
Alias para *PROTOCOL_TLS*.

Obsoleto desde la versión 3.6: Utilice en su lugar *PROTOCOL_TLS*.

ssl.**PROTOCOL_SSLv2**
Selecciona la versión 2 de SSL como protocolo de cifrado del canal.

This protocol is not available if OpenSSL is compiled with the `no-ssl2` option.

Advertencia: La versión 2 de SSL es insegura. Su uso es muy desaconsejado.

Obsoleto desde la versión 3.6: OpenSSL a eliminado el soporte para SSLv2.

ssl.**PROTOCOL_SSLv3**
Selecciona la versión 3 de SSL como protocolo de cifrado del canal.

This protocol is not available if OpenSSL is compiled with the `no-ssl3` option.

Advertencia: La versión 3 de SSL es insegura. Su uso es muy desaconsejado.

Obsoleto desde la versión 3.6: OpenSSL ha descontinuado todos los protocolos específicos de la versión. Utilice el protocolo predeterminado *PROTOCOL_TLS_SERVER* o *PROTOCOL_TLS_CLIENT* con *SSLContext.minimum_version* y *SSLContext.maximum_version* en su lugar.

ssl.**PROTOCOL_TLSv1**
Selecciona la versión 1.0 de TLS como protocolo de cifrado del canal.

Obsoleto desde la versión 3.6: OpenSSL ha descontinuado todos los protocolos específicos de la versión.

ssl.**PROTOCOL_TLSv1_1**
Selecciona la versión 1.1 de TLS como protocolo de cifrado del canal. Disponible sólo con openssl en versión 1.0.1+.

Nuevo en la versión 3.4.

Obsoleto desde la versión 3.6: OpenSSL ha descontinuado todos los protocolos específicos de la versión.

ssl.**PROTOCOL_TLSv1_2**
Selecciona la versión 1.1 de TLS como protocolo de cifrado del canal. Disponible sólo con openssl en versión 1.0.1+.

Nuevo en la versión 3.4.

Obsoleto desde la versión 3.6: OpenSSL ha descontinuado todos los protocolos específicos de la versión.

ssl.**OP_ALL**
Activa soluciones alternativas para varios errores presentes en otras implementaciones SSL. Esta opción esta activada por defecto. No necesariamente activa las mismas opciones como la constante `SSL_OP_ALL` de OpenSSL.

Nuevo en la versión 3.2.

ssl.**OP_NO_SSLv2**
Evita una conexión SSLv2. Esta opción sólo es aplicable junto con *PROTOCOL_TLS*. Evita que los pares elijan SSLv2 como versión del protocolo.

Nuevo en la versión 3.2.

Obsoleto desde la versión 3.6: SSLv2 es obsoleto

ssl.**OP_NO_SSLv3**
Evita una conexión SSLv3. Esta opción sólo es aplicable junto con *PROTOCOL_TLS*. Evita que los pares elijan SSLv3 como versión del protocolo.

Nuevo en la versión 3.2.

Obsoleto desde la versión 3.6: SSLv3 es obsoleto

ssl.**OP_NO_TLSv1**
Evita una conexión TLSv1. Esta opción sólo es aplicable junto con *PROTOCOL_TLS*. Evita que los pares elijan TLSv1 como versión del protocolo.

Nuevo en la versión 3.2.

Obsoleto desde la versión 3.7: Esta opción es obsoleta desde OpenSSL 1.1.0, utilice en su lugar los nuevos *SSLContext.minimum_version* y *SSLContext.maximum_version*.

ssl.**OP_NO_TLSv1_1**
Evita una conexión TLSv1.1. Esta opción sólo es aplicable junto con *PROTOCOL_TLS*. Evita que los pares elijan TLSv1.1 como versión del protocolo. Disponible sólo con openssl en versión 1.0.1+.

Nuevo en la versión 3.4.

Obsoleto desde la versión 3.7: Esta opción es obsoleta desde OpenSSL 1.1.0.

ssl.**OP_NO_TLSv1_2**
Evita una conexión TLSv1.2. Esta opción sólo es aplicable junto con *PROTOCOL_TLS*. Evita que los pares elijan TLSv1.2 como versión del protocolo. Disponible sólo con openssl en versión 1.0.1+.

Nuevo en la versión 3.4.

Obsoleto desde la versión 3.7: Esta opción es obsoleta desde OpenSSL 1.1.0.

ssl.**OP_NO_TLSv1_3**
Evita una conexión TLSv1.3. Esta opción sólo es aplicable junto con *PROTOCOL_TLS*. Evita que los pares elijan TLSv1.3 como versión del protocolo. TLS 1.3 está disponible con OpenSSL 1.1.1 o superior. Cuando Python es compilado contra una versión mas antigua de OpenSSL, la opción vale *0* por defecto.

Nuevo en la versión 3.7.

Obsoleto desde la versión 3.7: Esta opción es obsoleta desde OpenSSL 1.1.0. Ha sido agregada a 2.7.15, 3.6.3 y 3.7.0 por retro-compatibilidad con OpenSSL 1.0.2.

ssl.**OP_NO_RENEGOTIATION**
Desactiva toda re-negociación en TLSv1.2 y anteriores. No envía mensajes HelloRequest e ignora solicitudes de re-negociación vía ClientHello.

Esta opción sólo está disponible con OpenSSL 1.1.0h y posteriores.

Nuevo en la versión 3.7.

ssl.**OP_CIPHER_SERVER_PREFERENCE**
Utiliza la preferencia de ordenación de cifrado del servidor, en lugar de la del cliente. Esta opción no tiene efecto en los sockets del cliente ni en los sockets del servidor SSLv2.

Nuevo en la versión 3.3.

ssl.**OP_SINGLE_DH_USE**
Evita la reutilización de la misma clave DH para distintas sesiones SSL. Esto mejora el secreto hacia adelante pero requiere más recursos computacionales. Esta opción sólo se aplica a los sockets del servidor.

Nuevo en la versión 3.3.

ssl.**OP_SINGLE_ECDH_USE**
Evita la reutilización de la misma clave ECDH para distintas sesiones SSL. Esto mejora el secreto hacia adelante pero requiere más recursos computacionales. Esta opción sólo se aplica a los sockets del servidor.

Nuevo en la versión 3.3.

ssl.**OP_ENABLE_MIDDLEBOX_COMPAT**
Enviar mensajes Change Cipher Spec (CCS) ficticios en el handshake de TLS 1.3 para que una conexión TLS 1.3 se parezca más a una conexión TLS 1.2.

Esta opción sólo está disponible con OpenSSL 1.1.1 y posteriores.

Nuevo en la versión 3.8.

ssl.**OP_NO_COMPRESSION**
Desactivar la compresión en el canal SSL. Esto es útil si el protocolo de la aplicación soporta su propio esquema de compresión.

Nuevo en la versión 3.3.

class ssl.**Options**
Colección *enum.IntFlag* de constantes OP_*.

ssl.**OP_NO_TICKET**
Evita que el lado del cliente solicite un ticket de sesión.

Nuevo en la versión 3.6.

ssl.**OP_IGNORE_UNEXPECTED_EOF**
Ignore el cierre inesperado de las conexiones TLS.

Esta opción sólo está disponible con OpenSSL 1.0.0 y posteriores.

Nuevo en la versión 3.10.

ssl.**HAS_ALPN**
Si la biblioteca OpenSSL tiene soporte incorporado para la extensión TLS *Application-Layer Protocol Negotiation* como se describe en **RFC 7301**.

Nuevo en la versión 3.5.

ssl.**HAS_NEVER_CHECK_COMMON_NAME**
Si la biblioteca OpenSSL tiene soporte incorporado para no comprobar el nombre común del sujeto y *SSLContext.hostname_checks_common_name* es modificable.

Nuevo en la versión 3.7.

ssl.**HAS_ECDH**
Si la biblioteca OpenSSL tiene soporte incorporado para el intercambio de claves Diffie-Hellman basado en Elliptic Curve. Esto debería ser cierto a menos que la función haya sido desactivada explícitamente por el distribuidor.

Nuevo en la versión 3.3.

ssl.**HAS_SNI**
Si la biblioteca OpenSSL tiene soporte incorporado para la extensión *Server Name Indication* (como se define en **RFC 6066**).

Nuevo en la versión 3.2.

ssl.**HAS_NPN**
Si la biblioteca OpenSSL tiene soporte incorporado para *Next Protocol Negotiation* como se describe en Application Layer Protocol Negotiation. Cuando es verdadero, puede utilizar el método *SSLContext.set_npn_protocols()* para anunciar los protocolos que desea soportar.

Nuevo en la versión 3.3.

ssl.**HAS_SSLv2**
Si la biblioteca OpenSSL tiene soporte incorporado para el protocolo SSL 2.0.

Nuevo en la versión 3.7.

ssl.**HAS_SSLv3**
Si la biblioteca OpenSSL tiene soporte incorporado para el protocolo SSL 3.0.

Nuevo en la versión 3.7.

ssl.**HAS_TLSv1**
Si la biblioteca OpenSSL tiene soporte incorporado para el protocolo TLS 1.0.

Nuevo en la versión 3.7.

ssl.**HAS_TLSv1_1**
Si la biblioteca OpenSSL tiene soporte incorporado para el protocolo TLS 1.1.

Nuevo en la versión 3.7.

ssl.**HAS_TLSv1_2**
Si la biblioteca OpenSSL tiene soporte incorporado para el protocolo TLS 1.2.

Nuevo en la versión 3.7.

ssl.**HAS_TLSv1_3**
Si la biblioteca OpenSSL tiene soporte incorporado para el protocolo TLS 1.3.

Nuevo en la versión 3.7.

ssl.**CHANNEL_BINDING_TYPES**
Lista de tipos de enlace de canales TLS admitidos. Las cadenas de caracteres en esta lista pueden ser usadas como argumentos para *SSLSocket.get_channel_binding()*.

Nuevo en la versión 3.3.

ssl.**OPENSSL_VERSION**
La cadena de versión de la biblioteca OpenSSL cargada por el intérprete:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

Nuevo en la versión 3.2.

ssl.**OPENSSL_VERSION_INFO**
Una tupla de cinco números enteros representando la información de versión de la biblioteca OpenSSL:

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

Nuevo en la versión 3.2.

ssl.**OPENSSL_VERSION_NUMBER**
El número de versión en bruto de la biblioteca OpenSSL, como un único número entero:

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

Nuevo en la versión 3.2.

ssl.**ALERT_DESCRIPTION_HANDSHAKE_FAILURE**
ssl.**ALERT_DESCRIPTION_INTERNAL_ERROR**
ALERT_DESCRIPTION_*
Descripciones de alertas de **RFC 5246** y otras. El IANA TLS Alert Registry contiene esta lista y las referencias a las RFC donde se define su significado.

Se utiliza como valor de retorno de la función callback en *SSLContext.set_servername_callback()*.

Nuevo en la versión 3.4.

class ssl.**AlertDescription**
Colección *enum.IntEnum* de constantes ALERT_DESCRIPTION_*.

Nuevo en la versión 3.6.

Purpose.**SERVER_AUTH**
Opción para *create_default_context()* y *SSLContext.load_default_certs()*. Este valor indica que el contexto puede utilizarse para autenticar servidores web (por lo tanto, se utilizará para crear sockets del lado del cliente).

Nuevo en la versión 3.4.

Purpose.**CLIENT_AUTH**
Opción para *create_default_context()* y *SSLContext.load_default_certs()*. Este valor indica que el contexto puede utilizarse para autenticar clientes web (por lo tanto, se utilizará para crear sockets del lado del servidor).

Nuevo en la versión 3.4.

class ssl.**SSLErrorNumber**
Colección *enum.IntEnum* de constantes SSL_ERROR_*.

Nuevo en la versión 3.6.

class ssl.**TLSVersion**
Colección *enum.IntEnum* de versiones SSL y TLS para *SSLContext.maximum_version* y *SSLContext.minimum_version*.

Nuevo en la versión 3.7.

TLSVersion.**MINIMUM_SUPPORTED**

TLSVersion.**MAXIMUM_SUPPORTED**
La mínima o máxima versión soportada de SSL o TLS. Estas son constantes mágicas. Sus valores no reflejan la mas baja o mas alta versión TLS/SSL disponible.

TLSVersion.**SSLv3**

TLSVersion.**TLSv1**

TLSVersion.**TLSv1_1**

TLSVersion.**TLSv1_2**

TLSVersion.**TLSv1_3**
SSL 3.0 a TLS 1.3.

Obsoleto desde la versión 3.10: Todos los miembros de *TLSVersion*, excepto *TLSVersion.TLSv1_2* y *TLSVersion.TLSv1_3*, están en desuso.

18.3.2 Sockets SSL

class ssl.**SSLSocket**(*socket.socket*)
Los sockets SSL proporcionan los siguientes métodos de *Objetos Socket*:

- *accept()*
- *bind()*
- *close()*
- *connect()*
- *detach()*
- *fileno()*
- *getpeername()*, *getsockname()*
- *getsockopt()*, *setsockopt()*
- *gettimeout()*, *settimeout()*, *setblocking()*
- *listen()*
- *makefile()*
- *recv()*, *recv_into()* (pero no se admite pasar un argumento `flags` diferente de cero)
- *send()*, *sendall()* (con la misma limitación)

- *sendfile()* (pero *os.sendfile* sera utilizado sólo para sockets de texto simple, sino *send()* sera utilizado)
- *shutdown()*

Sin embargo, dado que el protocolo SSL (y TLS) tiene su propia estructura encima de TCP, la abstracción de los sockets SSL puede, en ciertos aspectos, divergir de la especificación de los sockets normales a nivel de SO. Ver especialmente las *notas sobre sockets no bloqueantes*.

Instancias de *SSLSocket* deben ser creadas usando el método *SSLContext.wrap_socket()*.

Distinto en la versión 3.5: El método `sendfile()` ha sido agregado.

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 the maximum total duration of the shutdown.

Obsoleto desde la versión 3.6: Crear una instancia de *SSLSocket* directamente es obsoleto, utilice *SSLContext.wrap_socket()* para envolver un socket.

Distinto en la versión 3.7: Las instancias de *SSLSocket* deben crearse con *wrap_socket()*. En versiones anteriores, era posible crear instancias directamente. Esto nunca fue documentado ni soportado oficialmente.

Distinto en la versión 3.10: Python ahora usa `SSL_read_ex` y `SSL_write_ex` internamente. Las funciones admiten la lectura y escritura de datos de más de 2 GB. La escritura de datos de longitud cero ya no falla con un error de violación de protocolo.

Los sockets SSL tienen también los siguientes métodos y atributos adicionales:

`SSLSocket.`**`read`**(*len=1024*, *buffer=None*)

Lee hasta *len* bytes de datos del socket SSL y retorna el resultado como una instancia `bytes`. Si *buffer* es especificado, entonces se lee hacia el búfer en su lugar, y retorna el número de bytes leídos.

Lanza *SSLWantReadError* o *SSLWantWriteError* si el socket es *no-bloqueante* y la lectura se bloquearía.

Como en cualquier momento es posible una re-negociación, una llamada a *read()* también puede provocar operaciones de escritura.

Distinto en la versión 3.5: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration to read up to *len* bytes.

Obsoleto desde la versión 3.6: Utilice `recv()` en lugar de *read()*.

`SSLSocket.`**`write`**(*buf*)

Escribe *buf* en el socket SSL y retorna el número de bytes escritos. El argumento *buf* debe ser un objeto que soporte la interfaz búfer.

Lanza *SSLWantReadError* o *SSLWantWriteError* si el socket es *no-bloqueante* y la escritura se bloquearía.

Como en cualquier momento es posible una re-negociación, una llamada a *write()* también puede provocar operaciones de lectura.

Distinto en la versión 3.5: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration to write *buf*.

Obsoleto desde la versión 3.6: Utilice `send()` en lugar de *write()*.

Nota: Los métodos *read()* y *write()* son los métodos de bajo nivel que leen y escriben datos no cifrados a nivel de aplicación y los descifran/cifran a datos cifrados a nivel de cable. Estos métodos requieren una conexión SSL activa, es decir, que se haya completado el handshake y no se haya llamado a *SSLSocket.unwrap()*.

Normalmente se deberían utilizar los métodos de la API de sockets como *recv()* y *send()* en lugar de estos métodos.

`SSLSocket.`**`do_handshake`**`()`

Realiza el handshake de configuración SSL.

Distinto en la versión 3.4: El método handshake también realiza *match_hostname()* cuando el atributo *check_hostname* del *context* del socket es verdadero.

Distinto en la versión 3.5: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the 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 sent to the peer.

SSLSocket.**getpeercert**(*binary_form=False*)

Si no hay un certificado para el peer en el otro extremo de la conexión, devuelve None. Si el handshake SSL no se ha realizado todavía, lanza *ValueError*.

Si el parámetro binary_form es *False*, y se ha recibido un certificado del peer, este método devuelve una instancia *dict*. Si el certificado no fue validado, el dict está vacío. Si el certificado fue validado, devuelve un dict con varias claves, entre ellas subject (la entidad para la que se emitió el certificado) y issuer (la entidad que emite el certificado). Si un certificado contiene una instancia de la extensión *Subject Alternative Name* (véase **RFC 3280**), también habrá una clave subjectAltName en el diccionario.

Los campos subject y issuer son tuplas que contienen la secuencia de nombres distinguidos relativos (RDNs) indicados en la estructura de datos del certificado para los campos respectivos, y cada RDN es una secuencia de pares nombre-valor. Este es un ejemplo del mundo real:

```
{'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: Para validar un certificado para un servicio concreto, puede utilizar la función *match_hostname()*.

Si el parámetro binary_form es *True*, y se proporcionó un certificado, este método devuelve la forma codificada en DER del certificado completo como una secuencia de bytes, o *None* si el par no proporcionó un certificado. El hecho de que el par proporcione un certificado depende del rol del socket SSL:

- para un socket SSL cliente, el servidor siempre proporcionará un certificado, independientemente de si se requirió la validación;
- para un socket SSL servidor, el cliente sólo proporcionará un certificado cuando lo solicite el servidor; por lo tanto *getpeercert()* devolverá *None* si ha utilizado *CERT_NONE* (en lugar de *CERT_OPTIONAL* o CERT REQUIRED).

Distinto en la versión 3.2: El diccionario devuelto incluye elementos adicionales tales como issuer y notBefore.

Distinto en la versión 3.4: *ValueError* se lanza cuando no se realiza el handshake. El diccionario retornado incluye elementos de extensión X509v3 adicionales como crlDistributionPoints, caIssuers y OCSP URIs.

Distinto en la versión 3.9: Las cadenas de direcciones IPv6 ya no tienen una nueva línea al final.

SSLSocket.**cipher**()
Retorna 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 el número de bits secretos que se están utilizando. Si no se ha establecido ninguna conexión, retorna None.

SSLSocket.**shared_ciphers**()
Return the list of ciphers available in both the client and server. 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**()
Retorna el algoritmo de compresión utilizado como una cadena de caracteres, o None si la conexión no está comprimida.

Si el protocolo de nivel superior soporta su propio mecanismo de compresión, puede utilizar *OP_NO_COMPRESSION* para desactivar la compresión a nivel de SSL.

Nuevo en la versión 3.3.

SSLSocket.**get_channel_binding**(*cb_type='tls-unique'*)
Obtiene los datos de enlace del canal para la conexión actual, como un objeto bytes. Retorna None si no está conectado o no se ha completado el handshake.

El parámetro *cb_type* permite seleccionar el tipo de enlace de canal deseado. Los tipos de enlace de canal válidos se enumeran en la lista *CHANNEL_BINDING_TYPES*. Actualmente, sólo se admite la vinculación de canal "tls-unique", definida por **RFC 5929**. *ValueError* se lanzará si se solicita un tipo de vinculación de canal no admitido.

Nuevo en la versión 3.3.

SSLSocket.**selected_alpn_protocol**()
Retorna el protocolo que fue seleccionado durante el handshake TLS. Si no se ha llamado a *SSLContext.set_alpn_protocols()*, si la otra parte no soporta ALPN, si este socket no soporta ninguno de los protocolos propuestos por el cliente, o si el handshake no ha ocurrido todavía, se devuelve None.

Nuevo en la versión 3.5.

SSLSocket.**selected_npn_protocol**()
Devuelve el protocolo de nivel superior que se seleccionó durante el handshake TLS/SSL. Si no se llamó a *SSLContext.set_npn_protocols()*, o si la otra parte no soporta NPN, o si el handshake aún no ha ocurrido, esto devolverá None.

Nuevo en la versión 3.3.

Obsoleto desde la versión 3.10: NPN ha sido reemplazada por ALPN

SSLSocket.**unwrap**()
Realiza el handshake de cierre de SSL, que elimina la capa TLS del socket subyacente, y devuelve el objeto socket subyacente. Esto puede utilizarse para pasar de una operación encriptada sobre una conexión a una sin encriptar. El socket devuelto debe utilizarse siempre para la comunicación posterior con el otro lado de la conexión, en lugar del socket original.

SSLSocket.**verify_client_post_handshake**()
Solicita la autenticación post-handshake (PHA) de un cliente TLS 1.3. PHA sólo puede iniciarse para una conexión TLS 1.3 desde un socket del lado del servidor, después del handshake TLS inicial y con PHA habilitado en ambos lados, ver *SSLContext.post_handshake_auth*.

El método no realiza un intercambio de certificados inmediatamente. El lado del servidor envía una CertificateRequest durante el siguiente evento de escritura y espera que el cliente responda con un certificado en el siguiente evento de lectura.

Si alguna precondición no se cumple (por ejemplo, no es TLS 1.3, PHA no está habilitado), se genera un *SSLError*.

Nota: Sólo está disponible con OpenSSL 1.1.1 y TLS 1.3 habilitados. Sin el soporte de TLS 1.3, el método lanza *NotImplementedError*.

Nuevo en la versión 3.8.

SSLSocket.**version**()
Return the actual SSL protocol version negotiated by the connection as a string, or None if no secure connection is established. As of this writing, possible return values include "SSLv2", "SSLv3", "TLSv1", "TLSv1.1" and "TLSv1.2". Recent OpenSSL versions may define more return values.

Nuevo en la versión 3.5.

SSLSocket.**pending**()
Retorna el número de bytes ya descifrados disponibles para leer, pendientes de la conexión.

SSLSocket.**context**
El objeto *SSLContext* al que está vinculado este socket SSL. Si el socket SSL fue creado usando la función obsoleta *wrap_socket()* (en lugar de *SSLContext.wrap_socket()*), este es un objeto de contexto personalizado creado para este socket SSL.

Nuevo en la versión 3.2.

SSLSocket.**server_side**
Un booleano que es True para los sockets del lado del servidor y False para los sockets del lado del cliente.

Nuevo en la versión 3.2.

SSLSocket.**server_hostname**
Hostname del servidor: tipo *str*, o None para el socket del lado del servidor o si el hostname no fue especificado en el constructor.

Nuevo en la versión 3.2.

Distinto en la versión 3.7: El atributo es ahora siempre texto ASCII. Cuando server_hostname es un nombre de dominio internacionalizado (IDN), este atributo almacena ahora la forma de etiqueta A ("xn--pythn-mua.org"), en lugar de la forma de etiqueta U ("pythön.org").

SSLSocket.**session**
La *SSLSession* para esta conexión SSL. La sesión está disponible para los sockets del lado del cliente y del servidor después de que se haya realizado el handshake TLS. Para los sockets del cliente la sesión puede ser establecida antes de que *do_handshake()* haya sido llamado para reutilizar una sesión.

Nuevo en la versión 3.6.

SSLSocket.**session_reused**
Nuevo en la versión 3.6.

18.3.3 Contextos SSL

Nuevo en la versión 3.2.

Un contexto SSL contiene varios datos más duraderos que las conexiones SSL individuales, como opciones de configuración SSL, certificado(s) y clave(s) privada(s). También gestiona un cache de sesiones SSL para sockets del lado del servidor, para acelerar conexiones repetidas de los mismos clientes.

class ssl.**SSLContext**(*protocol=None*)
Crea un nuevo contexto SSL. Puede pasar *protocolo* que debe ser una de las constantes PROTOCOL_* definidas en este módulo. El parámetro especifica la versión del protocolo SSL a utilizar. Típicamente, el servidor elige una versión particular del protocolo, y el cliente debe adaptarse a la elección del servidor. La mayoría de las versiones no son interoperables con las demás. Si no se especifica, el valor por defecto es *PROTOCOL_TLS*; proporciona la mayor compatibilidad con otras versiones.

Esta es una tabla que muestra qué versiones de un cliente (en la parte inferior) pueden conectarse a qué versiones de un servidor (en la parte superior):

cliente / **servidor**	**SSLv2**	**SSLv3**	**TLS**[3]	**TLSv1**	**TLSv1.1**	**TLSv1.2**
SSLv2	si	no	no[1]	no	no	no
SSLv3	no	si	no[2]	no	no	no
TLS (SSLv23)[3]	no[1]	no[2]	si	si	si	si
TLSv1	no	no	si	si	no	no
TLSv1.1	no	no	si	no	si	no
TLSv1.2	no	no	si	no	no	si

Ver también:

`create_default_context()` permite al módulo `ssl` elegir la configuración de seguridad para un propósito determinado.

Distinto en la versión 3.6: El contexto se crea con valores seguros por defecto. Las opciones `OP_NO_COMPRESSION`, `OP_CIPHER_SERVER_PREFERENCE`, `OP_SINGLE_DH_USE`, `OP_SINGLE_ECDH_USE`, `OP_NO_SSLv2` (excepto para `PROTOCOL_SSLv2`), y `OP_NO_SSLv3` (excepto para `PROTOCOL_SSLv3`) están establecidas por defecto. La lista inicial de conjuntos de cifrado sólo contiene cifrados `HIGH`, ningún cifrado `NULL` y ningún cifrado `MD5` (excepto para `PROTOCOL_SSLv2`).

Obsoleto desde la versión 3.10: `SSLContext` sin argumento de protocolo está en desuso. La clase de contexto requerirá el protocolo `PROTOCOL_TLS_CLIENT` o `PROTOCOL_TLS_SERVER` en el futuro.

Distinto en la versión 3.10: Los conjuntos de cifrado predeterminados ahora incluyen solo cifrados AES y ChaCha20 seguros con secreto directo y nivel de seguridad 2. Están prohibidas las claves RSA y DH con menos de 2048 bits y las claves ECC con menos de 224 bits. `PROTOCOL_TLS`, `PROTOCOL_TLS_CLIENT` y `PROTOCOL_TLS_SERVER` usan TLS 1.2 como versión mínima de TLS.

Los objetos `SSLContext` tienen los siguientes métodos y atributos:

SSLContext.**cert_store_stats**()

Obtiene estadísticas sobre las cantidades de certificados X.509 cargados, la cantidad de certificados X.509 marcados como certificados CA y las listas de revocación de certificados como diccionario.

Ejemplo para un contexto con un certificado CA y otro certificado:

```
>>> 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. Otherwise the private key will be taken from *certfile* as well. See the discussion of *Certificados* for more information on how the certificate is stored in the *certfile*.

El argumento *password* puede ser una función a la que llamar para obtener la contraseña para descifrar la clave privada. Sólo se llamará si la clave privada está encriptada y se necesita una contraseña. Se llamará sin argumentos, y deberá devolver una cadena de caracteres, bytes o bytearray. Si el valor devuelto es una cadena de caracteres, se codificará como UTF-8 antes de utilizarlo para descifrar la clave. Alternativamente, se puede suministrar un valor de cadena de caracteres, bytes o bytearray directamente como argumento *password*. Se ignorará si la clave privada no está cifrada y no se necesita una contraseña.

Si el argumento *password* no es especificado y una contraseña es requerida, el mecanismo de solicitud de contraseña incorporado de OpenSSL se usará para solicitarle una contraseña al usuario de forma interactiva.

[3] El protocolo TLS 1.3 estará disponible con `PROTOCOL_TLS` en OpenSSL >= 1.1.1. No existe una constante PROTOCOL dedicada sólo a TLS 1.3.

[1] `SSLContext` desactiva SSLv2 con `OP_NO_SSLv2` por defecto.

[2] `SSLContext` desactiva SSLv3 con `OP_NO_SSLv3` por defecto.

Un *SSLError* es lanzado si la clave privada no coincide con el certificado.

Distinto en la versión 3.3: Nuevo argumento opcional *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 all systems it calls *SSLContext.set_default_verify_paths()*. In the future the method may load CA certificates from other locations, too.

La opción *purpose* especifica qué tipo de certificados CA se cargan. La configuración por defecto *Purpose.SERVER_AUTH* carga certificados, que están marcados y son de confianza para la autenticación del servidor web TLS (sockets del lado del cliente). *Purpose.CLIENT_AUTH* carga certificados CA para la verificación de certificados de cliente en el lado del servidor.

Nuevo en la versión 3.4.

SSLContext.**load_verify_locations**(*cafile=None*, *capath=None*, *cadata=None*)
Carga un conjunto de certificados de «autoridad de certificación» (CA) usados para validar certificados de otros pares cuando *verify_mode* es distinto de *CERT_NONE*. Debe especificarse al menos uno de *cafile* o *capath*.

Este método puede cargar también listas de revocación de certificados (CRLs) en formato PEM o DER. Para poder usar CRLs, *SSLContext.verify_flags* debe ser configurado correctamente.

La cadena de caracteres *cafile*, si está presente, es la ruta a un archivo de certificados CA concatenados en formato PEM. Vea la discusión de *Certificados* para más información acerca de como organizar los certificados en este archivo.

La cadena de caracteres *capath*, si está presente, es la ruta a un directorio que contiene varios certificados CA en formato PEM, siguiendo la disposición específica de OpenSSL.

El objeto *cadata*, si está presente, es una cadena de caracteres ASCII de uno o más certificados codificados en PEM o un *bytes-like object* de certificados codificados en DER. Al igual que con *capath*, las líneas adicionales alrededor de los certificados codificados en PEM se ignoran, pero debe haber al menos un certificado.

Distinto en la versión 3.4: Nuevo argumento opcional *cadata*

SSLContext.**get_ca_certs**(*binary_form=False*)
Obtiene una lista de certificados de «autoridad de certificación» (CA) cargados. Si el parámetro binary_form es *False* cada entrada de la lista es un diccionario como la salida de *SSLSocket.getpeercert()*. En caso contrario, el método devuelve una lista de certificados codificados con DER. La lista devuelta no contiene certificados de *capath* a menos que un certificado haya sido solicitado y cargado por una conexión SSL.

Nota: Los certificados de un directorio capath no se cargan a menos que se hayan utilizado al menos una vez.

Nuevo en la versión 3.4.

SSLContext.**get_ciphers**()
Obtiene una lista de cifrados habilitados. La lista está en orden de prioridad de cifrado. Véase *SSLContext.set_ciphers()*.

Ejemplo:

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()
[{'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'}]
```

Nuevo en la versión 3.6.

SSLContext.**set_default_verify_paths**()
Carga un conjunto de certificados de «autoridad de certificación» (CA) por defecto desde una ruta del sistema de archivos definida al construir la biblioteca OpenSSL. Desafortunadamente, no hay una manera fácil de saber si este método tiene éxito: no se devuelve ningún error si no se encuentran certificados. Sin embargo, cuando la biblioteca OpenSSL se proporciona como parte del sistema operativo, es probable que esté configurada correctamente.

SSLContext.**set_ciphers**(*ciphers*)
Establece los cifrados disponibles para los sockets creados con este contexto. Debe ser una cadena de caracteres con el formato de la lista de cifrado de OpenSSL. Si no se puede seleccionar ningún cifrado (porque las opciones en tiempo de compilación u otra configuración prohíben el uso de todos los cifrados especificados), se lanzará un *SSLError*.

Nota: cuando se conecta, el método *SSLSocket.cipher()* de los sockets SSL dará el cifrado actualmente seleccionado.

OpenSSL 1.1.1 tiene suites de cifrado TLS 1.3 habilitadas por defecto. Las suites no se pueden desactivar con *set_ciphers()*.

SSLContext.**set_alpn_protocols**(*protocols*)
Especifica qué protocolos debe anunciar el socket durante el handshake SSL/TLS. Debe ser una lista de cadenas de caracteres ASCII, como `['http/1.1', 'spdy/2']`, ordenadas por preferencia. La selección de un protocolo ocurrirá durante el handshake, y se desarrollará de acuerdo con **RFC 7301**. Después de un handshake exitoso, el método *SSLSocket.selected_alpn_protocol()* devolverá el protocolo acordado.

Este método lanzará *NotImplementedError* si *HAS_ALPN* es `False`.

Nuevo en la versión 3.5.

SSLContext.**set_npn_protocols**(*protocols*)
Especifica qué protocolos debe anunciar el socket durante el handshake SSL/TLS. Debe ser una lista de cadenas, como `['http/1.1', 'spdy/2']`, ordenadas por preferencia. La selección de un protocolo ocurrirá durante el handshake, y se desarrollará de acuerdo a la Negociación del Protocolo de la Capa de Aplicación. Después de un handshake exitoso, el método *SSLSocket.selected_npn_protocol()* devolverá el protocolo acordado.

Este método lanzará *NotImplementedError* si *HAS_NPN* es `False`.

Nuevo en la versión 3.3.

Obsoleto desde la versión 3.10: NPN ha sido reemplazada por ALPN

SSLContext.**sni_callback**
Registra una función callback que se llamará después de que el servidor SSL/TLS haya recibido el mensaje de diálogo TLS Client Hello cuando el cliente TLS especifique una indicación de nombre de servidor. El mecanismo de indicación de nombre de servidor se especifica en **RFC 6066** sección 3 - Server Name Indication.

Sólo se puede establecer una función callback por `SSLContext`. Si *sni_callback* se establece como `None`, la función callback se desactiva. Si se llama a esta función una vez más, se desactivará la función callback registrada anteriormente.

La función callback será llamada con tres argumentos; el primero es el *ssl.SSLSocket*, el segundo es una cadena que representa el nombre del servidor con el que el cliente pretende comunicarse (o *None* si el TLS Client Hello no contiene un nombre de servidor) y el tercer argumento es el *SSLContext* original. El argumento del nombre del servidor es un texto. En el caso de los nombres de dominio internacionalizados, el nombre del servidor es un IDN etiqueta A (`"xn--pythn-mua.org"`).

Un uso típico de esta función callback es cambiar el atributo *SSLSocket.context* de *ssl.SSLSocket* por un nuevo objeto de tipo *SSLContext* que representa una cadena de certificados que coincide con el nombre del servidor.

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*. The *SSLSocket.getpeercert()*, *SSLSocket.cipher()* and *SSLSocket.compression()* methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not return meaningful values nor can they be called safely.

La función *sni_callback* debe devolver `None` para permitir que la negociación TLS continúe. Si se requiere un fallo TLS, se puede devolver una constante *ALERT_DESCRIPTION_**. Otros valores de retorno resultarán en un error fatal TLS con *ALERT_DESCRIPTION_INTERNAL_ERROR*.

Si se lanza una excepción desde la función *sni_callback* la conexión TLS terminará con un mensaje de alerta TLS fatal *ALERT_DESCRIPTION_HANDSHAKE_FAILURE*.

Este método lanzará *NotImplementedError* si la biblioteca OpenSSL tenía definido OPENSSL_NO_TLSEXT cuando se construyó.

Nuevo en la versión 3.7.

SSLContext.**set_servername_callback**(*server_name_callback*)
Se trata de una API heredada que se mantiene por compatibilidad con versiones anteriores. Cuando sea posible, debería utilizar *sni_callback* en su lugar. El *server_name_callback* dado es similar a *sni_callback*, excepto que cuando el nombre del servidor es un nombre de dominio internacionalizado codificado con IDN, el *server_name_callback* recibe una etiqueta U decodificada (`"pythön.org"`).

Si hay un error de decodificación en el nombre del servidor, la conexión TLS terminará con un mensaje fatal de alerta TLS *ALERT_DESCRIPTION_INTERNAL_ERROR* al cliente.

Nuevo en la versión 3.4.

SSLContext.**load_dh_params**(*dhfile*)
Carga los parámetros de generación de claves para el intercambio de claves Diffie-Hellman (DH). El uso del intercambio de claves DH mejora el secreto hacia adelante a expensas de recursos computacionales (tanto en el servidor como en el cliente). El parámetro *dhfile* debe ser la ruta de un archivo que contenga los parámetros DH en formato PEM.

Esta configuración no se aplica a los sockets de los clientes. También puede utilizar la opción *OP_SINGLE_DH_USE* para mejorar aún más la seguridad.

Nuevo en la versión 3.3.

SSLContext.**set_ecdh_curve**(*curve_name*)
Establece el nombre de la curva para el intercambio de claves Diffie-Hellman basado en la curva elíptica (ECDH). ECDH es significativamente más rápido que el DH normal, aunque podría decirse que es igual de seguro. El parámetro *curve_name* debe ser una cadena que describa una curva elíptica conocida, por ejemplo `prime256v1` para una curva ampliamente soportada.

Esta configuración no se aplica a los sockets de los clientes. También puede utilizar la opción *OP_SINGLE_ECDH_USE* para mejorar aún más la seguridad.

Este método no está disponible si *HAS_ECDH* es `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*)

Envuelve un socket Python existente *sock* y devuelve una instancia de *SSLContext.sslsocket_class* (por defecto *SSLSocket*). El socket SSL devuelto está ligado al contexto, su configuración y certificados. *sock* debe ser un socket *SOCK_STREAM*; otros tipos de socket no son soportados.

El parámetro `server_side` es un booleano que identifica si se desea un comportamiento del lado del servidor o del lado del cliente en este socket.

Para los sockets del lado del cliente, la construcción del contexto es perezosa; si el socket subyacente no está conectado todavía, la construcción del contexto se realizará después de llamar a `connect()` en el socket. Para los sockets del lado del servidor, si el socket no tiene un par remoto, se asume que es un socket a la escucha, y la envoltura SSL del lado del servidor se realiza automáticamente en las conexiones del cliente aceptadas a través del método `accept()`. El método puede lanzar *SSLError*.

En las conexiones de cliente, el parámetro opcional *server_hostname* especifica el nombre del servicio al que nos estamos conectando. Esto permite que un único servidor aloje varios servicios basados en SSL con certificados distintos, de forma similar a los hosts virtuales HTTP. Al especificar *server_hostname* se producirá un *ValueError* si *server_side* es verdadero.

El parámetro `do_handshake_on_connect` especifica si se hace el handshake SSL automáticamente después de hacer un `socket.connect()`, o si el programa de aplicación lo llamará explícitamente, invocando el método *SSLSocket.do_handshake()*. Llamar explícitamente a *SSLSocket.do_handshake()* da al programa el control sobre el comportamiento de bloqueo de la E/S del socket involucrada en el handshake.

El parámetro `suppress_ragged_eofs` especifica cómo el método `SSLSocket.recv()` debe señalar los EOF inesperados desde el otro extremo de la conexión. Si se especifica como *True* (el valor por defecto), devuelve un EOF normal (un objeto bytes vacío) en respuesta a los errores EOF inesperados que se produzcan desde el socket subyacente; si *False*, lanzará las excepciones al llamador.

session, véase *session*.

Distinto en la versión 3.5: Siempre permite pasar un server_hostname, incluso si OpenSSL no tiene SNI.

Distinto en la versión 3.6: Se agregó el argumento *session*.

Distinto en la versión 3.7: El método retorna una instancia de *SSLContext.sslsocket_class* en lugar de un *SSLSocket* rígidamente programado.

`SSLContext.`**`sslsocket_class`**

El tipo de retorno de *SSLContext.wrap_socket()*, por defecto es *SSLSocket*. El atributo puede anularse en la instancia de la clase para devolver una subclase personalizada de *SSLSocket*.

Nuevo en la versión 3.7.

`SSLContext.`**`wrap_bio`**(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)

Envuelve los objetos BIO *incoming* y *outgoing* y devuelve una instancia de *SSLContext.sslobject_class* (por defecto *SSLObject*). Las rutinas SSL leerán los datos de entrada de la BIO entrante y escribirán los datos en la BIO saliente.

Los parámetros *server_side*, *server_hostname* y *session* tienen el mismo significado que en *SSLContext.wrap_socket()*.

Distinto en la versión 3.6: Se agregó el argumento *session*.

Distinto en la versión 3.7: El método retorna una instancia de *SSLContext.sslobject_class* en lugar de un *SSLObject* rígidamente programado.

SSLContext.**sslobject_class**
El tipo de retorno de *SSLContext.wrap_bio()*, por defecto es *SSLObject*. El atributo puede anularse en la instancia de la clase para devolver una subclase personalizada de *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**
Si debe coincidir con el nombre de host del certificado de pares en *SSLSocket.do_handshake()*. El *verify_mode* del contexto debe establecerse en *CERT_OPTIONAL* o *CERT_REQUIRED*, y debe pasar *server_hostname* a *wrap_socket()* para que coincida con el nombre de host. La activación de la comprobación del nombre de host configura automáticamente *verify_mode* de *CERT_NONE* a *CERT_REQUIRED*. No se puede volver a establecer en *CERT_NONE* siempre que la comprobación del nombre de host esté habilitada. El protocolo *PROTOCOL_TLS_CLIENT* habilita la verificación del nombre de host de forma predeterminada. Con otros protocolos, la verificación del nombre de host debe habilitarse explícitamente.

Ejemplo:

```
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* ahora se cambia automáticamente a *CERT_REQUIRED* cuando la comprobación del hostname está activada y *verify_mode* es *CERT_NONE*. Anteriormente la misma operación habría fallado con un *ValueError*.

SSLContext.**keylog_filename**
Escribe las claves TLS en un archivo keylog, siempre que se genere o reciba material de claves. El archivo keylog está diseñado únicamente para fines de depuración. El formato del archivo está especificado por NSS y es utilizado por muchos analizadores de tráfico como Wireshark. El archivo de registro se abre en modo sólo añadir. Las escrituras se sincronizan entre hilos, pero no entre procesos.

Nuevo en la versión 3.8.

SSLContext.**maximum_version**
Un miembro enumeración de *TLSVersion* que representa la versión más alta de TLS soportada. El valor por defecto es *TLSVersion.MAXIMUM_SUPPORTED*. El atributo es de sólo lectura para protocolos distintos de *PROTOCOL_TLS*, *PROTOCOL_TLS_CLIENT* y *PROTOCOL_TLS_SERVER*.

Los atributos *maximum_version*, *minimum_version* y *SSLContext.options* afectan a las versiones SSL y TLS soportadas del contexto. La implementación no evita la combinación inválida. Por ejemplo, un contexto con *OP_NO_TLSv1_2* en *options* y *maximum_version* establecido en *TLSVersion.TLSv1_2* no podrá establecer una conexión TLS 1.2.

Nuevo en la versión 3.7.

SSLContext.**minimum_version**
Igual que *SSLContext.maximum_version* excepto que es la versión más baja soportada o *TLSVersion.MINIMUM_SUPPORTED*.

Nuevo en la versión 3.7.

SSLContext.**num_tickets**
Controla el número de tickets de sesión TLS 1.3 de un contexto TLS_PROTOCOL_SERVER. El ajuste no tiene impacto en las conexiones TLS 1.0 a 1.2.

Nuevo en la versión 3.8.

SSLContext.**options**
Un número entero que representa el conjunto de opciones SSL habilitadas en este contexto. El valor por defecto es *OP_ALL*, pero se pueden especificar otras opciones como *OP_NO_SSLv2* mediante la combinación OR.

Distinto en la versión 3.6: *SSLContext.options* retorna opciones *Options*:

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

Obsoleto desde la versión 3.7: Esta opción es obsoleta desde OpenSSL 1.1.0, utilice en su lugar los nuevos *SSLContext.minimum_version* y *SSLContext.maximum_version*.

SSLContext.**post_handshake_auth**
Habilita la autenticación del cliente TLS 1.3 post-handshake. La autenticación post-handshake está deshabilitada por defecto y un servidor sólo puede solicitar un certificado de cliente TLS durante el handshake inicial. Cuando se habilita, un servidor puede solicitar un certificado de cliente TLS en cualquier momento después del handshake.

Cuando se activa en los sockets del lado del cliente, el cliente indica al servidor que soporta la autenticación post-handshake.

Cuando se activa en los sockets del lado del servidor, *SSLContext.verify_mode* debe establecerse también como *CERT_OPTIONAL* o *CERT_REQUIRED*. El intercambio real de certificados del cliente se retrasa hasta que se llama a *SSLSocket.verify_client_post_handshake()* y se realiza alguna E/S.

Nuevo en la versión 3.8.

SSLContext.**protocol**
La versión del protocolo elegida cuando se construyó el contexto. Este atributo es de sólo lectura.

SSLContext.**hostname_checks_common_name**
Si *check_hostname* vuelve a verificar el nombre común del sujeto del certificado en ausencia de una extensión de nombre alternativo del sujeto (por defecto: true).

Nuevo en la versión 3.7.

Distinto en la versión 3.10: La bandera no tuvo ningún efecto con OpenSSL antes de la versión 1.1.1k. Python 3.8.9, 3.9.3 y 3.10 incluyen soluciones para versiones anteriores.

SSLContext.**security_level**
Un número entero que representa el security level para el contexto. Este atributo es de solo lectura.

Nuevo en la versión 3.10.

SSLContext.**verify_flags**
Los indicadores para las operaciones de verificación de certificados. Se pueden establecer indicadores como *VERIFY_CRL_CHECK_LEAF* mediante la combinación OR. Por defecto, OpenSSL no requiere ni verifica las listas de revocación de certificados (CRL). Disponible sólo con la versión 0.9.8+ de openssl.

Nuevo en la versión 3.4.

Distinto en la versión 3.6: *SSLContext.verify_flags* retorna opciones *VerifyFlags*:

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

SSLContext.**verify_mode**
Si se intenta verificar los certificados de otros pares y cómo comportarse si la verificación falla. Este atributo debe ser uno de los siguientes: *CERT_NONE*, *CERT_OPTIONAL* o *CERT_REQUIRED*.

Distinto en la versión 3.6: *SSLContext.verify_mode* retorna *VerifyMode* enum:

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

18.3.4 Certificados

En general, los certificados forman parte de un sistema de clave pública/clave privada. En este sistema, a cada *principal* (que puede ser una máquina, una persona o una organización) se le asigna una clave de cifrado única de dos partes. Una parte de la clave es pública y se llama *clave pública*; la otra parte se mantiene en secreto y se llama *clave privada*. Las dos partes están relacionadas, en el sentido de que si se cifra un mensaje con una de las partes, se puede descifrar con la otra parte, y **sólo** con la otra parte.

Un certificado contiene información sobre dos sujetos. Contiene el nombre de un *sujeto*, y la clave pública del sujeto. También contiene una declaración de un segundo titular, el *emisor*, de que el sujeto es quien dice ser, y de que ésta es efectivamente la clave pública del sujeto. La declaración del emisor está firmada con su clave privada, que sólo el emisor conoce. Sin embargo, cualquiera puede verificar la declaración del emisor encontrando la clave pública del emisor, descifrando la declaración con ella y comparándola con el resto de la información del certificado. El certificado también contiene información sobre el periodo de tiempo en el que es válido. Esto se expresa en dos campos, llamados *notBefore* y *notAfter*.

En el uso de certificados en Python, un cliente o servidor puede utilizar un certificado para demostrar quién es. El otro lado de una conexión de red también puede ser requerido para producir un certificado, y ese certificado puede ser validado a la satisfacción del cliente o servidor que requiere dicha validación. El intento de conexión puede configurarse para que lance una excepción si la validación falla. La validación se realiza automáticamente, por el subyacente framework OpenSSL; la aplicación no necesita preocuparse de su mecánica. Sin embargo, la aplicación normalmente necesita proporcionar conjuntos de certificados para permitir que este proceso tenga lugar.

Python utiliza archivos para contener certificados. Deben ser formateados como «PEM» (ver **RFC 1422**), que es una forma codificada en base-64 envuelta con una línea de cabecera y una línea de pie de página:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Cadenas de certificados

Los archivos de Python que contienen certificados pueden contener una secuencia de certificados, a veces llamada *cadena de certificados*. Esta cadena debería empezar con el certificado específico para el principal que «es» el cliente o servidor, y luego el certificado para el emisor de ese certificado, y luego el certificado para el emisor de *ese* certificado, y así sucesivamente hasta llegar a un certificado que es *auto-firmado*, es decir, un certificado que tiene el mismo sujeto y emisor, a veces llamado *certificado raíz*. Los certificados sólo deben concatenarse en el archivo de certificados. Por ejemplo, supongamos que tenemos una cadena de tres certificados, desde el certificado de nuestro servidor al certificado de la autoridad de certificación que firmó nuestro certificado del servidor, hasta el certificado raíz de la agencia que emitió el certificado de la autoridad de certificación:

```
-----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-----
```

Certificados CA

Si se requiere la validación del certificado del otro lado de la conexión, se necesita proporcionar un archivo «CA certs», con las cadenas de certificados para cada emisor en el que se está dispuesto a confiar. De nuevo, este archivo sólo contiene estas cadenas concatenadas. Para la validación, Python utilizará la primera cadena que encuentre en el archivo que coincida. El archivo de certificados de la plataforma se puede utilizar llamando a *SSLContext.load_default_certs()*, esto se hace automáticamente con *create_default_context()*.

Clave y certificado combinados

A menudo la clave privada se almacena en el mismo archivo que el certificado; en este caso, sólo es necesario pasar el parámetro `certfile` a *SSLContext.load_cert_chain()* y *wrap_socket()*. Si la clave privada se almacena con el certificado, debe ir antes del primer certificado de la cadena de certificados:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Certificados auto-firmados

Si va a crear un servidor que proporcione servicios de conexión encriptada SSL, necesitará adquirir un certificado para ese servicio. Hay muchas formas de adquirir los certificados adecuados, como comprar uno a una autoridad de certificación. Otra práctica común es generar un certificado auto-firmado. La forma más sencilla de hacerlo es con el paquete OpenSSL, utilizando algo como lo siguiente:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
.............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

La desventaja de un certificado auto-firmado es que es su propio certificado raíz, y nadie más lo tendrá en su caché de certificados raíz conocidos (y de confianza).

18.3.5 Ejemplos

Pruebas de compatibilidad con SSL

Para comprobar la presencia de soporte SSL en una instalación de Python, el código del usuario debe utilizar el siguiente modismo:

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

Operación del lado del cliente

Este ejemplo crea un contexto SSL con la configuración de seguridad recomendada para los sockets del cliente, incluyendo la verificación automática de certificados:

```
>>> context = ssl.create_default_context()
```

Si prefieres ajustar la configuración de seguridad tú mismo, puedes crear un contexto desde cero (pero ten en cuenta que podrías no acertar con la configuración):

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(este fragmento asume que tu sistema operativo coloca un paquete de todos los certificados CA en `/etc/ssl/certs/ca-bundle.crt`; si no es así, obtendrá un error y tendrá que ajustar la ubicación)

El protocolo *PROTOCOL_TLS_CLIENT* configura el contexto para la validación de certificados y la verificación del hostname. *verify_mode* se establece en *CERT_REQUIRED* y *check_hostname* se establece en `True`. Todos los demás protocolos crean contextos SSL con valores predeterminados inseguros.

Cuando se utiliza el contexto para conectarse a un servidor, *CERT_REQUIRED* y *check_hostname* validan el certificado del servidor: se asegura de que el certificado del servidor se ha firmado con uno de los certificados de la CA, se comprueba que la firma es correcta y se verifican otras propiedades como la validez y la identidad del hostname:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

A continuación, puede obtener el certificado:

```
>>> cert = conn.getpeercert()
```

La inspección visual muestra que el certificado sí identifica el servicio deseado (es decir, el host HTTPS `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'),)),
```

(continué en la próxima página)

(proviene de la página anterior)

```
 '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}
```

Ahora que se ha establecido el canal SSL y se ha verificado el certificado, se puede proceder a hablar con el servidor:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

Vea la discusión sobre *Consideraciones de seguridad* más abajo.

Operación del lado del servidor

Para el funcionamiento del servidor, normalmente necesitarás tener un certificado de servidor y una clave privada, cada uno en un archivo. Primero crearás un contexto que contenga la clave y el certificado, para que los clientes puedan comprobar tu autenticidad. Entonces abrirás un socket, lo enlazarás a un puerto, llamarás a `listen()` en él, y empezarás a esperar a que los clientes se conecten:

```
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.example.com', 10023))
bindsocket.listen(5)
```

Cuando un cliente se conecta, llamarás a `accept()` en el socket para obtener el nuevo socket del otro extremo, y utilizarás el método *SSLContext.wrap_socket()* del contexto para crear un socket SSL del lado del servidor para la conexión:

```
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()
```

Entonces leerás los datos del `connstream` y harás algo con ellos hasta que hayas terminado con el cliente (o el cliente haya terminado contigo):

```
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
```

Y volver a escuchar nuevas conexiones de clientes (por supuesto, un servidor real probablemente manejaría cada conexión de cliente en un hilo separado, o pondría los sockets en modo *no-bloqueo* y usaría un bucle de eventos).

18.3.6 Notas sobre los sockets no bloqueantes

Los sockets SSL se comportan de forma ligeramente diferente a los sockets normales en modo no bloqueante. Cuando se trabaja con sockets no bloqueantes, hay varias cosas que hay que tener en cuenta:

- La mayoría de los métodos de *SSLSocket* lanzarán *SSLWantWriteError* o *SSLWantReadError* en lugar de *BlockingIOError* si una operación de E/S se bloquea. *SSLWantReadError* se lanzará si es necesaria una operación de lectura en el socket subyacente, y *SSLWantWriteError* para una operación de escritura en el socket subyacente. Tenga en cuenta que los intentos de *escribir* en un socket SSL pueden requerir *leer* del socket subyacente primero, y los intentos de *leer* del socket SSL pueden requerir una *escritura* previa en el socket subyacente.

  Distinto en la versión 3.5: En versiones anteriores de Python, el método `SSLSocket.send()` devolvía cero en lugar de lanzar *SSLWantWriteError* o *SSLWantReadError*.

- Llamar a *select()* le indica que se puede leer (o escribir) en el socket a nivel del SO, pero no implica que haya suficientes datos en la capa superior SSL. Por ejemplo, puede que sólo haya llegado una parte de una trama SSL. Por lo tanto, debe estar preparado para manejar los fallos de `SSLSocket.recv()` y `SSLSocket.send()`, y re-intentar después de otra llamada a *select()*.
- Por el contrario, dado que la capa SSL tiene su propia estructura, un socket SSL puede tener datos disponibles para leer sin que *select()* lo sepa. Por lo tanto, debería llamar primero a `SSLSocket.recv()` para drenar cualquier dato potencialmente disponible, y luego sólo bloquear en una llamada a *select()* si todavía es necesario.

  (por supuesto, se aplican disposiciones similares cuando se utilizan otras primitivas como *poll()*, o las del módulo *selectors*)
- El handshake SSL en sí mismo será no bloqueante: el método *SSLSocket.do_handshake()* tiene que ser re-intentado hasta que regrese con éxito. Aquí hay una sinopsis usando *select()* para esperar la disponibilidad del socket:

```
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

Ver también:

El módulo *asyncio* soporta *sockets SSL no bloqueantes* y proporciona una API de alto nivel. Busca eventos usando el módulo *selectors* y maneja las excepciones *SSLWantWriteError*, *SSLWantReadError* y *BlockingIOError*. También ejecuta el handshake SSL de forma asíncrona.

18.3.7 Soporte de memoria BIO

Nuevo en la versión 3.5.

Desde que se introdujo el módulo SSL en Python 2.6, la clase *SSLSocket* ha proporcionado dos áreas de funcionalidad relacionadas pero distintas:

- Manejo del protocolo SSL
- E/S de red

La API de E/S de red es idéntica a la proporcionada por *socket.socket*, de la que también hereda *SSLSocket*. Esto permite que un socket SSL sea utilizado como un reemplazo de un socket normal, haciendo que sea muy fácil añadir soporte SSL a una aplicación existente.

La combinación del manejo del protocolo SSL y la E/S de red suele funcionar bien, pero hay algunos casos en los que no es así. Un ejemplo son los frameworks de E/S asíncronos que quieren utilizar un modelo de multiplexación de E/S diferente al modelo «selección/consulta de un descriptor de archivo» (basado en la preparación) que es asumido por *socket.socket* y por las rutinas internas de E/S de socket de OpenSSL. Esto es principalmente relevante para plataformas como Windows donde este modelo no es eficiente. Para este propósito, se proporciona una variante de ámbito reducido de *SSLSocket* llamada *SSLObject*.

class `ssl.`**SSLObject**

Una variante de alcance reducido de *SSLSocket* que representa una instancia del protocolo SSL que no contiene ningún método de E/S de red. Esta clase suele ser utilizada por los autores de frameworks que quieren implementar E/S asíncrona para SSL a través de búfers de memoria.

Esta clase implementa una interfaz sobre un objeto SSL de bajo nivel como el implementado por OpenSSL. Este objeto captura el estado de una conexión SSL pero no proporciona ninguna E/S de red en sí misma. La E/S debe realizarse a través de objetos «BIO» separados que son la capa de abstracción de E/S de OpenSSL.

Esta clase no tiene un constructor público. Se debe crear una instancia *SSLObject* utilizando el método *wrap_bio()*. Este método creará la instancia *SSLObject* y la vinculará a un par de BIOs. El BIO *de entrada* se utiliza para pasar los datos de Python a la instancia del protocolo SSL, mientras que el BIO *de salida* se utiliza para pasar los datos a la inversa.

Los siguientes métodos son disponibles:

- *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()*

En comparación con *SSLSocket*, este objeto carece de las siguientes características:

- Cualquier forma de E/S de red; `recv()` y `send()` leen y escriben sólo en los búfers subyacentes de *MemoryBIO*.
- No existe la maquinaria *do_handshake_on_connect*. Siempre hay que llamar manualmente a *do_handshake()* para iniciar el handshake.
- No hay manejo de *suppress_ragged_eofs*. Todas las condiciones de fin de archivo que violan el protocolo se reportan a través de la excepción *SSLEOFError*.
- La llamada al método *unwrap()* no devuelve nada, a diferencia de lo que ocurre con un socket SSL, que devuelve el socket subyacente.
- La función callback *server_name_callback* pasada a *SSLContext.set_servername_callback()* obtendrá una instancia de *SSLObject* en lugar de una instancia de *SSLSocket* como primer parámetro.

Algunas notas relacionadas con el uso de *SSLObject*:

- Toda la E/S en un *SSLObject* es *non-blocking*. Esto significa que, por ejemplo, *read()* lanzará un *SSLWantReadError* si necesita más datos de los que la BIO entrante tiene disponibles.
- No hay una llamada a nivel de módulo `wrap_bio()` como la que hay para *wrap_socket()*. Un *SSLObject* siempre se crea a través de un *SSLContext*.

Distinto en la versión 3.7: Las instancias *SSLObject* deben crearse con *wrap_bio()*. En versiones anteriores, era posible crear instancias directamente. Esto nunca fue documentado ni soportado oficialmente.

Un SSLObject se comunica con el mundo exterior utilizando búfers de memoria. La clase *MemoryBIO* proporciona un búfer de memoria que puede ser utilizado para este propósito. Envuelve un objeto BIO (Basic IO) de memoria de OpenSSL:

class ssl.**MemoryBIO**
Un búfer de memoria que se puede utilizar para pasar datos entre Python y una instancia del protocolo SSL.

pending
Retorna el número de bytes que se encuentran actualmente en la memoria búfer.

eof
Un booleano que indica si la memoria BIO es actual en la posición de fin de archivo.

read(*n*=-*1*)
Lee hasta *n* bytes del búfer de memoria. Si *n* no se especifica o es negativo, se devuelven todos los bytes.

write(*buf*)
Escribe los bytes de *buf* en la memoria BIO. El argumento *buf* debe ser un objeto que soporte el protocolo de búfer.

El valor de retorno es el número de bytes escritos, que siempre es igual a la longitud de *buf*.

write_eof()
Escribe un marcador EOF en la memoria BIO. Después de llamar a este método, es ilegal llamar a *write()*. El atributo *eof* se convertirá en verdadero después de que se hayan leído todos los datos que hay actualmente en el búfer.

18.3.8 Sesión SSL

Nuevo en la versión 3.6.

class ssl.**SSLSession**
Objeto sesión usado por *session*.

id

time

timeout

ticket_lifetime_hint

has_ticket

18.3.9 Consideraciones de seguridad

Los mejores valores por defecto

Para el **uso en el cliente**, si no tiene ningún requisito especial para su política de seguridad, es muy recomendable que utilice la función *create_default_context()* para crear su contexto SSL. Cargará los certificados CA de confianza del sistema, habilitará la validación de certificados y la comprobación del hostname, e intentará elegir una configuración de protocolo y cifrado razonablemente segura.

Por ejemplo, así es como se utiliza la clase *smtplib.SMTP* para crear una conexión segura y de confianza con un servidor SMTP:

```
>>> 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')
```

Si se necesita un certificado de cliente para la conexión, se puede añadir con *SSLContext.load_cert_chain()*.

Por el contrario, si crea el contexto SSL llamando usted mismo al constructor *SSLContext*, no tendrá activada por defecto la validación de certificados ni la comprobación de hostname. Si lo hace, lea los párrafos siguientes para conseguir un buen nivel de seguridad.

Ajustes manuales

Verificación de certificados

Cuando se llama al constructor de *SSLContext* directamente, *CERT_NONE* es el valor por defecto. Dado que no autentifica al otro peer, puede ser inseguro, especialmente en modo cliente, donde la mayoría de las veces se quiere asegurar la autenticidad del servidor con el que se está hablando. Por lo tanto, cuando se está en modo cliente, es muy recomendable utilizar *CERT_REQUIRED*. Sin embargo, no es suficiente por sí mismo; también hay que comprobar que el certificado del servidor, que se puede obtener llamando a *SSLSocket.getpeercert()*, coincide con el servicio deseado. Para muchos protocolos y aplicaciones, el servicio puede ser identificado por el hostname; en este caso, se puede utilizar la función *match_hostname()*. Esta comprobación común se realiza automáticamente cuando *SSLContext.check_hostname* está activado.

Distinto en la versión 3.7: Las hostname matchings son ahora realizadas por OpenSSL. Python ya no utiliza *match_hostname()*.

En el modo servidor, si quiere autenticar a sus clientes utilizando la capa SSL (en lugar de utilizar un mecanismo de autenticación de nivel superior), también tendrá que especificar *CERT_REQUIRED* y comprobar de forma similar el certificado del cliente.

Versiones del protocolo

Las versiones 2 y 3 de SSL se consideran inseguras y, por lo tanto, su uso es peligroso. Si desea la máxima compatibilidad entre clientes y servidores, se recomienda utilizar *PROTOCOL_TLS_CLIENT* o *PROTOCOL_TLS_SERVER* como versión del protocolo. SSLv2 y SSLv3 están desactivados por defecto.

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.minimum_version = ssl.TLSVersion.TLSv1_3
>>> client_context.maximum_version = ssl.TLSVersion.TLSv1_3
```

The SSL context created above will only allow TLSv1.3 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.

Selección de cifrado

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.

Multiprocesamiento

Si utiliza este módulo como parte de una aplicación multiproceso (utilizando, por ejemplo, los módulos *multiprocessing* o *concurrent.futures*), tenga en cuenta que el generador de números aleatorios interno de OpenSSL no maneja adecuadamente los procesos bifurcados. Las aplicaciones deben cambiar el estado del PRNG del proceso padre si utilizan cualquier función de SSL con *os.fork()*. Cualquier llamada exitosa de *RAND_add()*, *RAND_bytes()* o *RAND_pseudo_bytes()* es suficiente.

18.3.10 TLS 1.3

Nuevo en la versión 3.7.

Python tiene soporte provisional y experimental para TLS 1.3 con OpenSSL 1.1.1. El nuevo protocolo se comporta de forma ligeramente diferente a la versión anterior de TLS/SSL. Algunas de las nuevas características de TLS 1.3 aún no están disponibles.

- TLS 1.3 utiliza un conjunto disjunto de suites de cifrado. Todas las suites de cifrado AES-GCM y ChaCha20 están habilitadas por defecto. El método *SSLContext.set_ciphers()* aún no puede habilitar o deshabilitar ningún cifrado de TLS 1.3, pero *SSLContext.get_ciphers()* los devuelve.
- Los tickets de sesión ya no se envían como parte del handshake inicial y se manejan de forma diferente. *SSLSocket.session* y *SSLSession* no son compatibles con TLS 1.3.
- Los certificados del lado del cliente ya no se verifican durante el handshake inicial. Un servidor puede solicitar un certificado en cualquier momento. Los clientes procesan las solicitudes de certificados mientras envían o reciben datos de la aplicación desde el servidor.
- Las funciones de TLS 1.3, como los datos anticipados, la solicitud de certificado de cliente TLS diferida, la configuración del algoritmo de firma y la repetición de claves, aún no son compatibles.

Ver también:

Clase *socket.socket* Documentación de la clase *socket* subyacente

SSL/TLS Strong Encryption: An Introduction Introducción de la documentación del servidor HTTP Apache

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 (D 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: Siguiente **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 informs epoll about the expected number of events to be registered. It must be positive, or `-1` to use the default. It is only used on older systems where `epoll_create1()` is not available; otherwise it has no effect (though its value is still checked).

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 lanzará `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*]`)`
Esta es una interfaz sencilla para la llamada al sistema Unix `select()`. Los primeros tres argumentos son iterables de "objetos en espera": enteros que representan descriptores de archivos u objetos con un método sin parámetros llamado *`fileno()`* que retorna un entero de este tipo:

- *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)

Se permiten iterables vacíos, pero la aceptación de tres iterables vacíos depende de la plataforma. (Se sabe que funciona en Unix pero no en Windows). El argumento opcional *timeout* especifica un tiempo de espera como un número de punto flotante en segundos. Cuando se omite el argumento *timeout*, la función se bloquea hasta que al menos un descriptor de archivo esté listo. Un valor de tiempo de espera de cero especifica un sondeo y nunca se bloquea.

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.

Entre los tipos de objetos aceptables en los iterables se encuentran Python *objetos de archivo* (por ejemplo, `sys.stdin`, u objetos retornados por *`open()`* o *`os.popen()`*), objetos de socket retornados por *`socket.socket()`*. También puede definir una clase *wrapper* usted mismo, siempre que tenga un método *`fileno()`* apropiado (que realmente retorne un descriptor de archivo, no solo un entero aleatorio).

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(descriptor de archivo más alto) y poll() es O (número de descriptores de archivo), /dev/poll es O(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 a *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*])
Polls the set of registered file descriptors, and returns a possibly empty list containing (fd, event) 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor — POLLIN for waiting input, POLLOUT to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, -1, or *None*, the call will block until there is an event for this poll object.

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

Constante	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 *Edge Trigger*, el valor predeterminado es *Level Trigger*
EPOLLONESHOT	Establece el comportamiento en *one-shot*. 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.
EPOLLRDNORM	Equivalente a EPOLLIN
EPOLLRDBAND	La banda de datos de prioridad se puede leer.
EPOLLWRNORM	Equivalente a EPOLLOUT
EPOLLWRBAND	Se pueden escribir datos de prioridad.
EPOLLMSG	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.

Distinto en la versión 3.9: El método ya no ignora el error *EBADF*.

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 descriptores 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 (descriptor de archivo más alto), mientras que `poll()` es O(número de descriptores 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
`POLLIN`	Hay datos para leer
`POLLPRI`	Hay datos urgentes para leer
`POLLOUT`	Lista para la salida: la escritura no bloqueará
`POLLERR`	Condición de error de algún tipo
`POLLHUP`	Colgado
`POLLRDHUP`	Socket de flujo de conexión cerrada por pares, o apagado escribiendo la mitad de la conexión
`POLLNVAL`	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*])

Polls the set of registered file descriptors, and returns a possibly empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor — `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, negative, or *None*, the call will block until there is an event for this poll object.

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
`KQ_FILTER_READ`	Toma un descriptor y retorna cada vez que hay datos disponibles para leer
`KQ_FILTER_WRITE`	Toma un descriptor y retorna cada vez que hay datos disponibles para escribir
`KQ_FILTER_AIO`	Solicitudes de AIO
`KQ_FILTER_VNODE`	Retorna cuando ocurre uno o más de los eventos solicitados observados en *fflag*
`KQ_FILTER_PROC`	Vigila los eventos en un id de proceso
`KQ_FILTER_NETDEV`	Vigila los eventos en un dispositivo de red [no disponible en macOS]
`KQ_FILTER_SIGNAL`	Retorna cada vez que la señal observada se entrega al proceso
`KQ_FILTER_TIMER`	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	Disablesevent
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.**fflags**
Filtrar flags específicas.

KQ_FILTER_READ y KQ_FILTER_WRITE bandera de filtro:

Constante	Significado
KQ_NOTE_LOWAT	marca de agua baja de un *socket buffer*

KQ_FILTER_VNODE bandera de filtro:

Constante	Significado
KQ_NOTE_DELETE	*unlink()* 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 (*exited*)
KQ_NOTE_FORK	el proceso ha llamado a *fork()*
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 *fork()*
KQ_NOTE_CHILD	retornado en el proceso hijo para *NOTE_TRACK*
KQ_NOTE_TRACKERR	incapaz de adjuntar a un proceso hijo

Indicadores de filtro KQ_FILTER_NETDEV (no disponible en macOS):

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 `signal` — Establece gestores para eventos asíncronos

Source code: Lib/signal.py

Este módulo proporciona mecanismos para usar gestores de señales en Python.

18.6.1 Reglas generales

La función *`signal.signal()`* permite definir gestores personalizados que serán ejecutados cuando una señal es recibida. Un pequeño número de gestores por defecto son instalados: *`SIGPIPE`* es ignorada (por lo que los errores de escritura en tuberías y sockets se pueden informar como excepciones ordinarias de Python) y *`SIGINT`* es trasladada en una excepción *`KeyboardInterrupt`* si el proceso padre no lo ha cambiado.

El gestor para una señal en particular, una vez establecido, continua instalado hasta que se resetea explícitamente (Python emula el estilo de interfaz BSD independientemente de la implementación subyacente), con la excepción del gestor para *`SIGCHLD`*, que sigue la implementación subyacente.

Ejecución de los gestores de señales de Python

Un gestor de señales de Python no se ejecuta dentro del gestor de señales de bajo nivel (C). En vez de eso, el gestor de señales de bajo nivel establece una señal que le dice al *virtual machine* que ejecute la correspondiente señal del gestor de Python en una posición posterior (por ejemplo en la próxima instrucción *bytecode*). Esto tiene consecuencias:

- Tiene poco sentido detectar errores sincrónicos como *`SIGFPE`* o *`SIGSEGV`* que son causados por una operación no válida en código C. Python retornará desde el gestor de señales a código C, que es probable que extienda la misma señal otra vez, ocasionando que Python se cuelgue aparentemente. Desde Python 3.3 en adelante, puedes usar el módulo *`faulthandler`* para reportar errores síncronos.
- Un cálculo de larga duración implementado exclusivamente en C (como una coincidencia de expresiones regulares en un gran cuerpo de texto) puede funcionar interrumpidamente durante una cantidad arbitraria de tiempo, independientemente de las señales recibidas. Los gestores de señales de Python serán llamados cuando el cálculo finalice.
- If the handler raises an exception, it will be raised «out of thin air» in the main thread. See the *note below* for a discussion.

Señales e hilos

Los gestores de señales de Python se ejecutan siempre en el hilo principal de Python, incluso si la señal fue recibida desde otro hilo. Esto significa que las señales no pueden ser usadas como un medio de comunicación entre hilos. Puedes usar las primitivas de sincronización desde el módulo *threading* en su lugar.

Además, solo el hilo principal puede configurar un nuevo gestor de señal.

18.6.2 Contenidos del módulo

Distinto en la versión 3.5: señal (SIG*), gestor (*SIG_DFL*, *SIG_IGN*) y "sigmask" (*SIG_BLOCK*, *SIG_UNBLOCK*, *SIG_SETMASK*) las clases relacionadas abajo se cambian en las funciones *enums*. *getsignal()*, *pthread_sigmask()*, *sigpending()* y *sigwait()* que retornan *enums* que pueden ser leídas por humanos.

Las variables definidas en el módulo *signal* son:

signal.**SIG_DFL**
: Ésta es una de las dos opciones estándar de manejo de señales; simplemente realizará la función predeterminada para la señal. Por ejemplo, en la mayoría de los sistemas, la acción predeterminada para `SIGQUIT` es volcar el núcleo y salir, mientras que la acción predeterminada para *SIGCHLD* es simplemente ignorarlo.

signal.**SIG_IGN**
: Este es otro manejador de señales estándar, que simplemente ignorará la señal dada.

signal.**SIGABRT**
: Abortar señal de *abort(3)*.

signal.**SIGALRM**
: Señal de temporizador de *alarm(2)*.

  Disponibilidad: Unix.

signal.**SIGBREAK**
: Interrumpir desde el teclado (CTRL + BREAK).

  Disponibilidad: Windows.

signal.**SIGBUS**
: Error de bus (mal acceso a la memoria).

  Disponibilidad: Unix.

signal.**SIGCHLD**
: El proceso hijo se detuvo o terminó.

  Disponibilidad: Unix.

signal.**SIGCLD**
: Alias para *SIGCHLD*.

signal.**SIGCONT**
: Continuar el proceso si está detenido actualmente

  Disponibilidad: Unix.

signal.**SIGFPE**
: Excepción de coma flotante. Por ejemplo, división por cero.

  Ver también:

  ZeroDivisionError se genera cuando el segundo argumento de una operación de división o módulo es cero.

signal.**SIGHUP**
: Se detectó un bloqueo en el terminal de control o muerte del proceso de control.

  Disponibilidad: Unix.

signal.**SIGILL**
Instrucción ilegal.

signal.**SIGINT**
Interrumpir desde el teclado (CTRL + C).

La acción predeterminada es generar *KeyboardInterrupt*.

signal.**SIGKILL**
Señal de muerte.

No se puede detectar, bloquear ni ignorar.

Disponibilidad: Unix.

signal.**SIGPIPE**
Tubería rota: escriba en la tubería sin lectores.

La acción predeterminada es ignorar la señal.

Disponibilidad: Unix.

signal.**SIGSEGV**
Fallo de segmentación: referencia de memoria no válida.

signal.**SIGTERM**
Señal de terminación.

signal.**SIGUSR1**
Señal definida por el usuario 1.

Disponibilidad: Unix.

signal.**SIGUSR2**
Señal definida por el usuario 2.

Disponibilidad: Unix.

signal.**SIGWINCH**
Señal de cambio de tamaño de ventana.

Disponibilidad: Unix.

SIG*
Todos los números de señal se definen simbólicamente. Por ejemplo, la señal de colgar se define como *signal.SIGHUP*; los nombres de las variables son idénticos a los nombres utilizados en los programas C, como se encuentran en <signal.h>. La página de manual de Unix para "signal()" enumera las señales existentes (en algunos sistemas esto es *signal(2) `, en otros la lista está en :manpage:`signal(7)*). Tenga en cuenta que no todos los sistemas definen el mismo conjunto de nombres de señales; Este módulo solo define los nombres definidos por el sistema.

signal.**CTRL_C_EVENT**
La señal correspondiente al evento de pulsación de tecla Ctrl + C. Esta señal solo se puede utilizar con *os.kill()*.

Disponibilidad: Windows.

Nuevo en la versión 3.2.

signal.**CTRL_BREAK_EVENT**
La señal correspondiente al evento de pulsación de tecla Ctrl + Break. Esta señal solo se puede utilizar con *os.kill()*.

Disponibilidad: Windows.

Nuevo en la versión 3.2.

signal.**NSIG**
Uno más que el número de señal más alto.

signal.**ITIMER_REAL**
Reduce el temporizador de intervalo en tiempo real y entrega *SIGALRM* al vencimiento.

signal.**ITIMER_VIRTUAL**
Disminuye el temporizador de intervalo solo cuando el proceso se está ejecutando y entrega SIGVTALRM al vencimiento.

signal.**ITIMER_PROF**
Disminuye el temporizador de intervalo tanto cuando se ejecuta el proceso como cuando el sistema se está ejecutando en nombre del proceso. Junto con ITIMER_VIRTUAL, este temporizador generalmente se usa para perfilar el tiempo que pasa la aplicación en el espacio del usuario y del kernel. SIGPROF se entrega al vencimiento.

signal.**SIG_BLOCK**
Un valor posible para el parámetro *how* para *pthread_sigmask()* que indica que las señales deben bloquearse.

Nuevo en la versión 3.3.

signal.**SIG_UNBLOCK**
Un valor posible para el parámetro *how* para *pthread_sigmask()* que indica que las señales deben desbloquearse.

Nuevo en la versión 3.3.

signal.**SIG_SETMASK**
Un valor posible para el parámetro *how* para *pthread_sigmask()* que indica que la máscara de señal debe ser reemplazada.

Nuevo en la versión 3.3.

El módulo *signal* define una excepción:

exception signal.**ItimerError**
Se genera para señalar un error de la implementación subyacente *setitimer()* o *getitimer()*. Espere este error si se pasa un temporizador de intervalo no válido o un tiempo negativo a *setitimer()*. Este error es un subtipo de *OSError*.

Nuevo en la versión 3.3: Este error solía ser un subtipo de *IOError*, que ahora es un alias de *OSError*.

El módulo *signal* define las siguientes funciones:

signal.**alarm**(*time*)
Si *time* es distinto de cero, esta función solicita que se envíe una señal *SIGALRM* al proceso en *time* segundos. Se cancela cualquier alarma programada previamente (solo se puede programar una alarma en cualquier momento). El valor retornado es entonces el número de segundos antes de que se entregara cualquier alarma previamente configurada. Si *time* es cero, no se programa ninguna alarma y se cancela cualquier alarma programada. Si el valor de retorno es cero, no hay ninguna alarma programada actualmente.

Disponibilidad: Unix. Consulte la página de manual *alarm(2)* para obtener más información.

signal.**getsignal**(*signalnum*)
Retorna el manejador de señales actual para la señal *signalnum*. El valor retornado puede ser un objeto de Python invocable o uno de los valores especiales *signal.SIG_IGN*, *signal.SIG_DFL* o *None*. Aquí, *signal.SIG_IGN* significa que la señal fue previamente ignorada, *signal.SIG_DFL* significa que la forma predeterminada de manejar la señal estaba en uso anteriormente, y el gestor de señales no se instaló desde Python.

signal.**strsignal**(*signalnum*)
Retorna la descripción del sistema de la señal *signalnum*, como «Interrupción», «Fallo de segmentación», etc. Retorna *None* si no se reconoce la señal.

Nuevo en la versión 3.8.

signal.**valid_signals**()
Retorna el conjunto de números de señal válidos en esta plataforma. Esto puede ser menor que `rango(1, NSIG)` si el sistema reserva algunas señales para uso interno.

Nuevo en la versión 3.8.

signal.**pause**()
Hacer que el proceso duerma hasta que se reciba una señal; entonces se llamará al manejador apropiado. No retorna nada.

Disponibilidad: Unix. Consulte la página man *signal(2)* para obtener más información.

Vea también *sigwait()*, *sigwaitinfo()*, *sigtimedwait()* y *sigpending()*.

signal.**raise_signal**(*signum*)
Envía una señal al proceso de llamada. No retorna nada.

Nuevo en la versión 3.8.

signal.**pidfd_send_signal**(*pidfd*, *sig*, *siginfo=None*, *flags=0*)
Envía la señal *sig* al proceso referido por el descriptor de archivo *pidfd*. Python actualmente no soporta el parámetro *siginfo*; éste debe ser None. El argumento *flags* está proveído para extensiones futuras; no hay valores actualmente definidos para *flags*.

Para más información vea la página de manual *pidfd_send_signal(2)*.

Disponibilidad: Linux 5.1+

Nuevo en la versión 3.9.

signal.**pthread_kill**(*thread_id*, *signalnum*)
Envíe la señal *signalnum* al hilo *thread_id*, otro hilo en el mismo proceso que el llamador. El hilo de destino puede ejecutar cualquier código (Python o no). Sin embargo, si el hilo de destino está ejecutando el intérprete de Python, los manejadores de señales de Python serán *ejecutados por el hilo principal*. Por lo tanto, el único punto de enviar una señal a un hilo de Python en particular sería forzar que una llamada al sistema en ejecución falle con *InterruptedError*.

Utilice *threading.get_ident()* o el atributo *ident* de los objetos *threading.Thread* para obtener un valor adecuado para *thread_id*.

Si *signalnum* es 0, no se envía ninguna señal, pero se sigue realizando la comprobación de errores; esto se puede usar para verificar si el hilo de destino aún se está ejecutando.

Genera un *evento de auditoría* signal.pthread_kill con argumentos thread_id, signalnum.

Disponibilidad: Unix. Consulte la página del manual *pthread_kill(3)* para obtener más información.

Vea también *os.kill()*.

Nuevo en la versión 3.3.

signal.**pthread_sigmask**(*how*, *mask*)
Busca o cambia la máscara de señal del hilo de llamada. La máscara de señal es el conjunto de señales cuya entrega está actualmente bloqueada para la persona que llama. Retorna la máscara de señal anterior como un conjunto de señales.

El comportamiento de la llamada depende del valor de *how*, como sigue.

- *SIG_BLOCK*: El conjunto de señales bloqueadas es la unión del conjunto actual y el argumento *mask*.
- *SIG_UNBLOCK*: Las señales en *mask* se eliminan del conjunto actual de señales bloqueadas. Está permitido intentar desbloquear una señal que no esté bloqueada.
- *SIG_SETMASK*: El conjunto de señales bloqueadas se establece en el argumento *mask*.

mask es un conjunto de números de señales (por ejemplo, {*signal.SIGINT*, *signal.SIGTERM*}). Utilice *valid_signals()* para una máscara completa que incluya todas las señales.

Por ejemplo, signal.pthread_sigmask(signal.SIG_BLOCK, []) lee la máscara de señal del hilo de llamada.

SIGKILL y SIGSTOP no se pueden bloquear.

Disponibilidd: Unix. Consulte la página de manual *sigprocmask(2)* y *pthread_sigmask(3)* para obtener más información.

Vea también *pause()*, *sigpending()* y *sigwait()*.

Nuevo en la versión 3.3.

signal.**setitimer**(*which*, *seconds*, *interval=0.0*)
Establece el temporizador de intervalo dado (uno de *signal.ITIMER_REAL*, *signal.ITIMER_VIRTUAL* o *signal.ITIMER_PROF*) especificado por *which* disparar después de *seconds* (se acepta el números de punto flotante, diferente de *alarm()*) y luego cada *interval* segundos (si *interval* es distinto de cero). El temporizador de intervalo especificado por *which* se puede borrar estableciendo *seconds* en cero.

Cuando se dispara un temporizador de intervalos, se envía una señal al proceso. La señal enviada depende del temporizador que se utilice; *signal.ITIMER_REAL* entregará *SIGALRM*, *signal.ITIMER_VIRTUAL* envía SIGVTALRM, y *signal.ITIMER_PROF* entregará SIGPROF.

Los valores antiguos se retornan como una tupla: (retraso, intervalo).

Si intenta pasar un temporizador de intervalo no válido, se producirá un *ItimerError*.

Disponibilidad: Unix.

signal.**getitimer**(*which*)
Retorna el valor actual de un temporizador de intervalo dado especificado por *which*.

Disponibilidad: Unix.

signal.**set_wakeup_fd**(*fd*, *, *warn_on_full_buffer=True*)
Establezca el descriptor del archivo de activación en *fd*. Cuando se recibe una señal, el número de la señal se escribe como un solo byte en el fd. Esto puede ser utilizado por una biblioteca para despertar una encuesta o seleccionar una llamada, permitiendo que la señal se procese por completo.

Se retorna el antiguo fd de activación (o -1 si la activación del descriptor de archivo no estaba habilitada). Si *fd* es -1, la activación del descriptor de archivo está deshabilitada. Si no es -1, *fd* debe ser sin bloqueo. Depende de la biblioteca eliminar los bytes de *fd* antes de llamar a poll o seleccionar nuevamente.

Cuando los hilos están habilitados, esta función solo se puede llamar desde *el subproceso principal del intérprete principal*; intentar llamarlo desde otros hilos hará que se lance una excepción *ValueError*.

Hay dos formas habituales de utilizar esta función. En ambos enfoques, usa el fd para despertarse cuando llega una señal, pero luego difieren en cómo determinan *which* señal o señales han llegado.

En el primer enfoque, leemos los datos del búfer de fd, y los valores de bytes le dan los números de señal. Esto es simple, pero en casos raros puede surgir un problema: generalmente el fd tendrá una cantidad limitada de espacio en el búfer, y si llegan demasiadas señales demasiado rápido, entonces el búfer puede llenarse y algunas señales pueden perderse. Si usa este enfoque, entonces debe configurar `warn_on_full_buffer = True`, que al menos causará que se imprima una advertencia en stderr cuando se pierdan las señales.

En el segundo enfoque, usamos el wakeup fd *solo* para wakeups e ignoramos los valores de bytes reales. En este caso, lo único que nos importa es si el búfer de fd está vacío o no; un búfer lleno no indica ningún problema. Si utiliza este enfoque, debe configurar `warn_on_full_buffer = False`, para que sus usuarios no se confundan con mensajes de advertencia falsos.

Distinto en la versión 3.5: En Windows, la función ahora también admite identificadores de socket.

Distinto en la versión 3.7: Se agregó el parámetro `warn_on_full_buffer`.

signal.**siginterrupt**(*signalnum*, *flag*)
Cambiar el comportamiento de reinicio de la llamada al sistema: si *flag* es *False*, las llamadas al sistema se reiniciarán cuando las interrumpa la señal *signalnum*, de lo contrario, las llamadas al sistema se interrumpirán. No retorna nada.

Disponibilidad: Unix. Consulte la página man *siginterrupt(3)* para obtener más información.

Tenga en cuenta que la instalación de un gestor de señales con *signal()* restablecerá el comportamiento de reinicio a interrumpible llamando implícitamente a `siginterrupt()` con un valor de *flag* verdadero para la señal dada.

signal.**signal**(*signalnum*, *handler*)
Establece el gestor de la señal *signalnum* en la función *handler* (manejador). *handler* puede ser un objeto de Python invocable que toma dos argumentos (ver más abajo), o uno de los valores especiales *signal.SIG_IGN* o *signal.SIG_DFL*. Se retornará el manejador de señales anterior (vea la descripción de *getsignal()* arriba). (Consulte la página del manual de Unix *signal(2)* para obtener más información).

Cuando los hilos están habilitados, esta función solo se puede llamar desde *el subproceso principal del intérprete principal*; intentar llamarlo desde otros hilos hará que se lance una excepción *ValueError*.

El *handler* se llama con dos argumentos: el número de señal y el marco de pila actual (`None` o un objeto marco; para obtener una descripción de los objetos marco, consulte la descripción en la jerarquía de tipos o vea las descripciones de los atributos en el módulo *inspect*).

En Windows, *signal()* solo se puede llamar con *SIGABRT*, *SIGFPE*, *SIGILL*, *SIGINT*, *SIGSEGV*, *SIGTERM*, o *SIGBREAK*. Un *ValueError* se lanzará en cualquier otro caso. Tenga en cuenta que no todos los sistemas definen el mismo conjunto de nombres de señales; un *AttributeError* se lanzará si un nombre de señal no está definido como constante de nivel de módulo `SIG*`.

signal.**sigpending**()
Examine el conjunto de señales que están pendientes de entrega al hilo de llamada (es decir, las señales que se han generado mientras estaban bloqueadas). Retorna el conjunto de señales pendientes.

Disponibilidad: Unix. Consulte la página de manual *sigpending(2)* para obtener más información.

Vea también *pause()*, *pthread_sigmask()* y *sigwait()*.

Nuevo en la versión 3.3.

signal.**sigwait**(*sigset*)
Suspende la ejecución del hilo de llamada hasta la entrega de una de las señales especificadas en el conjunto de señales *sigset*. La función acepta la señal (la elimina de la lista pendiente de señales) y retorna el número de señal.

Disponibilidad: Unix. Consulte la página man *sigwait(3)* para obtener más información.

Vea también *pause()*, *pthread_sigmask()*, *sigpending()*, *sigwaitinfo()* y *sigtimedwait()*.

Nuevo en la versión 3.3.

signal.**sigwaitinfo**(*sigset*)
Suspende la ejecución del hilo de llamada hasta la entrega de una de las señales especificadas en el conjunto de señales *sigset*. La función acepta la señal y la elimina de la lista de señales pendientes. Si una de las señales en *sigset* ya está pendiente para el hilo de llamada, la función regresará inmediatamente con información sobre esa señal. No se llama al gestor de señales para la señal enviada. La función genera un *InterruptedError* si es interrumpida por una señal que no está en *sigset*.

El valor de retorno es un objeto que representa los datos contenidos en la estructura `siginfo_t`, a saber: `si_signo`, `si_code`, `si_errno`, `si_pid`, `si_uid`, `si_status`, `si_band`.

Disponibilidad: Unix. Consulte la página man *sigwaitinfo(2)* para obtener más información.

Vea también *pause()*, *sigwait()* y *sigtimedwait()*.

Nuevo en la versión 3.3.

Distinto en la versión 3.5: La función ahora se vuelve a intentar si es interrumpida por una señal que no está en *sigset* y el manejador de señales no genera una excepción (ver **PEP 475** para la justificación).

signal.**sigtimedwait**(*sigset*, *timeout*)
Como *sigwaitinfo()*, pero toma un argumento *timeout* adicional que especifica un tiempo de espera. Si *timeout* se especifica como `0`, se realiza una encuesta. Retorna *None* si se agota el tiempo de espera.

Disponibilidad: Unix. Consulte la página de manual *sigtimedwait(2)* para obtener más información.

Vea también *pause()*, *sigwait()* y *sigwaitinfo()*.

Nuevo en la versión 3.3.

Distinto en la versión 3.5: La función ahora se reintenta con el *timeout* recalculado si se interrumpe por una señal que no está en *sigset* y el manejador de señales no genera una excepción (ver **PEP 475** para la justificación).

18.6.3 Ejemplo

Aquí hay un programa de ejemplo mínimo. Utiliza la función *alarm()* para limitar el tiempo de espera para abrir un archivo; esto es útil si el archivo es para un dispositivo serial que puede no estar encendido, lo que normalmente haría que *os.open()* se cuelgue indefinidamente. La solución es configurar una alarma de 5 segundos antes de abrir el archivo; si la operación lleva demasiado tiempo, se enviará la señal de alarma y el gestor genera una excepción.

```
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
```

18.6.4 Nota sobre SIGPIPE

Canalizar la salida de su programa a herramientas como *head(1)* hará que se envíe una señal *SIGPIPE* a su proceso cuando el receptor de su salida estándar se cierre antes. Esto da como resultado una excepción como `BrokenPipeError: [Errno 32] Broken pipe`. Para manejar este caso, envuelva su punto de entrada para detectar esta excepción de la siguiente manera:

```
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 whenever any socket connection is interrupted while your program is still writing to it.

18.6.5 Note on Signal Handlers and Exceptions

If a signal handler raises an exception, the exception will be propagated to the main thread and may be raised after any *bytecode* instruction. Most notably, a *KeyboardInterrupt* may appear at any point during execution. Most Python code, including the standard library, cannot be made robust against this, and so a *KeyboardInterrupt* (or any other exception resulting from a signal handler) may on rare occasions put the program in an unexpected state.

To illustrate this issue, consider the following code:

```
class SpamContext:
    def __init__(self):
        self.lock = threading.Lock()

    def __enter__(self):
        # If KeyboardInterrupt occurs here, everything is fine
        self.lock.acquire()
        # If KeyboardInterrupt occurs here, __exit__ will not be called
        ...
        # KeyboardInterrupt could occur just before the function returns

    def __exit__(self, exc_type, exc_val, exc_tb):
        ...
        self.lock.release()
```

For many programs, especially those that merely want to exit on *KeyboardInterrupt*, this is not a problem, but applications that are complex or require high reliability should avoid raising exceptions from signal handlers. They should also avoid catching *KeyboardInterrupt* as a means of gracefully shutting down. Instead, they should install their own *SIGINT* handler. Below is an example of an HTTP server that avoids *KeyboardInterrupt*:

```
import signal
import socket
from selectors import DefaultSelector, EVENT_READ
from http.server import HTTPServer, SimpleHTTPRequestHandler

interrupt_read, interrupt_write = socket.socketpair()

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    interrupt_write.send(b'\0')
signal.signal(signal.SIGINT, handler)

def serve_forever(httpd):
    sel = DefaultSelector()
    sel.register(interrupt_read, EVENT_READ)
    sel.register(httpd, EVENT_READ)

    while True:
        for key, _ in sel.select():
            if key.fileobj == interrupt_read:
                interrupt_read.recv(1)
                return
            if key.fileobj == httpd:
                httpd.handle_request()

print("Serving on port 8000")
httpd = HTTPServer(('', 8000), SimpleHTTPRequestHandler)
serve_forever(httpd)
print("Shutdown...")
```

18.7 mmap — Soporte de archivos mapeados en memoria

Los objetos de archivos mapeados en memoria se comportan como objetos de tipo *bytearray* y *objetos archivo*. Puedes usar objetos mmap en la mayoría de lugares donde se esperen objetos de tipo *bytearray*; por ejemplo, puedes usar el módulo *re* para buscar entre un archivo mapeado en memoria. También puedes cambiar un solo byte al hacer `obj[index]==97`, o cambiar una subsecuencia al asignarle una rebanada: `obj[i1:i2] = b'...'`. También puedes leer y escribir datos que empiezan en la posición del archivo actual, y utilizar el método `seek()` a través del archivo para cambiar la posición actual.

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 `ACCES_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 `ACCES_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*. Algunos tipos tienen posibles banderas adicionales con la lista completa en *MAP_* constants*.

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.

To ensure validity of the created memory mapping the file specified by the descriptor *fileno* is internally automatically synchronized with the physical backing store on macOS.

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!")

pid = os.fork()

if pid == 0:  # In a child process
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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.

madvise(*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 `madvise()`.

  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, comenzando en la posición actual del archivo y hasta la siguiente nueva línea. La posición del archivo se actualiza para apuntar después de los bytes que se retornaron.

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 un 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 números 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 ACCES_READ, entonces escribirlo hará que se lance la excepción *TypeError*.

18.7.1 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**
mmap.**MADV_FREE_REUSABLE**

mmap.**MADV_FREE_REUSE**
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.

18.7.2 Constantes MAP_*

mmap.**MAP_SHARED**
mmap.**MAP_PRIVATE**
mmap.**MAP_DENYWRITE**
mmap.**MAP_EXECUTABLE**
mmap.**MAP_ANON**
mmap.**MAP_ANONYMOUS**
mmap.**MAP_POPULATE**
Estas son las diferentes banderas que se pueden pasar al método *mmap.mmap()*. Note que algunas opciones puedan no estar presentes en todos los sistemas.

Distinto en la versión 3.10: Se añadió la constante MAP_POPULATE.

CAPÍTULO 19

Manejo de Datos de Internet

Este capítulo describe los módulos que admiten el manejo de formatos de datos comúnmente usados en Internet.

19.1 `email` — Un paquete de manejo de correo electrónico y MIME

Código fuente Lib/email/__init__.py

El paquete *email* es una biblioteca para administrar mensajes de correo electrónico. Específicamente *no* está diseñado para realizar cualquier envío de mensajes de correo electrónico a SMTP (**RFC 2821**), NNTP u otros servidores; esas son funciones de módulos como *smtplib* y *nntplib*. El paquete *email* intenta ser lo más compatible con RFC, admitiendo **RFC 5322** y **RFC 6532**, así como RFC relacionados con MIME como **RFC 2045**, **RFC 2046**, **RFC 2047**, **RFC 2183** y **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 las cabeceras, para acceder al cuerpo del mensaje, y para crear o modificar la estructura del mensaje.

Un mensaje de correo electrónico 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 cabeceras emparejadas con una carga útil que representa al cuerpo del mensaje **RFC 5322**, que podría ser una lista de objetos

[1] Añadido originalmente en la versión 3.4 como un *módulo provisional*. La documentación para la clase heredada *message* se movió a *email.message.Message: Representar un mensaje de correo electrónico usando la API compat32*.

sub-EmailMessage. Además de los métodos normales de diccionario para acceder a las cabeceras y valores, tiene métodos para acceder a información especializada desde las cabeceras (por ejemplo, el tipo de contenido MIME), para operar en la carga útil, generar una versión serializada del mensaje, y recorrer recursivamente el árbol de objetos.

La interfaz tipo diccionario de *EmailMessage* está indexada por los nombres de las cabeceras, que deberán ser valores ASCII. Los valores del diccionario son cadenas con algunos métodos adicionales. Las cabeceras son almacenadas y retornadas de forma sensible a mayúsculas, pero los nombres de los campos son comparados sin discriminar entre mayúsculas. A diferencia de un dict real, las llaves están ordenadas y pueden estar duplicadas. Se proveen métodos adicionales para trabajar con cabeceras que tienen llaves duplicadas.

La carga útil 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 la política 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, la cabecera está incluida 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 la política especificada 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* de la política .

__str__()

Equivalente a as_string(policy=self.policy.clone(utf8=True)). Permite str(msg) para producir una cadena 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, la cabecera es incluida en la cadena de caracteres retornada. El valor predeterminado de *unixform* es False. El argumento *policy* puede ser usado para sobreescribir el valor predeterminado obtenido de la instancia del mensaje. Esto puede ser usado para controlar parte del formato producido por el método, ya que la política especificada 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 la carga útil del mensaje es una lista de objetos de sub-*EmailMessage*, de otra manera retorna `False`. Cuando *is_multipart()* retorna `False`, la carga útil deberá ser un objeto cadena de caracteres (que podría ser una carga útil binaria codificada 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 *unixform*, que debería ser una cadena de caracteres. (Consulte *mboxMessage* para una descripción de esta cabecera)

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 a la cabecera 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 cabeceras duplicadas. Además, en los diccionarios no hay un orden garantizado para las claves retornadas por *keys()*, pero en un objeto *EmailMessage*, las cabeceras siempre regresan en orden de aparición en el mensaje original, o en el que fueron agregadas luego. Cualquier cabecera borrada y vuelta 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 cabecera presente en el mensaje no se incluye en la interfaz de mapeo.

__len__()
Retorna el número total de cabeceras, incluidas las duplicadas.

__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 de la cabecera nombrada. *name* no incluye el separador de dos puntos, ":". Si la cabecera se pierde, regresa `None`, un *KeyError* no se genera nunca.

Tenga en cuenta que si el campo nombrado aparece más de una vez en la cabecera del mensaje, esa cantidad de veces el valor regresado será indefinido. Use el método *get_all()* para obtener los valores de todas las cabeceras existentes llamadas *name*.

Usando el *standard non-"compat32"*, el valor regresado es una instancia de una subclase de *email.headerregistry.BaseHeader*.

__setitem__(*name*, *val*)
Agrega una cabecera al mensaje con un campo "nombre" y un valor "val". El campo se agrega al final de las cabeceras existentes en el mensaje.

Tenga en cuenta que esto no sobrescribe ni borra ninguna cabecera con el mismo nombre. Si quiere asegurarse de que la nueva cabecera es la única en el mensaje con el campo "nombre", borre el campo primero, por ejemplo:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

Si el `policy` define ciertas cabeceras para ser únicos(como lo hace el *standard*), este método puede generar un *`ValueError`* cuando se intenta asignar un valor a una cabecera preexistente. Este comportamiento es intencional por consistencia, pero no dependa de ello, ya que podemos optar por hacer que tales asignaciones eliminen la cabecera en el futuro.

`__delitem__`(*name*)
Elimine todas las apariciones del campo "nombre" de las cabeceras del mensaje. No se genera ninguna excepción si el campo nombrado no está presente en las cabeceras.

`keys`()
Retorna una lista de los nombres de todos los campos de la cabecera del mensaje.

`values`()
Retorna una lista de todos los valores de los campos del mensaje.

`items`()
Retorna una lista de tuplas de dos elementos que contienen todos los campos cabeceras y valores del mensaje.

`get`(*name*, *failobj=None*)
Retorna el valor de la cabecera nombrada. Esto es idéntico a *`__getitem__()`* excepto que el opcional *failobj* es retornado si no se encuentra la cabecera nombrada (*failobj* es por defecto `None`).

Aquí hay algunos métodos adicionales útiles relacionados con la cabecera:

`get_all`(*name*, *failobj=None*)
Retorna una lista de todos los valores para el campo *nombre*. Si no se nombran tales cabeceras en el mensaje, regresa *failobj* (por defecto `None`)

`add_header`(*_name*, *_value*, ***_params*)
Configuración extendida de cabeceras. Este método es similar a *`__setitem__()`*, excepto que se pueden proporcionar parámetros de cabecera 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 una cabecera 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 una cabecera. Reemplaza la primer cabecera encontrada en el mensaje que coincida con *_name*, conservando el orden de cabeceras y uso de minúsculas (y mayúsculas) del nombre de campo de la cabecera 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 cabecera llamada *Content-Type* en el mensaje, regresa el valor de *get_default_type()*. Si *Content-Type* no es válido, retorna `text/plain`.

(De acuerdo con **RFC 2045**, los mensajes siempre tienen un tipo predeterminado, `get_content_type`siempre retornará un valor. :rfc:`2045`define el tipo predeterminado de un mensaje como :mimetype:`text/plain()` a menos que aparezca dentro de un contenedor *multipart/digest*, en cuyo caso sería *message/rfc822*. Si la cabecera *Content-Type* tiene una especificación de tipo que sea inválida, **RFC 2045** exige que el tipo predeterminado 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 sub-contenido del mensaje. Esta es la parte *subtype* de la cadena retornada por *get_content_type()*.

get_default_type()
Retorna el tipo de contenido predeterminado. La mayoría de los mensajes tienen como tipo de contenido predeterminado a *text/plain*, a excepción de los mensajes que son sub-partes de contenedores *multipart/digest*. Estas sub-partes tienen como tipo de contenido predeterminado a *message/rfc822*.

set_default_type(*ctype*)
Establece el tipo de contenido predeterminado. *ctype* debería ser *text/plain* o *message/rfc822*, aunque esto no está impuesto. El tipo de contenido predeterminado no se almacena en la cabecera *Content-Type*, así que sólo afecta al valor de retorno de los métodos `get_content_type` cuando ninguna cabecera *Content-Type* está presente en el mensaje.

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, reemplace su valor con *value*. Cuando *header* es `Content-Type` (el predeterminado) y el encabezado aún no existe en el mensaje, agréguelo, establece su valor en *text/plain* y agregue el nuevo valor del parámetro. *header* opcional especifica un encabezado alternativo a *Content-Type*.

Si el valor contiene caracteres *non-ASCII*, el *charset* y el lenguaje pueden ser especificados explícitamente con los parámetros *charset* y *language*. Opcionalmente *language* especifica el lenguaje **RFC 2231**, por defecto una cadena vacía. Ambos *charset* y *language* deberán ser cadenas. los valores predeterminados son `utf8` para *charset* y `None` para *language*.

Si *replace* es `False` (predeterminado) la cabecera se mueve al final de la lista de cabeceras. Si *replace* es `True`, la cabecera se actualizará en su lugar.

El uso del parámetro *requote* con objetos *EmailMessage* está obsoleto.

Tenga en cuenta que se puede acceder a los parámetros existentes de las cabeceras a través del atributo `params` de la cabecera (por ejemplo, `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*)
Elimina completamente el parámetro dado de la cabecera *Content-Type*. La cabecera será reescrita en su lugar, sin el parámetro o su valor. Opcionalmente *header* especifica una alternativa a *Content-Type*.

El uso del parámetro *requote* con objetos *EmailMessage* está obsoleto.

get_filename(*failobj=None*)
Retorna el valor del parámetro `filename` de la cabecera *Content-Disposition* del mensaje. Si la cabecera no tiene un parámetro `filename`, este método recae a buscar el parámetro `name` en la

cabecera *Content-Type*. Si ninguno se encuentra o falta la cabecera, se retorna *failobj*. La cadena retornada siempre estará sin comillas según *email.utils.unquote()*.

get_boundary(*failobj=None*)
Retorna el parámetro boundary de la cabecera *Content-Type* del mensaje, o *failobj* si la cabecera no se encuentra o si no tiene un parámetro boundary. La cadena retornada siempre estará sin comillas según *email.utils.unquote()*.

set_boundary(*boundary*)
Establece el parámetro boundary de la cabecera *Content-Type* como *boundary*. *set_boundary()* siempre que sea necesario pondrá comillas a *boundary*. Si el objeto mensaje no tiene cabecera *Content-Type* se lanza *HeaderParseError*.

Note que usar este método es sutilmente diferente a borrar la cabecera *Content-Type* antigua y agregar una nueva con el nuevo límite utilizando *add_header()*, porque *set_boundary()* preserva el orden de la cabecera *Content-Type* en la lista de cabeceras.

get_content_charset(*failobj=None*)
Retorna el parámetro charset de la cabecera *Content-Type`forzado a minúsculas. Si no hay una cabecera :mailheader:`Content-Type*, o si esa cabecera no tiene el parámetro charset, se retorna *failobj*.

get_charsets(*failobj=None*)
Retorna una lista que contiene los nombres de los *charset* en el mensaje. Si el mensaje es *multipart*, la lista contendrá un elemento por cada subparte en la carga útil, de lo contrario será una lista de longitud 1.

Cada elemento de la lista será una cadena que tendrá el valor del parámetro charset en la cabecera *Content-Type* para la subparte representada. Si la subparte no tiene cabecera *Content-Type*, no tiene parámetro charset, o no es del tipo MIME principal *text*, entonces ese elemento en la lista retornada será *failobj*.

is_attachment()
Retorna True si hay una cabecera *Content-Disposition* y su valor (sensible a mayúsculas) es attachment, de lo contrario retorna False.

Distinto en la versión 3.4.2: *is_attachment* ahora es un método en lugar de una propiedad, por consistencia con *is_multipart()*.

get_content_disposition()
Retorna el valor en minúsculas (sin parámetros) de la cabecera *Content-Disposition* si el mensaje la tiene, de lo contrario retorna None. Los valores posibles para este método son *inline*, *attachment* o None si el mensaje sigue **RFC 2183**.

Nuevo en la versión 3.5.

Los siguientes métodos se refieren a interrogar y manipular el contenido (*payload*) del mensaje.

walk()
El método *walk()* es un generador multipropósito que puede usarse para iterar sobre todas las partes y subpartes del árbol de un objeto mensaje, ordenando el recorrido en profundidad primero. Típicamente se usaría *walk()* como el iterador en un ciclo for; cada iteración retornará la siguiente subparte.

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` itera sobre las subpartes de cualquier parte donde *`is_multipart()`* retorna `True`, aun si `msg.get_content_maintype() == 'multipart'` pueda retornar `False`. Podemos ver esto en nuestro ejemplo al hacer uso de la función auxiliar de depuración `_structure`:

```
>>> from email.iterators import _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 `message` no son `multiparts`, pero sí contienen subpartes. `is_multipart()``retorna ``True` y `walk` desciende a las subpartes.

get_body (*preferencelist=('related', 'html', 'plain')*)
Retorna la parte MIME que es la mejor candidata para ser el «cuerpo» del mensaje.

preferencelist debe ser una secuencia de cadenas del conjunto `related`, `html`, y `plain`, e Indica el orden de preferencia para el tipo de contenido de la parte retornada.

Empieza a buscar coincidencias candidatas con el objeto en el que se llama al método `get_body`".

Si `related` no está incluido en la *preferencelist*, considere la parte raíz (o subparte de la parte raíz) de cualquier relacionado encontrado como candidato si la (sub-)parte coincide con una preferencia.

Cuando se encuentra una `multipart/related`, verifica el parámetro `start` y si se encuentra una parte con un *`Content-ID`* que coincida, considera solamente a ésta cuando se busca coincidencias candidatas. De otra forma considera sólo la primera parte (predeterminado *root*) de la `multipart/related`.

Si una parte tiene una cabecera *`Content-Disposition`* , sólo se considera a ésta parte como coincidencia candidata si el valor de la cabecera es `inline`.

Si ninguno de los candidatos coincide con ninguna de las preferencias en *preferencelist*, retorna `None`.

Notas: (1) Para la mayoría de las aplicaciones las únicas combinaciones de *preferencelist* que realmente tienen sentido son `('plain',)`, `('html', 'plain')`, y la predeterminada `('related', 'html', 'plain')`. (2) Debido a que la coincidencia comienza con el objeto en el que se llama a `get_body`, llamar a `get_body` en un `multipart/related` devolverá el objeto en sí a menos que la *preferencelist* tenga un valor no predeterminado. (3) Los mensajes (o partes de mensajes) que no especifican un *`Content-Type`* o cuya cabecera *`Content-Type`* sea inválida se tratarán como si fueran del tipo `text/plain`, que ocasionalmente puede ocasionar que `get_body` retorne resultados inesperados.

iter_attachments()
Devuelve un iterador sobre todas las subpartes inmediatas del mensaje que no son partes candidatas del «body». Es decir, omitir la primer ocurrencia de cada `text/plain`, `text/html`, `multipart/related`, o `multipart/alternative``(a menos que estén marcados explícitamente como adjuntos a través de :mailheader:`Content-Disposition: attachment`) y retornar todas las`

partes restantes. Cuando se aplica directamente a un ``multipart/related, retorna un iterador sobre todas las partes relacionadas excepto la parte raíz (es decir, la parte apuntada por el parámetro start , o la primera parte si no hay un parámetro start o el parámetro start no coincide con el *Content-ID* de ninguna de las partes). Cuando se aplica directamente a un multipart/alternative o a un no-multipart, retorna un iterador vacío.

iter_parts()
Retorna un iterador sobre todas las partes inmediatas del mensaje, que estará vacío para una no-multipart. (vea también *walk()*.)

get_content(**args*, *content_manager=None*, ***kw*)
Llama al método *get_content()* del *content_manager*, pasando *self* como el objeto del mensaje y pasando cualquier otro argumento o palabra clave como argumentos adicionales. Si no se especifica *content_manager* se usa el content_manager especificado por la *policy* actual.

set_content(**args*, *content_manager=None*, ***kw*)
Llama al método *set_content()* del *content_manager*, pasando *self* como el objeto *message* y pasando cualquier otro argumento o palabra clave como argumentos adicionales. Si no se especifica *content_manager* se usa el content_manager especificado por la *policy* actual.

make_related(*boundary=None*)
Convierte un mensaje no-multipart a un mensaje multipart/related, moviendo cualquier cabecera *Content-* y la carga útil a una (nueva) primera parte del multipart. Si *boundary* se especifica, se utiliza como la cadena límite en el *multipart*, de otra forma deja que el límite se cree automáticamente cuando se necesite (por ejemplo, cuando se serializa el mensaje).

make_alternative(*boundary=None*)
Convierte un mensaje no-multipart o un mensaje multipart/related a uno multipart/alternative``moviendo cualquier cabecera :mailheader:`Content-` y la carga útil existentes a una (nueva) primera parte del ``multipart. Si *boundary* se especifica, se utiliza como la cadena límite en el *multipart*, de otra forma deja que el límite se cree automáticamente cuando se necesite (por ejemplo, cuando se serializa el mensaje).

make_mixed(*boundary=None*)
Convierte un mensaje no-multipart, multipart/related o multipart-alternative a uno multipart/mixed moviendo cualquier cabecera *Content-* y la carga útil existentes a una (nueva) primera parte del multipart. Si *boundary* se especifica, se utiliza como la cadena límite en el *multipart*, de otra forma deja que el límite se cree automáticamente cuando se necesite (por ejemplo, cuando se serializa el mensaje).

add_related(**args*, *content_manager=None*, ***kw*)
Si el mensaje es un multipart/related, crea un nuevo objeto mensaje, pasa todos los argumentos a su método *set_content()* y lo une al multipart con *attach()*. Si el mensaje es un no-multipart, llama a *make_related()* y procede como arriba. Si el mensaje es cualquier otro tipo de multipart, lanza *TypeError*. Si *content_manager* no es especificado, usa el content_manager especificado por la *policy* actual. Si la parte agregada no tiene un encabezado *Content-Disposition*, se agrega uno con el valor inline.

add_alternative(**args*, *content_manager=None*, ***kw*)
Si el mensaje es un multipart/alternative, crea un nuevo objeto mensaje, pasa todos los argumentos a su método *set_content()* y lo une al multipart con *attach()*. Si el mensaje es un no-multipart o multipart/related, llama a *make_alternative()* y procede como arriba. Si el mensaje es cualquier otro tipo de multipart, lanza *TypeError*. Si *content_manager* no es especificado, usa el content_manager especificado por la *policy* actual.

add_attachment(**args*, *content_manager=None*, ***kw*)
Si el mensaje es un multipart/mixed, cree un nuevo objeto de mensaje, pase todos los argumentos a su método *set_content()* y *attach()* al multipart. Si el mensaje no es multipart, multipart/related o multipart/alternative, llame *make_mixed()* y luego proceda como se indicó anteriormente. Si no se especifica *content_manager*, use el content_manager especificado por el actual *policy*. Si la parte agregada no tiene encabezado *Content-Disposition*, agregue uno con el valor attachment. Este método se pue-

de utilizar tanto para adjuntos explícitos (*Content-Disposition: attachment*) como para adjuntos `inline` (*Content-Disposition: inline*), pasando las opciones apropiadas al `content_manager`.

clear()
Elimina la carga útil y todas las cabeceras.

clear_content()
Elimina la carga útil y todos los `Content-` *headers*, dejando a las demás cabeceras intactas y en su orden original.

Los objetos *EmailMessage* tienen los siguientes atributos de instancia:

preamble
El formato de un documento MIME permite algo de texto entre la linea en blanco después de las cabeceras y la primera cadena límite de 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, al ver el texto sin formato del mensaje, o al ver el mensaje en un lector no compatible con MIME, este texto puede volverse visible.

El atributo *preamble* contiene este texto extra para documentos MIME. Cuando el *Parser* descubre texto después de las cabeceras pero antes de la primer cadena límite, asigna este texto al atributo *preamble* del mensaje. Cuando el *Generator* escribe la representación de texto sin formato de un mensaje MIME y encuentra que el mensaje tiene un atributo *preamble* escribirá este texto en el área entre las cabeceras y el primer límite. Véase *email.parser* y *email.generator* para conocer detalles.

Cabe señalar que si el objeto *message* no tiene preámbulo, el atributo *preamble* será igual a `None`.

epilogue
El atributo *epilogue* actúa de igual forma al atributo *preamble* con la excepción de que contiene texto que aparece entre el último límite y el fin del mensaje. Como con el *preamble*, si no hay texto de epílogo este atributo será `None`.

defects
El atributo *defects* contiene una lista de todos los errores encontrados al hacer el análisis sintáctico del mensaje. Vea *email.errors* para una descripción detallada de los posibles efectos del análisis sintáctico.

class email.message.**MIMEPart**(*policy=default*)
Esta clase representa una subparte de un mensaje MIME. Es idéntica a *EmailMessage*, excepto que las cabeceras *MIME-Version* son añadidas cuando se llama a *set_content()*, ya que las subpartes no necesitan su propia cabecera *MIME-Version*.

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 él 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 *smtplib.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 entrada[1]. (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.

If optional *mangle_from_* is True, put a > character in front of any line in the body that starts with the exact string "From ", that is From followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the *mangle_from_* setting of the *policy* (which is True for the *compat32* policy and False for all others). *mangle_from_* is intended for use when messages are stored in Unix mbox format (see *mailbox* and 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

[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.

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*.

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.

If optional *mangle_from_* is True, put a > character in front of any line in the body that starts with the exact string "From ", that is From followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the *mangle_from_* setting of the *policy* (which is True for the *compat32* policy and False for all others). *mangle_from_* is intended for use when messages are stored in Unix mbox format (see *mailbox* and 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 *maxheaderlen* 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*, *maxheaderlen=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 rellenada 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 no*text*
- `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 trabajo 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 todos 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` (`sendmails'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 (*7 bit clean*)» (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 (*7 bit clean*). Se requiere que los datos en las cabeceras todavía sean sólo ASCII y por lo tanto estarán codificadas (véase *fold_binary()* y *utf8* 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.

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 *linesep* 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 los *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:

`none`	todos los valores de la fuente usan el doblamiento original
`long`	los valores de la fuente que tengan una línea que sea más grande que `max_line_length` serán redoblados
`all`	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 *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

[1] Se añadió originalmente en 3.3 como un *característica provisional*.

get_content() o *set_content()* del objeto messageEmailMessage 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 headerrregistry.

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*).

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.

- `InvalidDateDefect` – Al decodificar una fecha no válida o un campo no parseable. El valor original se mantiene 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`*. 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.

[1] Originalmente añadido en 3.3 como módulo provisional *provisional module*

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

If the header value can be recognized as a valid date of one form or another, this attribute will contain a *datetime* instance representing that date. If the timezone of the input date is specified as `-0000` (indicating it is in UTC but contains no information about the source timezone), then *datetime* will be a naive *datetime*. If a specific timezone offset is found (including `+0000`), then *datetime* will contain an aware `datetime` that uses *datetime.timezone* to record the timezone offset.

El valor `decoded` 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*, *default_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 lanzará 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 `multipart`. 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>
    <img src="cid:{asparagus_cid}" />
  </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


def magic_html_parser(html_text, partfiles):
    """Return safety-sanitized html linked to partfiles.

    Rewrite the href="cid:...." attributes to point to the filenames in partfiles.
    Though not trivial, this should be possible using html.parser.
    """
    raise NotImplementedError("Add the magic needed")


# 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':
```

(continué en la próxima página)

(proviene de la página anterior)

```
        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():
        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:
    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.

Un mensaje de correo electrónico consta de *headers* y *payload*. Los encabezados deben ser nombres y valores de estilo **RFC 5322**, donde el nombre del campo y el valor están separados por dos puntos. Los dos puntos no forman

parte del nombre del campo ni del valor del campo. La carga útil puede ser un mensaje de texto simple, un objeto binario o una secuencia estructurada de submensajes, cada uno con su propio conjunto de encabezados y su propia carga útil. El último tipo de carga útil se indica mediante el mensaje que tiene un tipo MIME como *multipart/** o *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 debes sobreescribirlo 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).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the Unix mbox format. For more flexibility, instantiate a *Generator* instance and use its *flatten()* method directly. For example:

```
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).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the Unix mbox format. For more flexibility, instantiate a *BytesGenerator* instance and use its *flatten()* method directly. For example:

```
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 junto 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 lanzará 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.emailmessage.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.

In a model generated from bytes, any header values that (in contravention of the RFCs) contain non-ASCII bytes will, when retrieved through this interface, be represented as *Header* objects with a charset of 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 caracter 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 sobreescribe 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 ile-

gales 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 remplaza el encabezado *Content-Type*, manteniendo todos los parámetros en su lugar. Si *requote* 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 remplazada 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 ``multiparts`, 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-Versión* (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`

A subclass of `MIMENonMultipart`, the `MIMEApplication` class is used to represent MIME message objects of major type `application`. *_data* contains the bytes for the raw application data. Optional *_subtype* specifies the MIME subtype and defaults to `octet-stream`.

_encoder opcional es una 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`

A subclass of `MIMENonMultipart`, the `MIMEAudio` class is used to create MIME message objects of major type `audio`. *_audiodata* contains the bytes for the raw audio data. If this data can be decoded by the standard Python module `sndhdr`, then the subtype will be automatically included in the `Content-Type` header. Otherwise you can explicitly specify the audio subtype via the *_subtype* argument. If the minor type could not be guessed and *_subtype* was not given, then `TypeError` is raised.

_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`

A subclass of `MIMENonMultipart`, the `MIMEImage` class is used to create MIME message objects of major type `image`. *_imagedata* contains the bytes for the raw image data. If this data can be decoded by the standard Python module `imghdr`, then the subtype will be automatically included in the `Content-Type` header. Otherwise you can explicitly specify the image subtype via the *_subtype* argument. If the minor type could not be guessed and *_subtype* was not given, then `TypeError` is raised.

_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 (Compat32). 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 caracter 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 caracter 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 caracter tabulador. Este caracter se antepondrá 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 (`Compat32`). 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 `euc-jp`, los encabezados se codificarán con base64, los cuerpos no se codificarán, pero el texto de salida se convertirá del conjunto de caracteres `euc-jp` al conjunto de caracteres `iso-2022-jp`.

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

If the character set must be encoded before it can be used in an email header, this attribute will be set

to `charset.QP` (for quoted-printable), `charset.BASE64` (for base64 encoding), or `charset.SHORTEST` for the shortest of QP or BASE64 encoding. Otherwise, it will be `None`.

body_encoding
Same as *header_encoding*, but describes the encoding for the mail message's body, which indeed may be different than the header encoding. `charset.SHORTEST` is not allowed for *body_encoding*.

output_charset
Algunos conjuntos de caracteres deben ser convertidos antes de poder ser usados en cabeceras o cuerpos de correos. Si el *input_charset* es uno de ellos, este atributo contendrá el nombre del conjunto de caracteres al que será convertido. De lo contrario, será `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.

Esta es la cadena de caracteres `quoted-printable` o `base64` dependiendo de la codificación usada, o es una función, en cuyo caso se deberá llamar a la función con un solo argumento, el objeto Mensaje a ser codificado. La función deberá luego establecer el encabezado *Content-Transfer-Encoding* en lo que sea apropiado.

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*)
Codifica como encabezado la cadena de caracteres *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__()
Retorna *input_charset* como una cadena de caracteres convertida a minúsculas. `__repr__()` es un alias para *__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 códecs:

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 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 códecs que el módulo no conozca. Consulte la documentación del módulo *codecs* para más información.

El registro global de conjuntos de caracteres se mantiene en el módulo global diccionario `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`.

El registro de alias global de conjuntos de caracteres se mantiene en el módulo global diccionario `ALIASES`.

email.charset.**add_codec**(*charset*, *codecname*)
Añade un códec que asigna caracteres en el conjunto de caracteres dado hacia y desde Unicode.

charset es el nombre canónico de un conjunto de caracteres. *codename* 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 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 activara 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

[1] El codificado con *encode_quopri()* también codifica todas las tabulaciones y caracteres de espacios en los datos.

comillas dobles, éstas son eliminadas. De igual manera si *str* empieza y termina con comillas angulares (< y >), éstas son eliminadas.

email.utils.**parseaddr**(*address*)
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.

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*)
Este método retorna una lista de 2-tuplas de la forma retornada por `parseaddr()`. *fieldvalues* es una secuencia de valores de campos de cabecera como la que puede ser retornado por *Message.get_all*. Aquí hay un ejemplo sencillo que obtiene todos los destinatarios de un mensaje:

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

email.utils.**parsedate**(*date*)
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*)
Lo inverso de *format_datetime()*. Realiza la misma función que *parsedate()*, pero en caso de éxito retorna un *datetime*; de lo contrario, `ValueError` se lanza si *date* contiene un valor no válido, como una hora superior a 23 o una diferencia de zona horaria no comprendida entre -24 y 24 horas. Si la fecha de entrada tiene una zona horaria de `-0000`, el `datetime` será un `datetime` ingenuo, y si la fecha se ajusta a las RFC, representará una hora en UTC pero sin indicación de la zona horaria de origen real del mensaje de donde proviene la fecha. . Si la fecha de entrada tiene cualquier otra compensación de zona horaria válida, el `datetime` será un `datetime` consciente con el correspondiente *timezone* `tzinfo`.

Nuevo en la versión 3.3.

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.

[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.**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 naíf, 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
    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 hace obsoleto a **RFC 4627**) y por ECMA-404, es un formato de intercambio de datos ligero inspirado en la sintaxis literal del objeto JavaScript (aunque no es un subconjunto estricto de JavaScript [# rfc-errata] _).

> **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:

```
>>> 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 is a subset of YAML 1.2. The JSON produced by this module's default settings (in particular, the default *separators* value) is also a subset of YAML 1.0 and 1.1. This module can thus also be used as a YAML serializer.

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.

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* (una invocación `.write()`- que soporta *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.

If *check_circular* is false (default: `True`), then the circular reference check for container types will be skipped and a circular reference will result in an *RecursionError* (or worse).

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 is an optional function that will be called with the result of any object literal decoded (a `dict`). The return value of *object_hook* will be used instead of the `dict`. This feature can be used to implement custom decoders (e.g. JSON-RPC class hinting).

object_pairs_hook 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 hace uso de otro tipo de datos o analizador para enteros JSON (por ejemplo `float`).

Distinto en la versión 3.10.7: 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 lanzará 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()`.

Si los datos que se deserializan no constituyen un documento JSON válido, se lanzará un `JSONDecodeError`.

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.

Distinto en la versión 3.9: El argumento de palabra llave *encoding* ha sido removido.

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, if specified, will be called with the result of every JSON object decoded and its return value will be used in place of the given *dict*. This can be used to provide custom deserializations (e.g. to support JSON-RPC class hinting).

object_pairs_hook, 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 hace 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 lanzará 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 invalido.

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 lanzar *TypeError*).

Si *skipkeys* es falso (lo es por defecto), un *TypeError* será lanzado cuando se trate de codificar llaves que no sean *str*, *int*, *float* o None. Si *skipkeys* es verdadero, tales elementos son simplemente pasados por alto.

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.

If *check_circular* is true (the default), then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an *RecursionError*). Otherwise, no such check takes place.

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 *default()* de esta manera:

```
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 por **RFC 7159** y por ECMA-404. Esta sección detalla el nivel de cumplimiento de la RFC de este módulo. Para simplificar, *JSONEncoder* y *JSONDecoder* subclases, y los parámetros distintos de los mencionados explícitamente, no se consideran.

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.

--no-ensure-ascii
Deshabilita el escape de caracteres que no sean ascii, véase `json.dumps()` para más información.

Nuevo en la versión 3.9.

--json-lines
Analiza cada línea de entrada como objeto JSON independiente.

Nuevo en la versión 3.8.

--indent, --tab, --no-indent, --compact
Opciones mutuamente exclusivas para control de espacios en blanco.

Nuevo en la versión 3.9.

-h, --help
Muestra el mensaje de ayuda.

Notas de pie de página

19.3 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.

19.3.1 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 lanza una excepción *KeyError* si el método se llamó *remove()* o *__delitem__()* pero no se lanza 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()*.

itervalues()
__iter__()
values()
Retorna un iterador sobre las representaciones de todos los mensajes si se llama `itervalores()` 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 `Maiblox`.

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 lanza 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:

maildir man page from Courier Una especificación del formato. Describe una extensión común para admitir carpetas.

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 Una especificación del formato, con detalles sobre el bloqueo.

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 - Message Handling System 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 de 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.

19.3.2 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 coreo, 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	Destruido	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 *flags* 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 *mboxMessage* o *MMDFMessage*, se omiten las cabeceras *Status* y *X-Status* y se producen las siguientes conversiones:

Estado resultante	Estado *mboxMessage* o *MMDFMessage*
subdirectorio «cur»	flag O
flag F	flag F
flag R	flag A
flag S	flag R
flag T	flag D

Cuando se crea una instancia *MaildirMessage* basada en una instancia *MHMessage*, se producen las siguientes conversiones:

Estado resultante	Estado *MHMessage*
subdirectorio «cur»	Secuencia «*unseen*» (no vista)
subdirectorio «cur» e indicador S	no hay una secuencia «*unseen*» (invisible)
flag F	secuencia «*flagged*» (marcada)
flag R	Secuencia «*replied*» (respondida)

Cuando se crea una instancia *MaildirMessage* basada en una instancia *BabylMessage*, se producen las siguientes conversiones:

Estado resultante	Estado *BabylMessage*
subdirectorio «cur»	etiqueta «*unseen*» (invisible)
subdirectorio «cur» e indicador S	no hay una etiqueta «*unseen*» (invisible)
flag P	etiqueta «*forwarded*» o «*resent*» (reenviado)
flag R	etiqueta de «*answered*» (contestado)
flag T	etiqueta «deleted» (borrado)

mboxMessage

class mailbox.**mboxMessage**(*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 *mboxMessage* 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 *mboxMessage* 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 *MaildirMessage*
flag R	flag S
flag O	subdirectorio «cur»
flag D	flag T
flag F	flag F
flag A	flag R

Cuando se crea una instancia *mboxMessage* basada en una instancia *MHMessage*, se producen las siguientes conversiones:

Estado resultante	Estado *MHMessage*
flag R y flag O	no hay una secuencia «*unseen*» (invisible)
flag O	Secuencia «*unseen*» (no vista)
flag F	secuencia «*flagged*» (marcada)
flag A	Secuencia «*replied*» (respondida)

Cuando se crea una instancia *mboxMessage* basada en una instancia *BabylMessage*, se producen las siguientes conversiones:

Estado resultante	Estado *BabylMessage*
flag R y flag O	no hay una etiqueta «*unseen*» (invisible)
flag O	etiqueta «*unseen*» (invisible)
flag D	etiqueta «deleted» (borrado)
flag 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 *MMDFMessage*
flag R	flag R
flag O	flag O
flag D	flag D
flag F	flag F
flag A	flag 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
unseen (no visto)	No leído, pero previamente detectado por la MUA
replied (contestado)	Contestado a
flagged (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 `MaildirMessage`
Secuencia «*unseen*» (no vista)	no hay indicador S
Secuencia «*replied*» (respondida)	flag R
secuencia «*flagged*» (marcada)	flag 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 `mboxMessage` o `MMDFMessage`
Secuencia «*unseen*» (no vista)	sin indicador R
Secuencia «*replied*» (respondida)	flag A
secuencia «*flagged*» (marcada)	flag F

Cuando se crea una instancia `MHMessage` basada en una instancia `BabylMessage`, se producen las siguientes conversiones:

Estado resultante	Estado `BabylMessage`
Secuencia «*unseen*» (no vista)	etiqueta «*unseen*» (invisible)
Secuencia «*replied*» (respondida)	etiqueta de «*answered*» (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
unseen (no visto)	No leído, pero previamente detectado por la MUA
deleted (borrado)	Marcado para su posterior eliminación
filed (archivado)	Copiado a otro archivo o buzón de correo
answered (contestado)	Contestado a
forwarded (reenviado)	Reenviado
edited (editado)	Modificado por el usuario
resent (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 `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 `MaildirMessage`
etiqueta «*unseen*» (invisible)	no hay indicador S
etiqueta «deleted» (borrado)	flag T
etiqueta de «*answered*» (contestado)	flag R
etiqueta «*forwarded*» (reenviado)	flag 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 `mboxMessage` o `MMDFMessage`
etiqueta «*unseen*» (invisible)	sin indicador R
etiqueta «deleted» (borrado)	flag D
etiqueta de «*answered*» (contestado)	flag A

Cuando se crea una instancia `BabylMessage` basada en una instancia `MHMessage`, se producen las siguientes conversiones:

Estado resultante	Estado `MHMessage`
etiqueta «*unseen*» (invisible)	Secuencia «*unseen*» (no vista)
etiqueta de «*answered*» (contestado)	Secuencia «*replied*» (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 *MaildirMessage*
flag R	flag S
flag O	subdirectorio «cur»
flag D	flag T
flag F	flag F
flag A	flag R

Cuando se crea una instancia *MMDFMessage* basada en una instancia *MHMessage*, se producen las siguientes conversiones:

Estado resultante	Estado *MHMessage*
flag R y flag O	no hay una secuencia «*unseen*» (invisible)
flag O	Secuencia «*unseen*» (no vista)
flag F	secuencia «*flagged*» (marcada)
flag A	Secuencia «*replied*» (respondida)

Cuando se crea una instancia *MMDFMessage* basada en una instancia *BabylMessage*, se producen las siguientes conversiones:

Estado resultante	Estado *BabylMessage*
flag R y flag O	no hay una etiqueta «*unseen*» (invisible)
flag O	etiqueta «*unseen*» (invisible)
flag D	etiqueta «deleted» (borrado)
flag 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 *mboxMessage*
flag R	flag R
flag O	flag O
flag D	flag D
flag F	flag F
flag A	flag A

19.3.3 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**
Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely generated file name already exists.

exception mailbox.**FormatError**
Se lanza cuando los datos de un archivo no pueden ser analizados, como cuando una instancia *MH* intenta leer un archivo `.mh_sequences` corrupto.

19.3.4 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()
```

19.4 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'
```

19.4.1 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.

19.5 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 4648**, que define los algoritmos Base16, Base32 y Base64, y para las codificaciones estándar de facto Ascii85 y Base85.

Las codificaciones **RFC 4648** son adecuadas para codificar datos binarios para que puedan enviarse de forma segura por correo electrónico, usarse como parte de las URL o incluirse como parte de una solicitud HTTP POST. El algoritmo de codificación no es el mismo que el del programa **uuencode**.

Hay dos interfaces proporcionadas por este módulo. La interfaz moderna admite la codificación de *objetos similares a bytes* a ASCII `bytes` y la decodificación de *objetos similares a bytes* o cadenas que contienen ASCII a `bytes`. Se admiten los dos alfabetos base 64 definidos en **RFC 4648** (normal y seguro para URL y sistema de archivos).

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.

  Optional *altchars* must be a *bytes-like object* of length 2 which specifies an alternative alphabet for the `+` and `/` characters. This allows an application to e.g. generate URL or filesystem safe Base64 strings. The default is `None`, for which the standard Base64 alphabet is used.

  May assert or raise a `ValueError` if the length of *altchars* is not 2. Raises a `TypeError` if *altchars* is not a *bytes-like object*.

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.

  Optional *altchars* must be a *bytes-like object* or ASCII string of length 2 which specifies the alternative alphabet used instead of the `+` and `/` characters.

  Una excepción `binascii.Error` se lanza si *s* está incorrectamente rellenado (*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`.

  May assert or raise a `ValueError` if the length of *altchars* is not 2.

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 4648 permite la asignación opcional del dígito 0 (cero) a la letra O (oh), y la asignación opcional del dígito 1 (uno) a la letra I (ojo) o la letra L (el) . El argumento opcional *map01* cuando no es `None`, especifica la letra a la cual el dígito 1 debería mapearse(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.**b32hexencode**(*s*)
Similar a *b32encode()* pero usa el Alfabeto Hexagonal Extendido, como se define en **RFC 4648**.

Nuevo en la versión 3.10.

base64.**b32hexdecode**(*s*, *casefold=False*)
Similar a *b32decode()* pero usa el Alfabeto Hexagonal Extendido, como se define en **RFC 4648**.

Esta versión no permite el dígito 0 (cero) a la letra O (oh) y el dígito 1 (uno) a las asignaciones de la letra I (ojo) o la letra L (el), todos estos caracteres están incluidos en el Alfabeto Hexagonal Extendido y no son intercambiables.

Nuevo en la versión 3.10.

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 rellenada (*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\x0b'*)
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.**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()` retorna 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.

Un ejemplo de uso del módulo:

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSBlbmNvZGVk'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

19.5.1 Consideraciones de Seguridad

Se agregó una nueva sección de consideraciones de seguridad a **RFC 4648** (sección 12); se recomienda revisar la sección de seguridad para cualquier código implementado en producción.

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.

19.6 `binhex` — Codificar y decodificar archivos binhex4

Código fuente: Lib/binhex.py

Obsoleto desde la versión 3.9.

Este módulo codifica y descodifica 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.

19.6.1 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 Macintosch, aún así se usará la antigua convención de nueva línea de Macintosh (retorno de carro al final de la línea).

19.7 `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 unicode con sólo caracteres ASCII 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.

Obsoleto desde la versión 3.9.

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.

Obsoleto desde la versión 3.9.

binascii.**rlecode_hqx**(*data*)
Realiza la compresión RLE de estilo binhex4 en *data* y retorna el resultado.

Obsoleto desde la versión 3.9.

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).

Obsoleto desde la versión 3.9.

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*])
Compute CRC-32, the unsigned 32-bit checksum of *data*, starting with an initial CRC of *value*. The default initial CRC is zero. The algorithm is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows:

```
print(binascii.crc32(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: The result is always unsigned. To generate the same numeric value when using Python 2 or earlier, 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'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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* Soporte para compatibilidad con RFC de codificación de estilo base64 en base 16, 32, 64 y 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.

19.8 quopri — Codificar y decodificar datos MIME imprimibles entre comillas

Código fuente: Lib/quopri.py

Este módulo realiza la codificación y descodificación 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

CAPÍTULO 20

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 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 `<a href="...">`.

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 subclasificar *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*`)`
This method is called to handle the start tag of an element (e.g. `<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 `<A HREF="https://www.cwi.nl/">`, 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 (`<img .../>`). 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

`&#NNN;` 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('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

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 HTML5[1] 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.

[1] See https://html.spec.whatwg.org/multipage/named-characters.html#named-character-references

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`.

> **Advertencia:** Los módulos XML no son seguros contra datos construidos errónea o maliciosamente. Si necesita analizar datos que no son de confianza o no autenticados, consulte las secciones *Vulnerabilidades XML* y *El paquete defusedxml*.

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 construidos 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	Seguro (5)	Seguro (2)	Seguro (3)	Seguro (5)	Seguro (4)
Recuperación de DTD	Seguro (5)	Seguro	Seguro	Seguro (5)	Seguro
bomba de descompresión	Seguro	Seguro	Seguro	Seguro	**Vulnerable**

1. Expat 2.4.1 y nuevas versiones no son vulnerables a las vulnerabilidades de «mil millones de risas» y «explosión cuadrática». Los elementos siguen listados como vulnerables debido a la posible dependencia de las bibliotecas proporcionadas por el sistemas. Verifique `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.

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 A quadratic blowup attack is similar to a Billion Laughs attack; it abuses entity expansion, too. Instead of nested entities it repeats one large entity with a couple of thousand chars over and over again. The attack isn't as efficient as the exponential case but it avoids triggering parser countermeasures that forbid deeply nested entities.

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.

La documentación de defusedxml en PyPI tiene más información sobre todos los vectores de ataque conocidos con ejemplos y referencias.

20.4.2 El paquete `defusedxml`

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` — La API XML de ElementTree

Código fuente: Lib/xml/etree/ElementTree.py

El módulo *xml.etree.ElementTree* implementa una API simple y eficiente para parsear y crear datos XML.

Distinto en la versión 3.3: Este módulo utilizará una implementación rápida siempre que esté disponible.

Obsoleto desde la versión 3.3: El módulo xml.etree.cElementTree es obsoleto.

> **Advertencia:** El módulo *xml.etree.ElementTree* no es seguro contra datos construidos maliciosamente. Si necesita parsear datos no fiables o no autentificados, vea *Vulnerabilidades XML*.

20.5.1 Tutorial

Este es un tutorial corto para usar *xml.etree.ElementTree* (ET en resumen). El objetivo es demostrar algunos de los componentes y conceptos básicos del módulo.

Árbol y elementos XML

XML es un formato de datos inherentemente jerárquico, y la forma más natural de representarlo es con un árbol. ET tiene dos clases para este propósito - *ElementTree* representa todo el documento XML como un árbol, y *Element* representa un solo nodo en este árbol. Las interacciones con todo el documento (leer y escribir en/desde archivos) se realizan normalmente en el nivel de *ElementTree*. Las interacciones con un solo elemento XML y sus sub-elementos se realizan en el nivel *Element*.

Procesando XML

Usaremos el siguiente documento XML como los datos de muestra para esta sección:

```
<?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>
```

Podemos importar estos datos leyendo desde un archivo:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

O directamente desde una cadena de caracteres:

```
root = ET.fromstring(country_data_as_string)
```

fromstring() analiza el XML de una cadena de caracteres directamente en un *Element*, que es el elemento raíz del árbol analizado. Otras funciones de análisis pueden crear un *ElementTree*. Compruebe la documentación para estar seguro.

Como un *Element*, root tiene una etiqueta y un diccionario de atributos:

```
>>> root.tag
'data'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> root.attrib
{}
```

También tiene nodos hijos sobre los cuales podemos iterar:

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

Los hijos están anidados, y podemos acceder a nodos hijos específicos por el índice:

```
>>> root[0][1].text
'2008'
```

Nota: No todos los elementos de la entrada XML acabarán siendo elementos del árbol analizado. Actualmente, este módulo omite cualquier comentario XML, instrucciones de procesamiento y declaraciones de tipo documento en la entrada. Sin embargo, los árboles construidos utilizando la API de este módulo, en lugar de analizar el texto XML, pueden contener comentarios e instrucciones de procesamiento, que se incluirán al generar la salida XML. Se puede acceder a una declaración de tipo documento pasando una instancia *TreeBuilder* personalizada al constructor *XMLParser*.

API de consulta para un procesamiento no bloqueante

La mayoría de las funciones de análisis proporcionadas por este módulo requieren que se lea todo el documento a la vez antes de retornar cualquier resultado. Es posible utilizar un *XMLParser* y alimentar los datos en él de forma incremental, pero se trata de una *push* API que llama a métodos en un objetivo invocable, que es demasiado bajo nivel e inconveniente para la mayoría de las necesidades. A veces, lo que el usuario realmente quiere es ser capaz de analizar XML de forma incremental, sin bloquear las operaciones, mientras disfruta de la comodidad de los objetos *Element* totalmente construidos.

La herramienta más potente para hacer esto es *XMLPullParser*. No requiere una lectura de bloqueo para obtener los datos XML, y en su lugar se alimenta de datos de forma incremental con llamadas a *XMLPullParser.feed()*. Para obtener los elementos XML analizados, llama a *XMLPullParser.read_events()*. He aquí un ejemplo:

```
>>> 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
```

El caso de uso obvio es el de las aplicaciones que operan de forma no bloqueante, donde los datos XML se reciben de un socket o se leen de forma incremental desde algún dispositivo de almacenamiento. En estos casos, las lecturas bloqueantes son inaceptables.

Debido a su flexibilidad, *XMLPullParser* puede ser un inconveniente para los casos de uso más simples. Si no te importa que tu aplicación se bloquee en la lectura de datos XML pero te gustaría tener capacidades de análisis incremental, echa un vistazo a *iterparse()*. Puede ser útil cuando estás leyendo un documento XML grande y no quieres mantenerlo completamente en memoria.

Encontrando elementos interesantes

Element tiene algunos métodos útiles que ayudan a iterar recursivamente sobre todo el sub-árbol por debajo de él (sus hijos, los hijos de sus hijos, y así sucesivamente). Por ejemplo, *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() encuentra sólo los elementos con una etiqueta que son hijos directos del elemento actual. *Element.find()* encuentra el *primer* hijo con una etiqueta determinada, y *Element.text* accede al contenido de texto del elemento. *Element.get()* accede a los atributos del elemento:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

Es posible especificar de forma más sofisticada qué elementos buscar utilizando *XPath*.

Modificando un archivo XML

ElementTree proporciona una forma sencilla de construir documentos XML y escribirlos en archivos. El método *ElementTree.write()* sirve para este propósito.

Una vez creado, un objeto *Element* puede ser manipulado cambiando directamente sus campos (como *Element.text*), añadiendo y modificando atributos (método *Element.set()*), así como añadiendo nuevos hijos (por ejemplo con *Element.append()*).

Digamos que queremos añadir uno al rango de cada país, y añadir un atributo `updated` al elemento rango:

```
>>> 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')
```

Nuestro XML tiene ahora este aspecto:

```
<?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>
```

Podemos eliminar elementos utilizando *Element.remove()*. Digamos que queremos eliminar todos los países con un rango superior a 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')
```

Tenga en cuenta que la modificación concurrente mientras se itera puede conducir a problemas, al igual que cuando se itera y modifica listas o diccionarios de Python. Por lo tanto, el ejemplo recoge primero todos los elementos coincidentes con `root.findall()`, y sólo entonces itera sobre la lista de coincidencias.

Nuestro XML tiene ahora este aspecto:

```
<?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>
```

Construyendo documentos XML

La función *SubElement()* también proporciona una forma cómoda de crear nuevos sub-elementos para un elemento dado:

```
>>> 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>
```

Procesando XML con espacio de nombres

Si la entrada XML tiene espacio de nombres, las etiquetas y los atributos con prefijos de la forma `prefix:sometag` se expanden a `{uri}sometag` donde el *prefix* se sustituye por el *URI* completo. Además, si hay un espacio de nombre por defecto, ese URI completo se antepone a todas las etiquetas sin prefijo.

A continuación se muestra un ejemplo de XML que incorpora dos espacios de nombres, uno con el prefijo «fictional» y el otro que sirve como espacio de nombres por defecto:

```
<?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>
```

Una forma de buscar y explorar este ejemplo XML es añadir manualmente el URI a cada etiqueta o atributo en el XPath de un *`find()`* o *`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)
```

Una mejor manera de buscar en el ejemplo de XML con espacio para nombres es crear un diccionario con sus propios prefijos y utilizarlos en las funciones de búsqueda:

```
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)
```

Estos dos enfoques dan como resultado:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

20.5.2 Soporte de XPath

Este módulo proporciona un soporte limitado para las expresiones XPath para localizar elementos en un árbol. El objetivo es soportar un pequeño subconjunto de la sintaxis abreviada; un motor XPath completo está fuera del alcance del módulo.

Ejemplo

A continuación se muestra un ejemplo que demuestra algunas de las capacidades de XPath del módulo. Utilizaremos el documento XML `countrydata` de la sección *Parsing XML*:

```
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]")
```

Para XML con espacios de nombre, use la notación calificada habitual `{namespace}tag`:

```
# All dublin-core "title" tags in the document
root.findall(".//{http://purl.org/dc/elements/1.1/}title")
```

Sintaxis XPath soportada

Sintaxis	Significado
`tag`	Selecciona todos los elementos hijos con la etiqueta dada. Por ejemplo, `spam` selecciona todos los elementos hijos llamados `spam`, y `spam/egg` selecciona todos los nietos llamados `egg` en todos los hijos llamados `spam`. `{namespace}*` selecciona todas las etiquetas en el espacio de nombres dado, `{*}spam` selecciona las etiquetas llamadas `spam` en cualquier (o ningún) espacio de nombres, y `{}*` sólo selecciona las etiquetas que no están en un espacio de nombres. Distinto en la versión 3.8: Se ha añadido la posibilidad de utilizar comodines asterisco.
`*`	Selecciona todos los elementos hijos, incluidos los comentarios y las instrucciones de procesamiento. Por ejemplo, `*/egg` selecciona todos los hijos llamados `egg`.
`.`	Selecciona el nodo actual. Esto es útil sobre todo al principio de la ruta, para indicar que es una ruta relativa.
`//`	Selecciona todos los sub-elementos, en todos los niveles por debajo del elemento actual. Por ejemplo, `.//egg` selecciona todos los elementos `egg` en todo el árbol.
`..`	Selecciona el elemento padre. Retorna `None` si la ruta intenta llegar a los ancestros del elemento inicial (el elemento `find` fue invocado).
`[@attrib]`	Selecciona todos los elementos que tengan el atributo dado.
`[@attrib='value']`	Selecciona todos los elementos para los que el atributo dado tiene el valor dado. El valor no puede contener comillas.
`[@attrib!='value']`	Selecciona todos los elementos para los que el atributo dado no tiene el valor dado. El valor no puede contener comillas. Nuevo en la versión 3.10.
`[tag]`	Selecciona todos los elementos que tienen un hijo llamado `tag`. Sólo se admiten los hijos inmediatos.
`[.='text']`	Selecciona todos los elementos cuyo contenido de texto completo, incluyendo los descendientes, es igual al «texto» dado. Nuevo en la versión 3.7.
`[.!='text']`	Selecciona todos los elementos cuyo contenido de texto completo, incluidos los descendientes, no es igual al `text` proporcionado. Nuevo en la versión 3.10.
`[tag='text']`	Selecciona todos los elementos que tienen un hijo llamado `tag` cuyo contenido de texto completo, incluyendo los descendientes, es igual al `text` dado.
`[tag!='text']`	Selecciona todos los elementos que tienen un hijo llamado `tag` cuyo contenido de texto completo, incluidos los descendientes, no es igual al `text` dado. Nuevo en la versión 3.10.
`[position]`	Selecciona todos los elementos que se encuentran en la posición dada. La posición puede ser un número entero (1 es la primera posición), la expresión `last()` (para la última posición), o una posición relativa a la última posición (por ejemplo, `last()-1`).

Los predicados (expresiones entre corchetes) deben ir precedidos de un nombre de etiqueta, un asterisco u otro predicado. Los predicados `position` deben ir precedidos de un nombre de etiqueta.

20.5.3 Referencia

Funciones

xml.etree.ElementTree.**canonicalize**(*xml_data=None*, *, *out=None*, *from_file=None*, **options*)

Función de transformación C14N 2.0.

La canonización es una forma de normalizar la salida de XML de manera que permita comparaciones byte a byte y firmas digitales. Reduce la libertad que tienen los serializadores XML y en su lugar genera una representación XML más restringida. Las principales restricciones se refieren a la colocación de las declaraciones de espacio de nombres, el orden de los atributos y los espacios en blanco ignorables.

Esta función toma una cadena de datos XML (*xml_data*) o una ruta de archivo o un objeto tipo archivo (*from_file*) como entrada, la convierte a la forma canónica y la escribe utilizando el objeto archivo (o tipo archivo) *out*, si se proporciona, o la devuelve como una cadena de texto si no. El archivo de salida recibe texto, no bytes. Por tanto, debe abrirse en modo texto con codificación `utf-8`.

Usos típicos:

```
xml_data = "<root>...</root>"
print(canonicalize(xml_data))

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    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)
```

Las opciones de configuración *options* son las siguientes:

- *with_comments*: configurar a `True` para incluir los comentarios (por defecto: `False`)
- ***strip_text*: configurar a `True` para eliminar los espacios en blanco antes y después del contenido del texto** (por defecto: `False`)
- ***rewrite_prefixes*: configurar a `True` para sustituir los prefijos de espacios de nombres por «n{number}»** (por defecto: `False`)
- ***qname_aware_tags*: un conjunto de nombres de etiquetas conscientes de qname en el que los prefijos** deben ser reemplazados en el contenido del texto (por defecto: vacío)
- ***qname_aware_attrs*: un conjunto de nombres de atributos conscientes de qname en el que los prefijos** deben ser reemplazados en el contenido del texto (por defecto: vacío)
- *exclude_attrs*: un conjunto de nombres de atributos que no deben serializarse
- *exclude_tags*: un conjunto de nombres de etiquetas que no deben serializarse

En la lista de opciones anterior, «un conjunto» se refiere a cualquier colección o iterable de cadenas, no se espera ningún orden.

Nuevo en la versión 3.8.

xml.etree.ElementTree.**Comment**(*text=None*)

Fábrica de elementos de comentario. Esta función de fábrica crea un elemento especial que será serializado como un comentario XML por el serializador estándar. La cadena de comentario puede ser una cadena de bytes o una cadena Unicode. *text* es una cadena que contiene la cadena de comentario. Devuelve una instancia de elemento que representa un comentario.

Tenga en cuenta que *XMLParser* omite los comentarios en la entrada en lugar de crear objetos de comentario para ellos. Un *ElementTree* sólo contendrá nodos de comentario si se han insertado en el árbol utilizando uno de los métodos *Element*.

xml.etree.ElementTree.**dump**(*elem*)
Escribe un árbol de elementos o una estructura de elementos en sys.stdout. Esta función debe utilizarse únicamente para debugging.

El formato de salida exacto depende de la implementación. En esta versión, se escribe como un archivo XML ordinario.

elem es un árbol de elementos o un elemento individual.

Distinto en la versión 3.8: La función *dump()* ahora preserva el orden de atributos especificado por el usuario.

xml.etree.ElementTree.**fromstring**(*text*, *parser=None*)
Analiza una sección XML a partir de una constante de cadena. Igual que *XML()*. *text* es una cadena que contiene datos XML. *parser* es una instancia de parser opcional. Si no se da, se utiliza el analizador estándar *XMLParser*. Devuelve una instancia de *Element*.

xml.etree.ElementTree.**fromstringlist**(*sequence*, *parser=None*)
Analiza un documento XML a partir de una secuencia de fragmentos de cadena de caracteres. *sequence* es una lista u otra secuencia que contiene fragmentos de datos XML. *parser* es una instancia de parser opcional. Si no se da, se utiliza el analizador estándar *XMLParser*. Retorna una instancia de *Element*.

Nuevo en la versión 3.2.

xml.etree.ElementTree.**indent**(*tree*, *space=' '*, *level=0*)
Añade espacios en blanco al subárbol para indentar el árbol visualmente. Esto puede utilizarse para generar una salida XML con una impresión bonita. *tree* puede ser un Element o ElementTree. *space* es la cadena de espacio en blanco que se insertará para cada nivel de indentación, dos caracteres de espacio por defecto. Para indentar subárboles parciales dentro de un árbol ya indentado, pase el nivel de indentación inicial como *level*.

Nuevo en la versión 3.9.

xml.etree.ElementTree.**iselement**(*element*)
Comprueba si un objeto parece ser un objeto elemento válido. *element* es una instancia de elemento. Retorna `True` si se trata de un objeto elemento.

xml.etree.ElementTree.**iterparse**(*source*, *events=None*, *parser=None*)
Analiza una sección XML en un árbol de elementos de forma incremental, e informa al usuario de lo que ocurre. *source* es un nombre de archivo o un *file object* que contiene datos XML. *events* es una secuencia de eventos para informar. Los eventos soportados son las cadenas `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` y `"end-ns"` (los eventos «ns» se utilizan para obtener información detallada del espacio de nombres). Si se omite *events*, sólo se informará de los eventos `"end"`. *parser* es una instancia opcional de parser. Si no se da, se utiliza el analizador estándar de *XMLParser*. *parser* debe ser una subclase de *XMLParser* y sólo puede utilizar el *TreeBuilder* por defecto como objetivo. Devuelve un *iterator* que proporciona pares `(event, elem)`.

Tenga en cuenta que mientras *iterparse()* construye el árbol de forma incremental, emite lecturas de bloqueo en la *source* (o en el fichero que nombra). Por lo tanto, no es adecuado para aplicaciones en las que no se pueden realizar lecturas de bloqueo. Para un análisis completamente no bloqueante, véase *XMLPullParser*.

Nota: *iterparse()* sólo garantiza que ha visto el carácter «>» de una etiqueta de inicio cuando emite un evento «start», por lo que los atributos están definidos, pero el contenido de los atributos text y tail está indefinido en ese momento. Lo mismo ocurre con los hijos del elemento; pueden estar presentes o no.

Si necesita un elemento totalmente poblado, busque los eventos «end» en su lugar.

Obsoleto desde la versión 3.4: El argumento *parser*.

Distinto en la versión 3.8: Los eventos `comment` y `pi` han sido añadidos.

xml.etree.ElementTree.**parse**(*source*, *parser=None*)
Analiza una sección XML en un árbol de elementos. *source* es un nombre de archivo o un objeto de archivo que contiene datos XML. *parser* es una instancia de parser opcional. Si no se da, se utiliza el analizador estándar *XMLParser*. Devuelve una instancia de *ElementTree*.

xml.etree.ElementTree.**ProcessingInstruction**(*target*, *text=None*)
Fábrica de elementos PI. Esta función de fábrica crea un elemento especial que será serializado como una instrucción de procesamiento XML. *target* es una cadena que contiene el objetivo de PI. *text* es una cadena que contiene el contenido de PI, si se da. Devuelve una instancia de elemento, representando una instrucción de procesamiento.

Tenga en cuenta que *XMLParser* omite las instrucciones de procesamiento en la entrada en lugar de crear objetos de comentario para ellas. Un *ElementTree* sólo contendrá nodos de instrucciones de procesamiento si se han insertado en el árbol utilizando uno de los métodos *Element*.

xml.etree.ElementTree.**register_namespace**(*prefix*, *uri*)
Registra un prefijo de espacio de nombres. El registro es global, y cualquier asignación existente para el prefijo dado o el URI del espacio de nombres será eliminado. *prefix* es un prefijo de espacio de nombres. *uri* es una uri del espacio de nombres. Las etiquetas y los atributos de este espacio de nombres se serializarán con el prefijo dado, si es posible.

Nuevo en la versión 3.2.

xml.etree.ElementTree.**SubElement**(*parent*, *tag*, *attrib={}*, ***extra*)
Fábrica de sub-elementos. Esta función crea una instancia de elemento y la añade a un elemento existente.

El nombre del elemento, los nombres de los atributos y los valores de los atributos pueden ser cadenas de bytes o cadenas de caracteres Unicode. *parent* es el elemento padre. *tag* es el nombre del sub-elemento. *attrib* es un diccionario opcional que contiene los atributos del elemento. *extra* contiene atributos adicionales, dados como argumentos de palabras clave. Devuelve una instancia de elemento.

xml.etree.ElementTree.**tostring**(*element*, *encoding='us-ascii'*, *method='xml'*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)
Genera una representación de cadena de caracteres de un elemento XML, incluyendo todos los sub-elementos. *element* es una instancia de *Element*. *encoding*[1] es la codificación de salida (por defecto es US-ASCII). Utilice `encoding="unicode"` para generar una cadena de caracteres Unicode (de lo contrario, se genera una cadena de bytes). *method* es `"xml"`, `"html"` o `"text"` (por defecto es `"xml"`). *xml_declaration*, *default_namespace* y *short_empty_elements* tienen el mismo significado que en *ElementTree.write()*. Devuelve una cadena (opcionalmente) codificada que contiene los datos XML.

Nuevo en la versión 3.4: El parámetro *short_empty_elements*.

Nuevo en la versión 3.8: Los parámetros *xml_declaration* y *default_namespace*.

Distinto en la versión 3.8: La función *tostring()* ahora preserva el orden de atributos especificado por el usuario.

xml.etree.ElementTree.**tostringlist**(*element*, *encoding='us-ascii'*, *method='xml'*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)
Genera una representación de cadena de caracteres de un elemento XML, incluyendo todos los sub-elementos. *element* es una instancia de *Element*. *encoding*[1] es la codificación de salida (por defecto es US-ASCII). Utilice `encoding="unicode"` para generar una cadena de caracteres Unicode (de lo contrario, se genera una cadena de bytes). *method* es `"xml"`, `"html"` o `"text"` (por defecto es `"xml"`). *xml_declaration*, *default_namespace* y *short_empty_elements* tienen el mismo significado que en *ElementTree.write()*. Devuelve una lista de cadenas (opcionalmente) codificadas que contienen los datos XML. No garantiza ninguna secuencia específica, excepto que `b"".join(tostringlist(element)) == tostring(element)`.

Nuevo en la versión 3.2.

Nuevo en la versión 3.4: El parámetro *short_empty_elements*.

Nuevo en la versión 3.8: Los parámetros *xml_declaration* y *default_namespace*.

[1] La cadena de caracteres de codificación incluida en la salida XML debe ajustarse a los estándares adecuados. 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.

Distinto en la versión 3.8: La función *tostringlist()* ahora preserva el orden de atributos especificado por el usuario.

xml.etree.ElementTree.**XML**(*text*, *parser=None*)
Analiza una sección XML a partir de una constante de cadena de caracteres. Esta función puede utilizarse para incrustar «literales XML» en el código de Python. *text* es una cadena de caracteres que contiene datos XML. *parser* es una instancia de parser opcional. Si no se da, se utiliza el analizador estándar *XMLParser*. Devuelve una instancia de *Element*.

xml.etree.ElementTree.**XMLID**(*text*, *parser=None*)
Analiza una sección XML a partir de una constante de cadena de caracteres, y también devuelve un diccionario que mapea los id:s de elementos a elementos. *text* es una cadena de caracteres que contiene datos XML. *parser* es una instancia de parser opcional. Si no se da, se utiliza el analizador estándar de *XMLParser*. Devuelve una tupla que contiene una instancia de *Element* y un diccionario.

20.5.4 Soporte de XInclude

Este módulo proporciona un soporte limitado para las directivas XInclude, a través del módulo de ayuda xml.etree.ElementInclude. Este módulo puede utilizarse para insertar subárboles y cadenas de texto en árboles de elementos, basándose en la información del árbol.

Ejemplo

Aquí hay un ejemplo que demuestra el uso del módulo XInclude. Para incluir un documento XML en el documento actual, utilice el elemento {http://www.w3.org/2001/XInclude}include y establezca el atributo **parse** como "xml", y utilice el atributo **href** para especificar el documento a incluir.

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="source.xml" parse="xml" />
</document>
```

Por defecto, el atributo **href** se trata como un nombre de archivo. Puede utilizar cargadores personalizados para anular este comportamiento. También tenga en cuenta que el ayudante estándar no soporta la sintaxis XPointer.

Para procesar este archivo, cárguelo como de costumbre y pase el elemento raíz al módulo *xml.etree.ElementTree*:

```
from xml.etree import ElementTree, ElementInclude

tree = ElementTree.parse("document.xml")
root = tree.getroot()

ElementInclude.include(root)
```

El módulo ElementInclude sustituye el elemento {http://www.w3.org/2001/XInclude}include por el elemento raíz del documento **source.xml**. El resultado podría ser algo así:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <para>This is a paragraph.</para>
</document>
```

Si se omite el atributo **parse**, el valor por defecto es «xml». El atributo href es obligatorio.

Para incluir un documento de texto, utilice el elemento {http://www.w3.org/2001/XInclude}include y establezca el atributo **parse** como «text»:

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
```

(continué en la próxima página)

(proviene de la página anterior)

```
  Copyright (c) <xi:include href="year.txt" parse="text" />.
</document>
```

El resultado podría ser algo así:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) 2003.
</document>
```

20.5.5 Referencia

Funciones

xml.etree.ElementInclude.**default_loader**(*href*, *parse*, *encoding=None*)
Cargador por defecto. Este cargador por defecto lee un recurso incluido del disco. *href* es una URL. *parse* es para el modo de análisis «xml» o «text». *encoding* es una codificación de texto opcional. Si no se da, la codificación es `utf-8`. Retorna el recurso expandido. Si el modo de análisis es `"xml"`, es una instancia de ElementTree. Si el modo de análisis es «text», se trata de una cadena Unicode. Si el cargador falla, puede retornar None o lanzar una excepción.

xml.etree.ElementInclude.**include**(*elem*, *loader=None*, *base_url=None*, *max_depth=6*)
Esta función expande las directivas XInclude. *elem* es el elemento raíz. *loader* es un cargador de recursos opcional. Si se omite, se utiliza por defecto *default_loader()*. Si se da, debe ser un callable que implemente la misma interfaz que *default_loader()*. *base_url* es la URL base del archivo original, para resolver las referencias relativas al archivo de inclusión. *max_depth* es el número máximo de inclusiones recursivas. Limitado para reducir el riesgo de explosión de contenido malicioso. Pase un valor negativo para desactivar la limitación.

Retorna el recurso expandido. Si el modo de análisis es `"xml"`, se trata de una instancia de ElementTree. Si el modo de análisis es «text», se trata de una cadena Unicode. Si el cargador falla, puede retornar None o lanzar una excepción.

Nuevo en la versión 3.9: Los parámetros *base_url* y *max_depth*.

Objetos Element

class xml.etree.ElementTree.**Element**(*tag*, *attrib={}*, ***extra*)
Clase Element. Esta clase define la interfaz Element, y provee una implementación de referencia de esta interfaz.

El nombre del elemento, los nombres de los atributos y los valores de los atributos pueden ser cadenas de bytes o cadenas Unicode. *tag* es el nombre del elemento. *attrib* es un diccionario opcional que contiene los atributos del elemento. *extra* contiene atributos adicionales, dados como argumentos de palabras clave.

tag
Una cadena de caracteres que identifica qué tipo de datos representa este elemento (el tipo de elemento, en otras palabras).

text
tail
Estos atributos pueden utilizarse para contener datos adicionales asociados al elemento. Sus valores suelen ser cadenas, pero pueden ser cualquier objeto específico de la aplicación. Si el elemento se crea a partir de un archivo XML, el atributo *text* contiene el texto entre la etiqueta inicial del elemento y su primera etiqueta hija o final, o `None`, y el atributo *tail* contiene el texto entre la etiqueta final del elemento y la siguiente etiqueta, o `None`. Para los datos XML

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

el elemento *a* tiene `None` para los atributos *text* y *tail*, el elemento *b* tiene *text* `"1"` y *tail* `"4"`, el elemento *c* tiene *text* `"2"` y *tail* `None`, y el elemento *d* tiene *text* `None` y *tail* `"3"`.

Para recoger el texto interior de un elemento, véase *itertext()*, por ejemplo `"".join(element.itertext())`.

Las aplicaciones pueden almacenar objetos arbitrarios en estos atributos.

attrib
Un diccionario que contiene los atributos del elemento. Ten en cuenta que aunque el valor *attrib* es siempre un diccionario mutable real de Python, una implementación de ElementTree puede elegir utilizar otra representación interna, y crear el diccionario sólo si alguien lo pide. Para aprovechar este tipo de implementaciones, utiliza los métodos de diccionario que aparecen a continuación siempre que sea posible.

Los siguientes métodos tipo diccionario funcionan con los atributos de los elementos.

clear()
Restablece un elemento. Esta función elimina todos los sub-elementos, borra todos los atributos y establece los atributos de texto y cola como `None`.

get(*key*, *default=None*)
Obtiene el atributo del elemento llamado *key*.

Retorna el valor del atributo, o *default* si el atributo no fue encontrado.

items()
Retorna los atributos del elemento como una secuencia de pares (nombre, valor). Los atributos se retornan en un orden arbitrario.

keys()
Retorna los nombres de los atributos de los elementos como una lista. Los nombres se retornan en un orden arbitrario.

set(*key*, *value*)
Establecer el atributo *key* en el elemento a *value*.

Los siguientes métodos actúan sobre los hijos del elemento (sub-elementos).

append(*subelement*)
Añade el elemento *subelement* al final de la lista interna de sub-elementos de este elemento. Lanza *TypeError* si *subelement* no es un *Element*.

extend(*subelements*)
Añade *subelements* de un objeto de secuencia con cero o más elementos. Lanza *TypeError* si un subelemento no es un *Element*.

Nuevo en la versión 3.2.

find(*match*, *namespaces=None*)
Encuentra el primer sub-elemento que coincide con *match*. *match* puede ser un nombre de etiqueta o un *path*. Retorna una instancia de elemento o `None`. *namespaces* es un mapeo opcional del prefijo del espacio de nombres al nombre completo. Pasa `''` como prefijo para mover todos los nombres de etiquetas sin prefijo en la expresión al espacio de nombres dado.

findall(*match*, *namespaces=None*)
Encuentra todos los sub-elementos coincidentes, por nombre de etiqueta o *path*. Retorna una lista que contiene todos los elementos coincidentes en el orden del documento. *namespaces* es un mapeo opcional del prefijo del espacio de nombres al nombre completo. Pasa `''` como prefijo para mover todos los nombres de etiquetas sin prefijo en la expresión al espacio de nombres dado.

findtext(*match*, *default=None*, *namespaces=None*)
Busca el texto del primer sub-elemento que coincida con *match*. *match* puede ser un nombre de etiqueta o un *path*. Retorna el contenido del texto del primer elemento que coincida, o *default* si no se encuentra ningún elemento. Tenga en cuenta que si el elemento coincidente no tiene contenido de texto se devuelve una cadena vacía. *namespaces* es un mapeo opcional del prefijo del espacio de nombres al nombre completo. Pasa `''` como prefijo para mover todos los nombres de etiquetas sin prefijo en la expresión al espacio de nombres dado.

insert(*index*, *subelement*)
Inserta *subelement* en la posición dada en este elemento. Lanza *TypeError* si *subelement* no es un *Element*.

iter(*tag=None*)
Crea un árbol *iterator* con el elemento actual como raíz. El iterador itera sobre este elemento y todos los elementos por debajo de él, en el orden del documento (profundidad primero). Si *tag* no es `None` o `'*'`, sólo los elementos cuya etiqueta es igual a *tag* son retornados por el iterador. Si la estructura del árbol se modifica durante la iteración, el resultado es indefinido.

Nuevo en la versión 3.2.

iterfind(*match*, *namespaces=None*)
Encuentra todos los subelementos que coinciden, por nombre de etiqueta o *ruta*. Retorna un iterable con todos los elementos coincidentes en el orden del documento. *namespaces* es un mapeo opcional del prefijo del espacio de nombres al nombre completo.

Nuevo en la versión 3.2.

itertext()
Crea un iterador de texto. El iterador hace un bucle sobre este elemento y todos los subelementos, en el orden del documento, y retorna todo el texto interior.

Nuevo en la versión 3.2.

makeelement(*tag*, *attrib*)
Crea un nuevo objeto elemento del mismo tipo que este elemento. No llame a este método, utilice la función de fábrica *SubElement()* en su lugar.

remove(*subelement*)
Elimina el *subelement* del elemento. A diferencia de los métodos find*, este método compara los elementos basándose en la identidad de la instancia, no en el valor de la etiqueta o el contenido.

Los objetos *Element* también soportan los siguientes métodos de tipo secuencia para trabajar con subelementos: `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()`.

Precaución: Los elementos que no tengan subelementos serán evaluados como `False`. Este comportamiento cambiará en futuras versiones. Utilizar en su lugar el test específico `len(elem)` o `elem is None`.

```
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")
```

Antes de Python 3.8, el orden de serialización de los atributos XML de los elementos se hacía predecible artificialmente ordenando los atributos por su nombre. Basado en el ordenamiento -ahora garantizado- de los diccionarios, este reordenamiento arbitrario fue eliminado en Python 3.8 para preservar el orden en que los atributos fueron originalmente analizados o creados por el código del usuario.

En general, el código del usuario debería intentar no depender de un orden específico de los atributos, dado que el XML Information Set excluye explícitamente el orden de los atributos para transmitir información. El código debe estar preparado para hacer frente a cualquier orden en la entrada. En los casos en los que se requiere una salida XML determinista, por ejemplo, para la firma criptográfica o los conjuntos de datos de prueba, la serialización canónica está disponible con la función *canonicalize()*.

En los casos en los que la salida canónica no es aplicable, pero un orden de atributos específico sigue siendo deseable en la salida, el código debe tratar de crear los atributos directamente en el orden deseado, para evitar desajustes perceptivos para los lectores del código. En los casos en que esto es difícil de lograr, una receta como la siguiente se puede aplicar antes de la serialización para hacer cumplir un orden independientemente de la creación de elementos:

```
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)
```

Objetos ElementTree

class xml.etree.ElementTree.**ElementTree**(*element=None*, *file=None*)

Clase envoltorio de un ElementTree. Esta clase representa una jerarquía de elementos completa, y añade algún soporte extra para la serialización hacia y desde XML estándar.

element es el elemento raíz. El árbol se inicializa con el contenido del *file* XML si se da.

_setroot(*element*)

Reemplaza el elemento raíz de este árbol. Esto descarta el contenido actual del árbol, y lo reemplaza con el elemento dado. Utilícelo con cuidado. *element* es una instancia de elemento.

find(*match*, *namespaces=None*)

Igual que *Element.find()*, empezando por la raíz del árbol.

findall(*match*, *namespaces=None*)

Igual que *Element.findall()*, empezando por la raíz del árbol.

findtext(*match*, *default=None*, *namespaces=None*)

Igual que *Element.findtext()*, empezando por la raíz del árbol.

getroot()

Retorna el elemento raíz de este árbol.

iter(*tag=None*)

Crea y retorna un iterador de árbol para el elemento raíz. El iterador recorre todos los elementos de este árbol, en orden de sección. *tag* es la etiqueta a buscar (por defecto devuelve todos los elementos).

iterfind(*match*, *namespaces=None*)

Igual que *Element.iterfind()*, empezando por la raíz del árbol.

Nuevo en la versión 3.2.

parse(*source*, *parser=None*)

Carga una sección XML externa en este árbol de elementos. *source* es un nombre de archivo o un *file object*. *parser* es una instancia opcional del analizador. Si no se da, se utiliza el analizador estándar *XMLParser*. Retorna el elemento raíz de la sección.

write(*file*, *encoding='us-ascii'*, *xml_declaration=None*, *default_namespace=None*, *method='xml'*, ***, *short_empty_elements=True*)

Escribe el árbol de elementos en un archivo, como XML. *file* es un nombre de archivo, o un *file object* abierto para escritura. *encoding*[1] es la codificación de salida (por defecto es US-ASCII). *xml_declaration* controla si se debe añadir una declaración XML al archivo. Utilice `False` para nunca, `True` para siempre, `None` para sólo si no es US-ASCII o UTF-8 o Unicode (por defecto es `None`). *default_namespace* establece el espacio de nombres XML por defecto (para «xmlns»). *method* es `"xml"`, `"html"` o `"text"` (por defecto es `"xml"`). El parámetro *short_empty_elements* controla el formato de los elementos sin contenido. Si es `True` (por defecto), se emiten como una sola etiqueta autocerrada, de lo contrario se emiten como un par de etiquetas de inicio/fin.

La salida es una cadena de caracteres (*str*) o binaria (*bytes*). Esto es controlado por el argumento *encoding*. Si *encoding* es `unicode`, la salida es una cadena de caracteres; en caso contrario, es binaria. Tenga en cuenta que esto puede entrar en conflicto con el tipo de *file* si es un *file object* abierto; asegúrese de no intentar escribir una cadena en un flujo binario y viceversa.

Nuevo en la versión 3.4: El parámetro *short_empty_elements*.

Distinto en la versión 3.8: El método *write()* ahora conserva el orden de los atributos especificado por el usuario.

Este es el archivo XML que será manipulado:

```
<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>
```

Ejemplo de cambio del atributo «target» de cada enlace en el primer párrafo:

```
>>> 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")
```

Objetos QName

class xml.etree.ElementTree.**QName**(*text_or_uri*, *tag=None*)

QName wrapper. Se puede utilizar para envolver un valor de atributo QName, con el fin de obtener un manejo adecuado del espacio de nombres en la salida. *text_or_uri* es una cadena de caracteres que contiene el valor QName, en la forma {uri}local, o, si se da el argumento tag, la parte URI de un QName. Si se da *tag*, el primer argumento se interpreta como un URI, y este argumento se interpreta como un nombre local. Las instancias de *QName* son opacas.

Objetos TreeBuilder

class xml.etree.ElementTree.**TreeBuilder**(*element_factory=None*, ***, *comment_factory=None*, *pi_factory=None*, *insert_comments=False*, *insert_pis=False*)

Constructor genérico de estructuras de elementos. Este constructor convierte una secuencia de llamadas a los métodos start, data, end, comment y pi en una estructura de elementos bien formada. Puedes utilizar esta clase para construir una estructura de elementos utilizando un analizador XML personalizado, o un analizador para algún otro formato similar a XML.

element_factory, cuando se da, debe ser un callable que acepta dos argumentos posicionales: una etiqueta y un diccionario de atributos. Se espera que retorne una nueva instancia de elemento.

Las funciones *comment_factory* y *pi_factory*, cuando se dan, deben comportarse como las funciones *Comment()* y *ProcessingInstruction()* para crear comentarios e instrucciones de procesamiento. Si no se dan, se utilizarán las fábricas por defecto. Cuando *insert_comments* y/o *insert_pis* es verdadero, los comentarios/pis se insertarán en el árbol si aparecen dentro del elemento raíz (pero no fuera de él).

close()

Vacía los buffers del constructor y retorna el elemento del documento de nivel superior. Retorna una instancia de *Element*.

data(*data*)
Añade texto al elemento actual. *data* es una cadena. Debe ser una cadena de bytes o una cadena Unicode.

end(*tag*)
Cierra el elemento actual. *tag* es el nombre del elemento. Retorna el elemento cerrado.

start(*tag*, *attrs*)
Abre un nuevo elemento. *tag* es el nombre del elemento. *attrs* es un diccionario que contiene los atributos del elemento. Retorna el elemento abierto.

comment(*text*)
Crea un comentario con el *texto* dado. Si `insert_comments` es verdadero, esto también lo añadirá al árbol.

Nuevo en la versión 3.8.

pi(*target*, *text*)
Crea un comentario con el nombre *target* y el *texto* dados. Si `insert_pis` es verdadero, esto también lo añadirá al árbol.

Nuevo en la versión 3.8.

Además, un objeto *TreeBuilder* personalizado puede proporcionar los siguientes métodos:

doctype(*name*, *pubid*, *system*)
Maneja una declaración de doctype. *name* es el nombre del doctype. *pubid* es el identificador público. *system* es el identificador del sistema. Este método no existe en la clase por defecto *TreeBuilder*.

Nuevo en la versión 3.2.

start_ns(*prefix*, *uri*)
Se llama cada vez que el analizador encuentra una nueva declaración de espacio de nombres, antes de la llamada de retorno `start()` para el elemento de apertura que lo define. *prefix* es `''` para el espacio de nombres por defecto y el nombre del prefijo del espacio de nombres declarado en caso contrario. *uri* es el URI del espacio de nombres.

Nuevo en la versión 3.8.

end_ns(*prefix*)
Se llama después de la llamada de retorno `end()` de un elemento que declaró un mapeo de prefijo de espacio de nombres, con el nombre del *prefijo* que salió del ámbito.

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_attrs=None*, *exclude_tags=None*)
Un escritor C14N 2.0. Los argumentos son los mismos que para la función *canonicalize()*. Esta clase no construye un árbol, sino que traduce los eventos de devolución de llamada directamente a una forma serializada utilizando la función *write*.

Nuevo en la versión 3.8.

Objetos XMLParser

class xml.etree.ElementTree.**XMLParser**(*, *target=None*, *encoding=None*)

Esta clase es el bloque de construcción de bajo nivel del módulo. Utiliza *xml.parsers.expat* para un análisis eficiente de XML basado en eventos. Puede ser alimentada con datos XML de forma incremental con el método *feed()*, y los eventos de análisis se traducen en una API push - invocando callbacks en el objeto *target*. Si se omite *target*, se utiliza la clase estándar *TreeBuilder*. Si se da *encoding*[1], el valor anula la codificación especificada en el archivo XML.

Distinto en la versión 3.8: Los parámetros son ahora *keyword-only*. El argumento *html* ya no se admite.

close()

Finaliza la alimentación de datos al analizador. Retorna el resultado de llamar al método `close()` del *target* pasado durante la construcción; por defecto, es el elemento del documento de nivel superior.

feed(*data*)

Introduce los datos en el analizador sintáctico. *data* son datos codificados.

XMLParser.feed() llama al método `start(tag, attrs_dict)` de *target* para cada etiqueta de apertura, a su método `end(tag)` para cada etiqueta de cierre, y los datos son procesados por el método `data(data)`. Para más métodos de callback soportados, véase la clase *TreeBuilder*. *XMLParser.close()* llama al método `close()` de *target*. *XMLParser* puede utilizarse no sólo para construir una estructura de árbol. Este es un ejemplo de contar la profundidad máxima de un archivo XML:

```
>>> 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
```

Objetos XMLPullParser

class xml.etree.ElementTree.**XMLPullParser**(*events=None*)

Un analizador sintáctico pull adecuado para aplicaciones no bloqueantes. Su API de entrada es similar a la de *XMLParser*, pero en lugar de enviar llamadas a un objetivo de devolución de llamada, *XMLPullParser* recoge una lista interna de eventos de análisis y permite al usuario leer de ella. *events* son una secuencia de eventos a reportar. Los eventos soportados son las cadenas `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` y `"end-ns"` (los eventos «ns» se utilizan para obtener información detallada del espacio de nombres). Si se omite *events*, sólo se informará de los eventos `"end"`.

feed(*data*)

Introduce los datos de los bytes dados en el analizador.

close()

Señala al analizador que el flujo de datos ha terminado. A diferencia de *XMLParser.close()*, este método siempre retorna *None*. Cualquier evento que no haya sido recuperado cuando el analizador se cierra puede ser leído con *read_events()*.

read_events()

Retorna un iterador sobre los eventos que se han encontrado en los datos alimentados al analizador. El iterador retorna pares `(event, elem)`, donde *event* es una cadena de caracteres que representa el tipo de evento (por ejemplo, `"fin"`) y *elem* es el objeto *Element* encontrado, u otro valor de contexto como el siguiente.

- `start`, `end`: el Element actual.
- `comment`, `pi`: el comentario / la instrucción de procesamiento actual
- `start-ns`: una tupla `(prefix, uri)` que nombra el mapeo del espacio de nombres declarado.
- `end-ns`: *None* (esto puede cambiar en una versión futura)

Los eventos proporcionados en una llamada anterior a *read_events()* no serán retornados nuevamente. Los eventos se consumen de la cola interna sólo cuando se recuperan del iterador, por lo que múltiples lectores iterando en paralelo sobre iteradores obtenidos de *read_events()* tendrán resultados impredecibles.

Nota: *XMLPullParser* sólo garantiza que ha visto el carácter «>» de una etiqueta de inicio cuando emite un evento «start», por lo que los atributos están definidos, pero el contenido de los atributos text y tail está indefinido en ese momento. Lo mismo ocurre con los hijos del elemento; pueden estar presentes o no.

Si necesita un elemento totalmente poblado, busque los eventos «end» en su lugar.

Nuevo en la versión 3.4.

Distinto en la versión 3.8: Los eventos `comment` y `pi` han sido añadidos.

Excepciones

class xml.etree.ElementTree.**ParseError**

Error de análisis de XML, lanzado por los distintos métodos de análisis de este módulo cuando el análisis falla. La representación en cadena de caracteres de una instancia de esta excepción contendrá un mensaje de error fácil de entender. Además, tendrá los siguientes atributos disponibles:

code

Un código de error numérico del analizador sintáctico expat. Consulte la documentación de *xml.parsers.expat* para ver la lista de códigos de error y sus significados.

position

Una tupla de números de *line*, *column*, que especifica dónde se produjo el error.

Notas al pie de página

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 documento 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 método 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; está 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, lanza 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 (sección 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 (sección 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 (sección 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 definirlas. 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
DOMImplementation	*Objetos DOMImplementation*	Interfaz para las implementaciones subyacentes.
Node	*Objetos Nodo*	Interfaz base para la mayoría de objetos en un documento.
NodeList	*Objetos NodeList*	Interfaz para una secuencia de nodos.
DocumentType	*Objetos DocumentType*	Información acerca de la declaraciones necesarias para procesar un documento.
Document	*Objetos Documento*	Objeto que representa un documento entero.
Element	*Objetos Elemento*	Nodos elemento en la jerarquía del documento.
Attr	*Objetos Atributo*	Nodos de los valores de los atributos en los elementos nodo.
Comment	*Objetos Comentario*	Representación de los comentarios en el documento fuente.
Text	*Objetos Texto y CDATASection*	Nodos con contenido textual del documento.
ProcessingInstruction	*Objetos ProcessingInstruction*	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 tienes 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 una 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 el. 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 lanzado.

`Element`.**removeAttributeNode**(*oldAttr*)
Remueve y retorna *oldAttr* de la lista de atributos, si está presenta. Si *oldAttr* no está presente, *NotFoundErr* es lanzado.

`Element`.**removeAttributeNS**(*namespaceURI*, *localName*)
Remueve un atributo por nombre (*name*). Note que esto usa un *localName*, no un *qname*. Ninguna excepción es lanzada 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 `namesapceURI` y `localName` coinciden. Si un reemplazo ocurre, el viejo atributo será retornado. Si *newAttr* está en uso, *InuseAttributeErr* será lanzado.

`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	*DomstringSizeErr*
HIERARCHY_REQUEST_ERR	*HierarchyRequestErr*
INDEX_SIZE_ERR	*IndexSizeErr*
INUSE_ATTRIBUTE_ERR	*InuseAttributeErr*
INVALID_ACCESS_ERR	*InvalidAccessErr*
INVALID_CHARACTER_ERR	*InvalidCharacterErr*
INVALID_MODIFICATION_ERR	*InvalidModificationErr*
INVALID_STATE_ERR	*InvalidStateErr*
NAMESPACE_ERR	*NamespaceErr*
NOT_FOUND_ERR	*NotFoundErr*
NOT_SUPPORTED_ERR	*NotSupportedErr*
NO_DATA_ALLOWED_ERR	*NoDataAllowedErr*
NO_MODIFICATION_ALLOWED_ERR	*NoModificationAllowedErr*
SYNTAX_ERR	*SyntaxErr*
WRONG_DOCUMENT_ERR	*WrongDocumentErr*

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	bool o int
int	int
long int	int
unsigned int	int
DOMString	str o bytes
null	None

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 si 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"
```

When you are finished with a DOM tree, you may optionally call the unlink() method to encourage early cleanup of the now-unneeded objects. unlink() is an *xml.dom.minidom*-specific extension to the DOM API that renders the node and its descendants essentially useless. Otherwise, Python's garbage collector will eventually take care of the objects in the tree.

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=''*, *encoding=None*, *standalone=None*)

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.

Similarly, explicitly stating the *standalone* argument causes the standalone document declarations to be added to the prologue of the XML document. If the value is set to `True`, `standalone="yes"` is added, otherwise it is set to `"no"`. Not stating the argument will omit the declaration from the document.

Distinto en la versión 3.8: El método *writexml()* ahora conserva el orden de los atributos especificado por el usuario.

Distinto en la versión 3.9: Se agregó el parámetro *standalone*.

Node.**toxml**(*encoding=None*, *standalone=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.

El argumento *standalone* se comporta exactamente como en *writexml()*.

Distinto en la versión 3.8: El método *toxml()* ahora conserva el orden de los atributos especificado por el usuario.

Distinto en la versión 3.9: Se agregó el parámetro *standalone*.

Node.**toprettyxml**(*indent='\t'*, *newl='\n'*, *encoding=None*, *standalone=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()*.

El argumento *standalone* se comporta exactamente como en *writexml()*.

Distinto en la versión 3.8: El método *toprettyxml()* ahora conserva el orden de los atributos especificado por el usuario.

Distinto en la versión 3.9: Se agregó el parámetro *standalone*.

[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.

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>
<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))
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 construidos . 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) *strnig* (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 es 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 construidos 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 `créate_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 SAD y úsalo para analizar un documento. El documento, aprobado como *filename_or_steam*, 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 adicionar 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

La API de SAX define cuatro tipos de manejadores: manejadores de contenido, manejadores de DTD, manejadores de errores y resolvedores de entidades. Normalmente, las aplicaciones solo necesitan implementar las interfaces cuyos eventos les interesan; pueden implementar las interfaces en un solo objeto o en múltiples objetos. Las implementaciones de los manejadores deben heredar de las clases base provistas en el módulo *xml.sax.handler*, de modo que todos los métodos obtengan implementaciones por defecto.

class xml.sax.handler.**ContentHandler**
Esta es la principal interfaz de devolución de llamada en el SAX, y la más importante para las aplicaciones. El orden de los eventos en esta interfaz refleja el orden de la información en el documento.

class xml.sax.handler.**DTDHandler**
Manejar eventos de DTD.

Esta interfaz especifica solo los eventos DTD necesarios para el análisis básico (entidades y atributos no analizados).

class xml.sax.handler.**EntityResolver**
Interfaz básica para resolver entidades. Si creas un objeto implementando esta interfaz, y luego registras el objeto con tu Parser, el parser (analizador) invocara al método en tu objeto para resolver todas las entidades externas.

class xml.sax.handler.**ErrorHandler**
Interfaz utilizada por el parser para presentar mensajes de error y advertencia a la aplicación. Los métodos de este objeto controlan si los errores se convierten inmediatamente en excepciones o se manejan de alguna otra manera.

class xml.sax.handler.**LexicalHandler**
Interface usada por el parser para representar eventos de baja frecuencia que pueden no ser de interés para muchas aplicaciones.

Además de estas clases, *xml.sax.handler* proporciona constantes simbólicas para los nombres de las características y propiedades.

xml.sax.handler.**feature_namespaces**

value: "http://xml.org/sax/features/namespaces"
true: Realizar el procesamiento del Namespace (espacio de nombres).
false: Opcionalmente no realizar el procesamiento del Namespace (espacio de nombres) (implica prefijos de espacio de nombres; por defecto).
acceso: (parsing) solo de lectura; (not parsing) lectura/escritura

xml.sax.handler.**feature_namespace_prefixes**

value: "http://xml.org/sax/features/namespace-prefixes"
true: Reporte de los nombres prefijados originales y los atributos utilizados para las declaraciones del Namespace (espacio de nombres).
false: No informar de los atributos utilizados para las declaraciones del Namespace (espacio de nombres), y opcionalmente no informar de los nombres prefijados originales (por defecto).
acceso: (parsing) solo de lectura; (not parsing) lectura/escritura

xml.sax.handler.**feature_string_interning**

value: "http://xml.org/sax/features/string-interning"
true: Todos los nombres de elementos, prefijos, nombres de atributos, URIs del Namespace (espacio de nombres) y nombres locales son internados usando la función interna incorporada.
false: Los nombres no están necesariamente internados, aunque pueden estarlo (por defecto).
acceso: (parsing) solo de lectura; (not parsing) lectura/escritura

xml.sax.handler.**feature_validation**

value: "http://xml.org/sax/features/validation"
true: Reportar de todos los errores de validación (implica entidades generales externas y entidades de parámetros externos).
false: No informe de los errores de validación.
acceso: (parsing) solo de lectura; (not parsing) lectura/escritura

xml.sax.handler.**feature_external_ges**

value: "http://xml.org/sax/features/external-parameter-entities"

true: Incluye todas las entidades externas generales de texto (text).
false: No incluya entidades generales externas.
acceso: (parsing) solo de lectura; (not parsing) lectura/escritura

xml.sax.handler.**feature_external_pes**

value: "http://xml.org/sax/features/external-parameter-entities"
true: Incluye todas las entidades paramétricas externas, incluyendo el subconjunto DTD externo.
false: No incluya ninguna entidad paramétrica externa, ni siquiera el subconjunto DTD externo.
acceso: (parsing) solo de lectura; (not parsing) lectura/escritura

xml.sax.handler.**all_features**
Lista de todas las características.

xml.sax.handler.**property_lexical_handler**

value: "http://xml.org/sax/properties/lexical-handler"
data type: xml.sax.sax2lib.LexicalHandler (no soportado en Python 2)
descripción: Un handler (manipulador) de extensión opcional para eventos léxicos como los comentarios.
acceso: read/write (leer/escribir)

xml.sax.handler.**property_declaration_handler**

value: "http://xml.org/sax/properties/declaration-handler"
data type: xml.sax.sax2lib.DeclHandler (no soportado en Python 2)
description: Un gestor de extensión opcional para eventos relacionados con la DTD que no sean anotaciones y entidades no preparadas.
acceso: read/write (leer/escribir)

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)
descripción: Cuando se analiza, el nodo DOM actual que se visita si se trata de un iterador DOM; cuando no se analiza, el nodo DOM raíz para la iteración.
acceso: (parsing) solo de lectura; (not parsing) lectura/escritura

xml.sax.handler.**property_xml_string**

value: "http://xml.org/sax/properties/xml-string"
data type: Bytes
description: La cadena literal de caracteres que fue la fuente del evento actual.
acceso: solo-lectura

xml.sax.handler.**all_properties**
Lista de todos los nombres de propiedades conocidas.

20.10.1 Objetos ContentHandler

Se espera que los usuarios subclasifiquen *ContentHandler* para apoyar su aplicación. Los siguientes métodos son llamados por el analizador en los eventos apropiados en el documento de entrada:

ContentHandler.**setDocumentLocator**(*locator*)
Invocado por el parser (analizador) para dar a la aplicación un localizador para determinar el origen de los eventos del documento.

Se recomienda encarecidamente a los parsers (analizadores) SAX (aunque no es absolutamente necesario) que proporcionen un localizador: si lo hace, debe proporcionar el localizador a la aplicación invocando este método antes de invocar cualquiera de los otros métodos de la interfaz DocumentHandler.

El localizador permite a la aplicación determinar la posición final de cualquier evento relacionado con el documento, incluso si el analizador no informa de un error. Normalmente, la aplicación utilizará esta información para informar de sus propios errores (como el contenido de los caracteres que no coinciden con las reglas de negocio de la aplicación). Es probable que la información retornada por el localizador no sea suficiente para su uso con un motor de búsqueda.

Tenga en cuenta que el localizador retornará la información correcta solo durante la invocación de los eventos en esta interfaz. La aplicación no debe intentar utilizarlo en ningún otro momento.

ContentHandler.**startDocument**()
Recibir la notificación del inicio de un documento.

El parser (analizador) SAX invocará este método solo una vez, antes que cualquier otro método en esta interfaz o en DTDHandler (excepto *setDocumentLocator()*).

ContentHandler.**endDocument**()
Recibir una notificación del final de un documento.

El analizador (parser) SAX invocará este método solo una vez, y será el último método invocado durante el análisis. El analizador no invocará este método hasta que no haya abandonado el análisis sintáctico (debido a un error irrecuperable) o haya llegado al final de la entrada.

ContentHandler.**startPrefixMapping**(*prefix*, *uri*)
Comienza el alcance de un mapeo del prefijo-URI Namespace.

La información de este evento no es necesaria para el procesamiento normal del Namespace (espacio de nombres): el lector de XML SAX sustituirá automáticamente los prefijos de los nombres de elementos y atributos cuando se active la función `feature_namespaces` default (el valor por defecto).

Sin embargo, hay casos en que las aplicaciones necesitan utilizar prefijos en los datos de los caracteres o en los valores de los atributos, en los que no se pueden expandir automáticamente de forma segura; los eventos *startPrefixMapping()* y *endPrefixMapping()* suministran la información a la aplicación para expandir los prefijos en esos contextos por sí mismos, si es necesario.

Note que los eventos *startPrefixMapping()* y *endPrefixMapping()* no están garantizados de estar apropiadamente anidados en relación a cada uno: todos los eventos *startPrefixMapping()* ocurrirán antes del correspondiente evento *startElement()*, y todos los eventos *endPrefixMapping()* ocurrirán después del correspondiente evento *endElement()*, pero su orden no está garantizado.

ContentHandler.**endPrefixMapping**(*prefix*)
Terminar con el alcance de un mapeo de prefijos-URI.

Ver *startPrefixMapping()* para más detalles. Este evento siempre ocurrirá después del correspondiente evento *endElement()*, pero el orden de los eventos *endPrefixMapping()* no está garantizado.

ContentHandler.**startElement**(*name*, *attrs*)
Señala el inicio de un elemento en modo no espacial.

El parámetro *name* contiene el nombre XML 1.0 en bruto del tipo de elemento como una cadena y el parámetro *attrs* contiene un objeto de la interfaz `Attributes` (véase *La Interfaz Attributes*) que contiene los atributos del elemento. El objeto pasado como *attrs* puede ser reutilizado por el parser (analizador); mantener una referencia a él no es una forma fiable de conservar una copia de los atributos. Para guardar una copia de los atributos, usa el método *copy()* del objeto *attrs*.

ContentHandler.**endElement**(*name*)
Señala el final de un elemento en modo no espacial.

El parámetro *name* contiene el nombre del tipo de elemento, al igual que con el evento *startElement()*.

ContentHandler.**startElementNS**(*name*, *qname*, *attrs*)
Señala el inicio de un elemento en el modo de Namespace (espacio de nombres).

El parámetro *name* contiene el nombre del tipo de elemento como una tupla `(uri, localname)`, el parámetro *qname* contiene el nombre XML 1.0 en bruto usado en el documento fuente, y el parámetro *attrs* contiene una instancia de la interfaz `AttributesNS` (ver *La Interfaz AttributesNS*) que contiene los atributos del elemento. Si no hay ningún Namespace (espacio de nombres) asociado al elemento, el componente *uri* de *name* será `Ninguno`. El objeto pasado como *attrs* puede ser reutilizado por el analizador; mantener una referencia a él no es una forma fiable de conservar una copia de los atributos. Para guardar una copia de los atributos, usa el método `copy`()` del objeto *attrs*.

Los parsers (analizadores) pueden establecer el parámetro *qname* como `None`, a menos que se active la función `feature_namespace_prefixes`.

ContentHandler.**endElementNS**(*name*, *qname*)
Señala el final de un elemento en el modo de namespace.

El parámetro *name* contiene el nombre del tipo de elemento, al igual que con el método *startElementNS()*, así como el parámetro *qname*.

ContentHandler.**characters**(*content*)
Recibir la notificación de los datos de los caracteres.

El Parser (analizador) llamará a este método para informar de cada dato de carácter. Los parsers (analizadores) SAX pueden retornar todos los datos de caracteres contiguos en un solo trozo, o pueden dividirlos en varios trozos; sin embargo, todos los caracteres de un mismo evento deben provenir de la misma entidad externa para que el Localizador proporcione información útil.

content puede ser una cadena o una instancia de bytes; el módulo lector de «Expat» siempre produce cadenas.

Nota: La anterior interfaz SAX 1 proporcionada por el Grupo de Interés Especial de Python XML utilizaba una interfaz más parecida a Java para este método. Dado que la mayoría de los parsers (analizadores) utilizados de Python no aprovecharon la interfaz más antigua, se eligió la firma más simple para reemplazarla. Para convertir el código antiguo en la nueva interfaz, se utilizó *content* en lugar de cortar el contenido con los antiguos parámetros *offset* y *length*.

ContentHandler.**ignorableWhitespace**(*whitespace*)
Recibir notificación de espacios en blanco ignorables en el contenido de los elementos.

Los parsers validadores deben utilizar este método para informar de cada trozo de espacio en blanco desconocido (véase la recomendación W3C XML 1.0, sección 2.10): los analizadores no validadores también pueden utilizar este método si son capaces de analizar y utilizar modelos de contenido.

Los parsers (analizadores) SAX pueden retornar todos los espacios blancos contiguos en un solo trozo, o pueden dividirlo en varios trozos; sin embargo, todos los caracteres de un mismo evento deben provenir de la misma entidad externa, para que el Localizador proporcione información útil.

ContentHandler.**processingInstruction**(*target*, *data*)
Recibir la notificación de una instrucción de procesamiento.

El Parser (analizador) invocará este método una vez por cada instrucción de procesamiento encontrada: nótese que las instrucciones de procesamiento pueden ocurrir antes o después del elemento principal del documento.

Un parser (analizador) SAX nunca debe informar de una declaración XML (XML 1.0, sección 2.8) o una declaración de texto (XML 1.0, sección 4.3.1) utilizando este método.

ContentHandler.**skippedEntity**(*name*)
Recibir la notificación de una entidad salteada.

El Parser invocará este método una vez por cada entidad omitida. Los procesos no validadores pueden omitir entidades si no han visto las declaraciones (porque, por ejemplo, la entidad fue declarada en un subconjunto DTD externo). Todos los procesos pueden omitir entidades externas, dependiendo de los valores de las propiedades `feature_external_ges` y `feature_external_pes`.

20.10.2 Objetos DTDHandler

DTDHandler la instancia provee los siguientes métodos:

DTDHandler.**notationDecl**(*name*, *publicId*, *systemId*)
Manejar un evento de declaración de anotación.

DTDHandler.**unparsedEntityDecl**(*name*, *publicId*, *systemId*, *ndata*)
Manejar un evento de declaración de entidad no preparada.

20.10.3 Objetos EntityResolver

EntityResolver.**resolveEntity**(*publicId*, *systemId*)
Resolver el identificador de sistema de una entidad y retornar ya sea el identificador de sistema para leerlo como una cadena, o una fuente de entrada para leerlo. La implementación por defecto retorna *systemId*.

20.10.4 Objetos ErrorHandler

Los objetos con esta interfaz se usan para recibir información de error y advertencia del *XMLReader*. Si creas un objeto que implemente esta interfaz, y luego registras el objeto con tu *XMLReader*, el parser (analizador) invocará a los métodos de tu objeto para informar de todas las advertencias y errores. Hay tres niveles de errores disponibles: advertencias, (posiblemente) errores recuperables y errores no recuperables. Todos los métodos toman un `SAXParseException` como único parámetro. Los errores y advertencias pueden ser convertidos en una excepción lanzando el objeto de excepción pasado.

ErrorHandler.**error**(*exception*)
Invocación cuando el parser (analizador) encuentra un error recuperable. Si este método no lanza una excepción, el análisis sintáctico puede continuar, pero la aplicación no debe esperar más información del documento. Permitir que el parser (analizador) continúe puede permitir que se descubran errores adicionales en el documento de entrada.

ErrorHandler.**fatalError**(*exception*)
Invocación cuando el parser (analizador) encuentra un error del que no puede recuperarse; se espera que el análisis sintáctico termine cuando vuelva este método.

ErrorHandler.**warning**(*exception*)
Invocación cuando el parser (analizador) presenta información de advertencia menor a la aplicación. Se espera que el análisis sintáctico continúe cuando vuelva este método, y la información del documento seguirá pasando a la aplicación. Si se lanza una excepción con este método, el análisis sintáctico terminará.

20.10.5 Objetos DTDHandler

Handler (manipulador) de extensión opcional para eventos léxicos SAX2.

Este handler se usa para obtener información léxica de un documento XML. La información léxica incluye información que describe la codificación del documento usado u los comentarios XML embebidos en el documento, como también una sección para el DTD y para cada sección CDTA. Estos manipuladores léxicos son usado de la misma forma que los manipuladores de contenido (content handlers).

Configurar el LexicalHandlerde un XMLReader usando el método setProperty con la propiedad identificadora `'http://xml.org/sax/properties/lexical-handler'`.

LexicalHandler.**comment**(*content*)
Reporta un comentario en cualquier lugar en el documento (incluyendo el DTD y elementos por fuera del documento).

LexicalHandler.**startDTD**(*name*, *public_id*, *system_id*)
Reporta el estado de las declaraciones del DTD en caso de tenga asociado un DTD.

LexicalHandler.**endDTD**()
Reporta el fin de la declaración del DTD.

LexicalHandler.**startCDATA**()
Reporta el inicio de la sección CDATA marcada.

El contenido de la sección marcada del CDATA se reportará a través de manipulador de caracteres.

LexicalHandler.**endCDATA**()
Reporta el final de la sección marcada del CDATA.

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 `'&amp'`, `'<'` 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 *fuente* 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 construidos 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.

[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.

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>
```

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` 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 todos 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.**AttlistDeclHandler**(*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 ExpatError

Las excepciones *`ExpatError`* tienen una serie de atributos interesantes:

ExpatError.**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, ExpatError, errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError 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.

ExpatError.**lineno**
: Número de línea en la que se detectó el error. La primera línea está numerada como `1`.

ExpatError.**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))
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 "&#0;").

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

CAPÍTULO 21

Protocolos y soporte de Internet

Los módulos descritos en este capítulo implementan protocolos de Internet y son compatibles con la tecnología relacionada. Todos están implementados 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 descripción general:

21.1 webbrowser — Controlador de navegador web conveniente

Código fuente: Lib/webbrowser.py

El módulo *webbrowser* proporciona una interfaz de alto nivel que permite mostrar documentos basados en web a los usuarios. En la mayoría de circunstancias, simplemente llamar a la función *open()* desde este módulo hará lo correcto.

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 "https://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 declaras.

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 instanciaciones 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'	MacOSXOSAScript('default')	(3)
'safari'	MacOSXOSAScript('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) Solo en la plataforma macOS.

Nuevo en la versión 3.3: Ha sido añadido soporte para Chrome/Chromium.

Aquí están algunos ejemplos simples:

```
url = 'https://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 `wsgiref` — Utilidades WSGI e implementación de referencia

Source code: Lib/wsgiref

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.

21.2.1 `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'
    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.

21.2.2 `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.

21.2.3 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()
Returns the currently set application callable.

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 sobreescribir 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.

21.2.4 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 sobreescriben 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()
```

21.2.5 `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 se puede configurar para pasar el `PATH_INFO` correcto, pero esto causa otro error donde `PATH_TRANSLATED` es incorrecto. Afortunadamente, esta variable rara vez se usa y no está garantizada por WSGI. Sin embargo, en IIS<7, la configuración solo se puede realizar en un nivel de vhost, lo que afecta a todas las demás asignaciones de scripts, muchas de las cuales se rompen cuando se exponen al error `PATH_TRANSLATED`. Por esta razón, IIS<7 casi nunca se implementa con la solución (incluso IIS7 rara vez lo usa porque todavía no tiene una interfaz de usuario).

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.BufferedIOBase*.

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 cabecera `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()
Set the `environ` attribute to a fully populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the *get_stdin()*, *get_stderr()*, and *add_cgi_vars()* methods and the *wsgi_file_wrapper* attribute. It also inserts a `SERVER_SOFTWARE` key if not present, as long as the *origin_server* attribute is a true value and the *server_software* attribute is set.

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.

21.2.6 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
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()
```

(continué en la próxima página)

(proviene de la página anterior)

```
    except KeyboardInterrupt:
        print("Shutting down.")
        httpd.server_close()
```

21.3 `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

21.4 `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 actuar como un *gestor de contexto*, y tiene las propiedades *url*, *headers* y *status*. Véase *`urllib.response.addinfourl`* para más detalles sobre estas propiedades.

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.HAS_SNI* 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.

Distinto en la versión 3.10: La conexión HTTPS ahora envía una extensión ALPN con indicador de protocolo `http/1.1` cuando no se proporciona *context*. El *context* personalizado debe establecer protocolos ALPN con `set_alpn_protocol()`.

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. *handler*s 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 *handler*s, a no ser que el *handler*s 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 retorna 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 proxy desde la Configuración del Sistema de macOS 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 should be a dictionary, and will be treated as if *add_header()* was called with each key and value as arguments. This is often used to «spoof» the User-Agent header value, which is used by a browser to identify itself – some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as "Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11", while *urllib*'s default user agent string is "Python-urllib/2.6" (on Python 2.6). All header keys are sent in camel case.

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 macOS se obtiene la información de proxy desde el Framework de Configuración del Sistema.

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 *passwd_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 lanzará 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 lanzará 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.

21.4.1 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*)
Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the *key* collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header. Note that headers added using this method are also added to redirected requests.

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.

21.4.2 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 espacial). Tenga en cuenta que, en los siguientes, *protocol* debe ser remplazado con el protocolo actual a manejar, por ejemplo `http_response()` sería el protocolo HTTP del manejador de respuesta. También *type* debe ser remplazado 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.

21.4.3 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.

This method, if implemented, will be called by the parent *OpenerDirector*. It should return a file-like object as described in the return value of the *open()* method of *OpenerDirector*, or None. It should raise *URLError*, unless a truly exceptional thing happens (for example, *MemoryError* should not be mapped to 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()*.

21.4.4 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 predeterminadas 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".

21.4.5 Objetos HTTPCookieProcessor

Las instancias *HTTPCookieProcessor* tienen un atributo:

HTTPCookieProcessor.**cookiejar**
El *http.cookiejar.CookieJar* en el cual las cookies están almacenadas.

21.4.6 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.

21.4.7 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.

21.4.8 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.

21.4.9 Objetos AbstractBasicAuthHandler

AbstractBasicAuthHandler.**http_error_auth_reqed**(*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).

21.4.10 Objetos HTTPBasicAuthHandler

`HTTPBasicAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Reintenta la petición con la información de autenticación, si está disponible.

21.4.11 Objetos ProxyBasicAuthHandler

`ProxyBasicAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Reintenta la petición con la información de autenticación, si está disponible.

21.4.12 Objetos AbstractDigestAuthHandler

`AbstractDigestAuthHandler.`**`http_error_auth_reqed`**(*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.

21.4.13 Objetos HTTPDigestAuthHandler

`HTTPDigestAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Reintenta la petición con la información de autenticación, si está disponible.

21.4.14 Objetos ProxyDigestAuthHandler

`ProxyDigestAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Reintenta la petición con la información de autenticación, si está disponible.

21.4.15 Objetos HTTPHandler

`HTTPHandler.`**`http_open`**(*req*)
Envía una petición HTTP, que puede ser GET o POST, dependiendo de `req.has_data()`.

21.4.16 Objetos HTTPSHandler

`HTTPSHandler.`**`https_open`**(*req*)
Envía una petición HTTPS, que puede ser GET o POST, dependiendo de `req.has_data()`.

21.4.17 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*.

21.4.18 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 lanzará un *ValueError* en este caso.

21.4.19 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.

21.4.20 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*.

21.4.21 Objetos UnknownHandler

UnknownHandler.**unknown_open**()
Genera una excepción *URLError*.

21.4.22 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* lanzará 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()*.

21.4.23 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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'))
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

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/xhtm
```

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()
```

(continué en la próxima página)

(proviene de la página anterior)

```
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.

This example replaces the default *ProxyHandler* with one that uses programmatically supplied proxy URLs, and adds proxy authorization support with *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')
...
```

21.4.24 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() lanzará `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`**`()`
Limpia 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* lanzarán 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 redireccionadas 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.

21.4.25 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 acordemente. 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 *_urlopener* para ajustarlo a tus necesidades.

21.5 `urllib.response` — Clases de respuesta usadas por urllib

El módulo *urllib.response* define funciones y clases que definen una interfaz mínima *file-like*, incluyendo `read()` y `readline()`. Las funciones definidas en este módulo son usadas internamente por el módulo *urllib.request*. El objeto de respuesta típico es una instancia de *urllib.response.addinfourl*:

class urllib.response.**addinfourl**

url
:URL del recurso obtenido, comúnmente usado para determinar si se ha seguido una redirección.

headers
Retorna las cabeceras de la respuesta en la forma de una instancia de *EmailMessage*.

status
Nuevo en la versión 3.9.

Código de estado retornado por el servidor.

geturl()
Obsoleto desde la versión 3.9: Obsoleto en favor de *url*.

info()
Obsoleto desde la versión 3.9: Obsoleto en favor de *headers*.

code
Obsoleto desde la versión 3.9: Obsoleto en favor de *status*.

getcode()
Obsoleto desde la versión 3.9: Obsoleto en favor de *status*.

21.6 `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 RFC de internet sobre localizadores de recursos uniformes relativos. Admite los siguientes esquemas URL: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`, `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.

21.6.1 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*)

Analiza una URL en seis componentes, retornando una *tupla nombrada* de 6 elementos. Esto corresponde a la estructura general de una URL: `scheme://netloc/path;parameters?query#fragment`. Cada elemento de la tupla es una cadena de caracteres, 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 escape no están expandidos. Los delimitadores como se muestra anteriormente no forman parte del resultado, excepto una barra diagonal inicial en el componente *path*, el cual es conservado si está presente. Por ejemplo:

```
>>> from urllib.parse import urlparse
>>> urlparse("scheme://netloc/path;parameters?query#fragment")
ParseResult(scheme='scheme', netloc='netloc', path='/path;parameters', params='
↪',
            query='query', fragment='fragment')
>>> o = urlparse("http://docs.python.org:80/3/library/urllib.parse.html?"
...              "highlight=params#url-parsing")
>>> o
ParseResult(scheme='http', netloc='docs.python.org:80',
            path='/3/library/urllib.parse.html', params='',
            query='highlight=params', fragment='url-parsing')
>>> o.scheme
'http'
>>> o.netloc
'docs.python.org:80'
>>> o.hostname
'docs.python.org'
>>> o.port
80
>>> o._replace(fragment="").geturl()
'http://docs.python.org:80/3/library/urllib.parse.html?highlight=params'
```

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
`scheme`	0	Especificador de esquema de URL	parámetro *scheme*
`netloc`	1	Parte de ubicación de red	cadena vacía
`path`	2	Hierarchical path	cadena vacía
`params`	3	Parámetros para el último elemento de ruta de acceso	cadena vacía
`query`	4	Componente de consulta	cadena vacía
`fragment`	5	Identificador de fragmento	cadena vacía
`username`		Nombre de usuario	*None*
`password`		Contraseña	*None*
`hostname`		Nombre de host (minúsculas)	*None*
`port`		Número de puerto como entero, si está presente	*None*

La lectura del atributo `port` lanzará 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` lanzarán 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 es analizado para todos los esquemas de URL (a menos que *allow_fragment* sea falso), de acuerdo con **RFC 3986**. Anteriormente, existía una lista de permisos de esquemas que admitían 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 lanzarán *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 verdadero indica que los espacios

en blanco deben conservarse como cadenas en blanco. El valor falso 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 falso (valor predeterminado), los errores se omiten silenciosamente. Si es verdadero, los errores generan una excepción `ValueError`.

Los parámetros opcionales *encoding* y *errors* especifican cómo descodificar 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.

El argumento opcional *separator* es el símbolo que se usa para separar los argumentos de la cadena de consulta. El valor por defecto es `&`.

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 3.10: Añadido parámetro *separator* con un valor por defecto de `&`. Versiones de Python anterior a Python 3.10 permitían el uso de `;` y `&` como separadores de la cadena de consulta. Esto ha sido cambiado para aceptar sólo una llave separadora, siendo `&` el separador por defecto.

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 verdadero indica que los espacios en blanco deben conservarse como cadenas en blanco. El valor falso 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 falso (valor predeterminado), los errores se omiten silenciosamente. Si es verdadero, los errores generan una excepción `ValueError`.

Los parámetros opcionales *encoding* y *errors* especifican cómo descodificar 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.

El argumento opcional *separator* es el símbolo que se usa para separar los argumentos de la cadena de consulta. El valor por defecto es `&`.

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 3.10: Añadido parámetro *separator* con un valor por defecto de `&`. Versiones de Python anterior a Python 3.10 permitían el uso de `;` y `&` como separadores de la cadena de consulta. Esto ha sido cambiado para aceptar sólo una llave separadora, siendo `&` el separador por defecto.

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
scheme	0	Especificador de esquema de URL	parámetro *scheme*
netloc	1	Parte de ubicación de red	cadena vacía
path	2	Hierarchical path	cadena vacía
query	3	Componente de consulta	cadena vacía
fragment	4	Identificador de fragmento	cadena vacía
username		Nombre de usuario	*None*
password		Contraseña	*None*
hostname		Nombre de host (minúsculas)	*None*
port		Número de puerto como entero, si está presente	*None*

La lectura del atributo `port` lanzará 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` lanzarán 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 lanzarán *ValueError*.

Distinto en la versión 3.10: Los caracteres ASCII de nueva línea y tab se eliminan de la URL.

Distinto en la versión 3.10.12: 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, comienza con `//` 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',
...         '//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 coincidir 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
`url`	0	URL sin fragmento	cadena vacía
`fragment`	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*)

Extrae la url de una URL envuelta (es decir, una cadena de caracteres formateada como `<URL:scheme://host/path>`, `<scheme://host/path>`, `URL:scheme://host/path` or `scheme://host/path`). Si *url* no es una URL envuelta, se retorna sin cambios.

21.6.2 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.

21.6.3 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 decodificación 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

21.6.4 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.

21.6.5 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*)
Reemplaza caracteres especiales en *string* con la secuencia de 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 un objeto *str* o *bytes*.

Distinto en la versión 3.7: Se ha movido de **RFC 2396** a **RFC 3986** para citar cadenas URL. Ahora se incluye «~» en el conjunto de caracteres reservados.

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 espacios con signos más, como se requiere para citar valores de formularios HTML al momento de construir una cadena de consulta que será parte de una URL. Los signos más en la cadena de caracteres original se escapan, a no ser que se incluyan en *safe*. Además el valor de *safe* no es / por defecto.

Ejemplo: quote_plus('/El Niño/') produce '%2FEl+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'*)
Reemplaza secuencias de escape %xx con los caracteres individuales correspondientes. Los parámetros opcionales *encoding* y *errors* especifican cómo descodificar secuencias codificadas porcentualmente a caracteres Unicode, tal como lo acepta el método *bytes.decode()*.

string puede ser un objeto *str* o *bytes*.

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/'.

Distinto en la versión 3.9: El parámetro *string* admite bytes y cadenas de caracteres (anteriormente sólo cadenas de caracteres).

urllib.parse.**unquote_plus**(*string*, *encoding='utf-8'*, *errors='replace'*)
Como *unquote()*, pero también reemplaza los signos más por espacios, como es requerido al decodificar valores de formularios HTML.

string debe ser *str*.

Ejemplo: unquote_plus('/El+Ni%C3%B1o/') produce '/El Niño/'.

urllib.parse.**unquote_to_bytes**(*string*)
Reemplaza los escapes %xx por sus equivalentes de un solo octeto y retorna un objeto *bytes*.

string puede ser un objeto *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 porcentualmente. 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 valor 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 *urllib.parse.urlencode()* para generar una cadena de consulta para una dirección URL o datos para POST.

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 estándar actual Grupo de trabajo para el estándar URL que define URLs, dominio, dirección IP, formato *application/x-www-form-urlencoded* y su 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.

21.7 `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 autentificación.

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).

21.8 urllib.robotparser — Analizador para robots.txt

Código fuente: Lib/urllib/robotparser.py

Este módulo proporciona una clase única, *RobotFileParser*, que responde preguntas sobre si un agente de usuario en particular puede obtener una URL en el sitio web que publicó el archivo `robots.txt`. Para obtener más detalles sobre la estructura de los archivos `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
```

21.9 `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
>>> HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[<HTTPStatus.CONTINUE: 100>, <HTTPStatus.SWITCHING_PROTOCOLS:101>, ...]
```

21.9.1 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
103	EARLY_HINTS	Un código de estado HTTP para indicar pistas **RFC 8297**
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 (Experimer
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**, Section 4.1
305	USE_PROXY	HTTP/1.1 **RFC 7231**, Sección 6.4.5
307	TEMPORARY_REDIRECT	HTTP/1.1 **RFC 7231**, Section 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	Autentificación 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
418	IM_A_TEAPOT	HTCPCP/1.0 **RFC 2324**, Section 2.3.2
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
425	TOO_EARLY	Uso de datos iniciales en HTTP **RFC 8470**
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**

continué en la

Tabla 1 – proviene de la página anterior

Código	Nombre de la enumeración	Detalle
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 8
507	INSUFFICIENT_STORAGE	WebDAV **RFC 4918**, Sección 11.5
508	LOOP_DETECTED	Extensiones de unión WebDAV **RFC 5842**, Sección 7.2 (Experimenta
510	NOT_EXTENDED	Un framework de extensión HTTP **RFC 2774**, Sección 7 (Experimenta
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`.

Nuevo en la versión 3.9: Agregados códigos de estado `103 EARLY_HINTS`, `418 IM_A_TEAPOT` y `425 TOO_EARLY`.

21.10 `http.client` — Cliente de protocolo HTTP

Código fuente: Lib/http/client.py

This module defines classes that implement the client side of the HTTP and HTTPS protocols. It is normally not used directly — the module *urllib.request* uses it to handle URLs that use HTTP and 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*)

An *HTTPConnection* instance represents one transaction with an HTTP server. It should be instantiated by passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. If the optional *timeout* parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional *source_address* parameter may be a tuple of a (host, port) to use as the source address the HTTP connection is made from. The optional *blocksize* parameter sets the buffer size in bytes for sending a file-like message body.

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: The *strict* parameter was removed. HTTP 0.9-style «Simple Responses» are no longer supported.

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 *Consideraciones de seguridad* 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.

Distinto en la versión 3.10: Esta clase ahora envía una extensión ALPN con el indicador de protocolo http/1.1 cuando no se proporciona *context*. El *context* personalizado debe configurar los protocolos ALPN con set_alpn_protocol().

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.

21.10.1 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.

Lanza un *evento de auditoría* `http.client.connect` con los argumentos `self`, `host`, `port`.

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 de estilo **RFC 822**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()*.

Lanza un *evento de auditoría* `http.client.send` con los argumentos `self`, `data`.

21.10.2 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*)
Return the value of the header *name*, or *default* if there is no header matching *name*. If there is more than one header with the name *name*, return all of the values joined by ', '. If *default* is any iterable other than a single string, its elements are similarly returned joined by commas.

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.**url**
URL del recurso recuperado, comúnmente utilizado para determinar si se siguió una redirección.

HTTPResponse.**headers**
Cabeceras de la respuesta en forma de una instancia *email.message.EmailMessage*.

HTTPResponse.**status**
Código del estado retornado por el servidor.

HTTPResponse.**reason**
Una frase de la razón es retornada 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.

HTTPResponse.**geturl**()
Obsoleto desde la versión 3.9: Deprecada a favor de *url*.

HTTPResponse.**info**()
Obsoleto desde la versión 3.9: Deprecada a favor de *headers*.

HTTPResponse.**getcode**()
Obsoleto desde la versión 3.9: Deprecada a favor de *status*.

21.10.3 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
```

Here is an example session that uses the `POST` method:

```
>>> 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="https://bugs.python.org/issue12524">https://bugs.python.
↪org/issue12524</a>'
>>> conn.close()
```

Client side HTTP `PUT` requests are very similar to `POST` requests. The difference lies only on the server side where HTTP servers will allow resources to be created via `PUT` requests. It should be noted that custom HTTP methods

are also handled in *urllib.request.Request* by setting the appropriate method attribute. Here is an example session that uses the PUT method:

```
>>> # This creates an HTTP request
>>> # 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
```

21.10.4 Objetos de HTTPMessage

Una instancia de http.client.HTTPMessage contiene los encabezados de una respuesta HTTP. Se implementa utilizando la clase *email.message.Message*.

21.11 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. Puede usar esto para escribir programas de Python que realicen una variedad de trabajos FTP automatizados, como duplicar otros servidores FTP. También es utilizado por el módulo *urllib.request* para manejar URL que usan FTP. Para más información sobre FTP (File Transfer Protocol), ver internet :rfc:”959”.

La codificación predeterminada es UTF-8, siguiendo **RFC 2640**.

Aquí hay una sesión de ejemplo usando el módulo *ftplib*:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.us.debian.org')   # connect to host, default port
>>> ftp.login()                      # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')                # change into "debian" directory
'250 Directory successfully changed.'
>>> 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()
'221 Goodbye.'
```

El módulo define los siguientes elementos:

class ftplib.**FTP**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *timeout=None*, *source_address=None*, ***, *encoding='utf-8'*)

Retorna una nueva instancia de la clase *FTP*. Cuando se da *host*, se realiza la llamada al método connect(host). Cuando se da *user*, además se realiza la llamada al método login(user, passwd,

acct) (donde *passwd* y *acct* predeterminan la cadena de caracteres vacía cuando no se dan). El parámetro opcional *timeout* especifica un tiempo de espera en segundos para bloquear operaciones como el intento de conexión (si no se especifica, se utilizará la configuración de tiempo de espera global predeterminada). *source_address* es una tupla de dos elementos (host, port) para que el socket se vincule como su dirección de origen antes de conectarse. El parámetro *encoding* especifica la codificación de directorios y nombres de archivo.

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*.

Distinto en la versión 3.9: Si el parámetro *timeout* se establece en cero, lanzará un *ValueError* para evitar la creación de un socket sin bloqueo. Se agregó el parámetro *encoding*, y el valor predeterminado se cambió de Latin-1 a UTF-8 para seguir **RFC 2640**.

class ftplib.**FTP_TLS**(*host=", user=", passwd=", acct=", keyfile=None, certfile=None, context=None, timeout=None, source_address=None, *, encoding='utf-8'*)

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 *Consideraciones de seguridad* 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.

Distinto en la versión 3.9: Si el parámetro *timeout* se establece en cero, lanzará un *ValueError* para evitar la creación de un socket sin bloqueo. Se agregó el parámetro *encoding*, y el valor predeterminado se cambió de Latin-1 a UTF-8 para seguir **RFC 2640**.

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',
↪'phpbuddy', 'phpbench', 'pinscaster', 'ping', 'posto', 'pub', 'public',
↪'public_keys', 'pure-ftpd', 'qscan', 'qtc', 'sharedance', 'skycache', 'sound
↪', 'tmp', 'ucarp']
```

(continúe en la próxima página)

(proviene de la página anterior)

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.

21.11.1 Objetos FTP

Hay varios métodos disponibles en dos versiones: uno para manejar archivos de texto y otro para archivos binarios. Estos reciben el nombre del comando que se utiliza seguido de `lines` para la versión de texto o `binary` para la versión binaria.

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* is 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 la codificación especificada por el parámetro *encoding* en la inicialización. *cmd* debe ser un comando `RETR` apropiado (ver *retrbinary()*) o un comando como `LIST` o `NLST` (generalmente solo la cadena de caracteres `'LIST'`). `LIST` recupera una lista de archivos e información sobre esos archivos. `NLST` recupera una lista de nombres de archivos. La función *callback* se llama para 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 modo de línea. *cmd* debe ser un comando `STOR` apropiado (ver *storbinary()*). Las líneas se leen hasta EOF del *file object* *fp* (abierto en modo binario) usando su método *readline()* para proporcionar los datos que se almacenarán. *callback* es un parámetro único opcional que se puede llamar en cada línea después de su envío.

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 se proporciona *rest* opcional, se envía un comando `REST` al servidor, pasando *rest* como argumento. *rest* suele ser un desplazamiento de bytes en el archivo solicitado, que le indica al servidor que reinicie el envío de los bytes del archivo en el desplazamiento solicitado, omitiendo los bytes iniciales. Sin embargo, tenga en cuenta que el método `transfercmd()` convierte *rest* en una cadena de caracteres con el parámetro *encoding* especificado en la inicialización, pero no se realiza ninguna comprobación del contenido de la cadena de caracteres. Si el servidor no reconoce el comando `REST`, se lanzará una excepción `error_reply`. Si esto sucede, simplemente llame a `transfercmd()` sin un argumento *rest*.

`FTP.`**`ntransfercmd`**(*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=[]*)
Enumera un directorio en un formato estandarizado usando el comando `MLSD` (**RFC 3659**). Si se omite *path*, se asume el directorio actual. *facts* es una lista de cadenas de caracteres que representan el tipo de información deseada (por ejemplo, `["type", "size", "perm"]`). Retorna un objeto generador que produce una tupla de dos elementos por cada archivo encontrado en la ruta. El primer elemento es el nombre del archivo, el segundo es un diccionario que contiene datos sobre el nombre del archivo. El contenido de este diccionario puede estar limitado por el argumento *facts*, pero no se garantiza que el servidor retorne todos los datos solicitados.

Nuevo en la versión 3.3.

`FTP.`**`nlst`**(*argument*[, ...])
Retorna una lista de nombres de archivos tal como los retorna el comando `NLST`. El *argument* opcional es un directorio para listar (el predeterminado es el directorio del servidor actual). Se pueden usar varios argumentos para pasar opciones no estándar al comando `NLST`.

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()`*.

21.11.2 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`**`()`
Revierte 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.

21.12 `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`.

Distinto en la versión 3.9: Si el parámetro *timeout* se establece en cero, lanzará un *ValueError* para evitar la creación de un socket sin bloqueo.

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 *Consideraciones de seguridad* 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.

Distinto en la versión 3.9: Si el parámetro *timeout* se establece en cero, lanzará un *ValueError* para evitar la creación de un socket sin bloqueo.

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.

21.12.1 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 mesgnum uid`, de otra forma el resultado es una lista `(respuesta, ['mengnum 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 *Consideraciones de seguridad* 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.

21.12.2 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.

21.13 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*, *timeout=None*)
Esta clase implementa el protocolo 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). El parámetro opcional *timeout* especifica un timeout en segundos para intentar realizar la conexión. Si timeout no se especifica, o es None, se usa el timeout global por defecto de sockets.

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`.

Distinto en la versión 3.9: El parámetro opcional *timeout* fue agregado.

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*, *timeout=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 *Consideraciones de seguridad* 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*.

El parámetro opcional *timeout* especifica un timeout en segundos para intentar realizar la conexión. Si timeout no se especifica, o es None, se usa el timeout global por defecto de sockets.

Distinto en la versión 3.3: El parámetro *ssl_context* fue 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.

Distinto en la versión 3.9: El parámetro opcional *timeout* fue agregado.

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*)
Convierte un número entero en una representación de bytes utilizando caracteres del conjunto [`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:

Documentos describiendo el protocolo, y fuentes de servidores que lo implementan, del Centro de Información IMAP de la Universidad de Washington, pueden ser encontrados en (**Código Fuente**) https://github.com/uw-imap/imap (**Fuera de mantención**).

21.13.1 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 AUTENTICATE 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)'`).

Cada comando retorna una tupla: `(type, [data, ...])` donde *type* suele ser `'OK'` o `'NO'`, y *data* es el texto de la respuesta del comando o resultados obligatorios del comando. Cada *data* es un objeto `bytes` o una tupla. Si es una tupla, la primera parte es el encabezado de la respuesta, y la segunda parte contiene los datos (es decir, el valor "literal").

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 quota 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*, *timeout=None*)
Abre un socket a *port* y *host*. El parámetro opcional *timeout* especifica un timeout en segundos para intentar realizar la conexión. Si timeout no se especifica, o es None, se usa el timeout global por defecto de sockets. Nota también que si el parámetro *timeout* es cero se lanzará un *ValueError* para evitar crear un socket sin bloqueo. Este método es implícitamente llamado por el constructor *IMAP4*. Los objetos de conexiones establecidas por este método serán usados en los métodos *IMAP4.read()*, *IMAP4.readline()*, *IMAP4.send()*, y *IMAP4.shutdown()*. Este método se puede sobreescribir.

Genera un *evento de auditoría* `imaplib.open` con argumentos `self`, `host`, `port`.

Distinto en la versión 3.9: El parámetro *timeout* fue agregado.

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*)
Seleccione 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 *Consideraciones de seguridad* 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.**unselect**()
imaplib.IMAP4.unselect() libera recursos del servidor asociados al buzón de correo seleccionado y devuelve el servidor al estado autenticado. Este comando realiza las mismas acciones que *imaplib.IMAP4.close()*, con la excepción de que ningún mensaje es permanentemente borrado del buzón de correo actualmente seleccionado.

Nuevo en la versión 3.9.

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.

21.13.2 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()
```

21.14 `smtplib` — Cliente de protocolo SMTP

Código fuente: Lib/smtplib.py

El módulo *smtplib* define un objeto de sesión de cliente SMTP que se puede utilizar para enviar correo a cualquier máquina de Internet con un demonio de escucha SMTP o ESMTP. Para obtener detalles sobre el funcionamiento de SMTP y ESMTP, consulte **RFC 821** (Protocolo simple de transferencia de correo) 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 de puerto y host opcionales, se llama al método SMTP *connect()* con esos parámetros durante la inicialización. Si se especifica, *local_hostname* se utiliza como FQDN del host local en el comando HELO / EHLO. De lo contrario, el nombre de host local se encuentra mediante *socket.getfqdn()*. Si la llamada *connect()* retorna algo que no sea un código de éxito, se lanza un *SMTPConnectError*. El parámetro opcional *timeout* especifica un tiempo de espera en segundos para bloquear operaciones como el intento de conexión (si no se especifica, se utilizará la configuración de tiempo de espera global predeterminada). Si expira el tiempo de espera, se lanza *TimeoutError*. El parámetro opcional source_address permite la vinculación a alguna dirección de origen específica en una máquina con múltiples interfaces de red y/o algún puerto TCP de origen específico. Se necesita una tupla de 2 (host, puerto), para que el socket se vincule como su dirección de origen antes de conectarse. Si se omite (o si el host o el puerto son `''` y / o 0 respectivamente), se utilizará el comportamiento predeterminado del sistema operativo.

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 agrego el argumento source_address.

Nuevo en la versión 3.5: La extensión SMTPUTF8 (**RFC 6531**) ahora es compatible.

Distinto en la versión 3.9: Si el parámetro *timeout* se mantiene en cero, lanzará un `ValueError` para evitar la creación de un socket no bloqueante

class smtplib.**SMTP_SSL**(*host=''*, *port=0*, *local_hostname=None*, *keyfile=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 *Consideraciones de seguridad* 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 agrego 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.

Distinto en la versión 3.9: Si el parámetro *timeout* se mantiene en cero, lanzará un `ValueError` para evitar la creación de un socket no bloqueante

class smtplib.**LMTP**(*host=''*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*[, *timeout*])

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.

Distinto en la versión 3.9: Se ha añadido el parámetro opcional *timeout*.

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.

21.14.1 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* `smtplib.connect` con argumentos `self`, `host`, `port`.

SMTP.**helo**(*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 helo_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 mediante 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 utilice *has_extn()*. También establece varios atributos informativos: el mensaje retornado por el servidor se almacena como el atributo ehlo_resp, does_esmtp se establece en True o False dependiendo de si el servidor admite ESMTP, y esmtp_features será un diccionario que contiene los nombres de las extensiones de servicio SMTP de este servidor soportes, 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 *smtplib* 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 esmtp_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 deberías 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, lanzará 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.

21.14.2 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*.

21.15 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
El UUID fue generado por la plataforma de una manera segura para multiprocesamiento.

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 `UID.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
`time_low`	los primeros 32 bits del UUID
`time_mid`	los siguientes 16 bits del UUID
`time_hi_version`	los siguientes 16 bits del UUID
`clock_seq_hi_variant`	los siguientes 8 bits del UUID
`clock_seq_low`	los siguientes 8 bits del UUID
`node`	los siguientes 48 bits del UUID
`time`	el timestamp de 60-bit
`clock_seq`	el número de secuencia de 14-bit

`UUID.`**`hex`**
The UUID as a 32-character lowercase hexadecimal string.

`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 multiprocesamiento.

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 `uuuid` define los siguientes identificadores de espacios de nombres para su uso con *uuid3()* o `uuuid5()`.

`uuid.`**NAMESPACE_DNS**
When this namespace is specified, the *name* string is a fully qualified domain name.

`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 - Un espacio de nombres URN de identificador único universal (UUID) 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.

21.15.1 Ejemplo

Aquí hay algunos ejemplos del uso típico del modulo *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
>>> 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'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> # 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')
```

21.16 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*)
Esto 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 intenta invocar automáticamente *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 asincrónico.

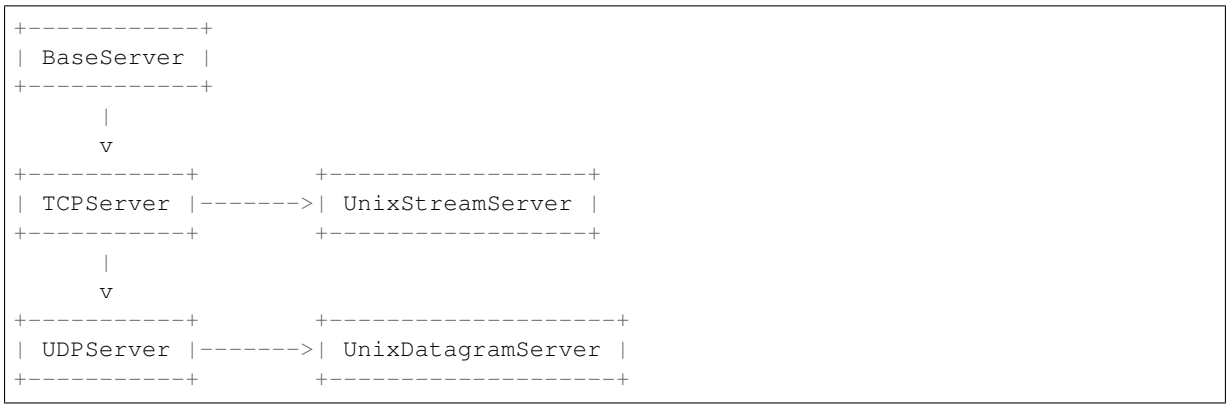
La creación de un servidor requiere varios pasos. Primero, debes crear una clase de controlador de solicitudes subclasificando la clase *BaseRequestHandler* y anulando su método *handle()*; este método procesará las solicitudes entrantes. En segundo lugar, debe crear una instancia de una de las clases 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 subprocesos, debe declarar explícitamente cómo desea que se comporten sus subprocesos 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 subprocesos 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.

21.16.1 Notas de creación del servidor

Hay cinco clases en un diagrama de herencia, cuatro de las cuales representan servidores síncronos de cuatro tipos:



Note that *UnixDatagramServer* derives from *UDPServer*, not from *UnixStreamServer* — the only difference between an IP and a Unix server is the address family.

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 subprocesos que no son demonios, excepto si el atributo `socketserver.ThreadingMixIn.block_on_close` es falso. Use subprocesos demoníacos configurando `ThreadingMixIn.daemon_threads` en `Verdadero` para no esperar hasta que se completen los subprocesos.

Distinto en la versión 3.7: `socketserver.ForkingMixIn.server_close()` y `socketserver.ThreadingMixIn.server_close()` ahora espera hasta que se completen todos los procesos secundarios 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 explicito 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.

21.16.2 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 da 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()*.

21.16.3 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**
These *BaseRequestHandler* subclasses override the *setup()* and *finish()* methods, and provide `self.rfile` and `self.wfile` attributes. The `self.rfile` and `self.wfile` attributes can be read or written, respectively, to get the request data or return data to the client. The `rfile` attributes support the *io.BufferedIOBase* readable interface, and `wfile` attributes support the `io.BufferedIOBase` writable interface.

Distinto en la versión 3.6: `StreamRequestHandler.wfile` también admite la interfaz de escritura *io.BufferedIOBase*.

21.16.4 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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

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()
        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()*.

21.17 `http.server` — Servidores HTTP

Código fuente: Lib/http/server.py

Este módulo define clases para implementar servidores HTTP.

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
Contiene la ruta de la solicitud. Si el componente de consulta de la URL está presente, `path` incluye la consulta. Usando la terminología de: rfc: *3986*, `path` aquí incluye `hier-part` y `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.BufferedIOBase*, 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.BufferedIOBase*.

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, implemente 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 `100 Continue` seguido de encabezados `200 OK`. Este método se puede anular para generar un error si el servidor no desea que el cliente continúe. Por ejemplo, el servidor puede optar por 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 archivos del directorio *directory* e inferior, o del directorio actual si no se proporciona *directory*, mapeando directamente la estructura del directorio a las solicitudes HTTP.

Nuevo en la versión 3.7: El parámetro *directory*.

Distinto en la versión 3.9: El parámetro *directory* acepta un *path-like object*.

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 asigna sufijos a tipos MIME contiene sobreescrituras personalizadas para las asignaciones predeterminadas del sistema. El mapeo se usa sin distinción entre mayúsculas y minúsculas, por lo que solo debe contener claves en minúsculas.

Distinto en la versión 3.9: Este diccionario ya no contiene las asignaciones predeterminadas del sistema, sino que solo contiene anulaciones.

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.

For example usage, see the implementation of the `test` function in Lib/http/server.py.

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 can also be invoked directly using the `-m` switch of the interpreter. Similar to the previous example, this serves files relative to the current directory:

```
python -m http.server
```

The server listens to port 8000 by default. The default can be overridden by passing the desired port number as an argument:

```
python -m http.server 9000
```

By default, the server binds itself to all interfaces. The option `-b/--bind` specifies a specific address to which it should bind. Both IPv4 and IPv6 addresses are supported. For example, the following command causes the server to bind to localhost only:

```
python -m http.server --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

By default, the server uses the current directory. The option `-d/--directory` specifies a directory to which it should serve the files. For example, the following command uses a specific directory:

```
python -m http.server --directory /tmp/
```

Nuevo en la versión 3.7: `--directory` argument was introduced.

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
```

21.17.1 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 3.10.9: Control characters are scrubbed in stderr logs.

21.18 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 la invalidez de **RFC 2109**: 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.

21.18.1 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, analícela como un `HTTP_COOKIE` y agregue los valores que se encuentran allí como *Morsel*s. Si es un diccionario, equivale a:

```
for k, v in rawdata.items():
    cookie[k] = v
```

21.18.2 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()*.

21.18.3 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
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()
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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
```

21.19 `http.cookiejar` — Manejo de cookies para clientes HTTP

Código fuente: Lib/http/cookiejar.py

El módulo *`http.cookiejar`* define clases para el manejo automático de cookies HTTP. Es útil para acceder a sitios web que requieren pequeñas piezas de datos – *cookies* – para establecerse en el equipo del cliente a través de una respuesta HTTP de un servidor web, y que además retornan al servidor más tarde en forma de requests de HTTP.

Se manejan tanto el protocolo de cookies normal de Netscape como el protocolo definido por **RFC 2965**. El manejo de RFC 2965 está desactivado de forma predeterminada. Las cookies **RFC 2109** se analizan como cookies de Netscape y, posteriormente, se tratan como cookies de Netscape o RFC 2965 de acuerdo con la «política» en vigor. Tenga en cuenta que la gran mayoría de las cookies en Internet son cookies de Netscape. *`http.cookiejar`* intenta seguir el protocolo de cookies de Netscape de facto (que difiere sustancialmente del establecido en la especificación original de Netscape), incluida la toma de nota de los atributos de cookies `max-age` y `port` introducidos con RFC 2965.

Nota: Diversos parámetros nombrados que se encuentran en *`Set-Cookie`* y *`Set-Cookie2`* (ejem. `domain` y `expires`) son convencionalmente referidos como *attributes*. Para distinguirlos de los atributos de Python, la documentación de este módulo utiliza el término *cookie-attribute*.

El módulo define la siguiente excepción:

exception http.cookiejar.**LoadError**

Las instancias de *`FileCookieJar`* provocan esta excepción al no cargar las cookies desde un archivo. *`LoadError`* es una subclase de *`OSError`*.

Distinto en la versión 3.3: LoadError fue transformado a una subclase de *`OSError`* en vez de *`IOError`*.

Se proporcionan las siguientes clases:

class http.cookiejar.**CookieJar**(*policy=None*)

policy es un objeto implementando la interfaz *`CookiePolicy`*.

La clase *`CookieJar`* almacena HTTP cookies. Ésta extrae cookies de los request de HTTP, y las retorna en forma de responses de HTTP. Las instancias de *`CookieJar`* automáticamente expiran las cookies contenidas

cuando es necesario. Las subclases también son responsables de almacenar y recuperar cookies de un archivo o base de datos.

class http.cookiejar.**FileCookieJar**(*filename=None*, *delayload=None*, *policy=None*)
policy es un objeto implementando la interfaz `CookiePolicy`. Para los otros argumentos, te recomendamos visitar la documentación correspondiente a los atributos.

Una `CookieJar` puede cargar cookies de, y posiblemente almacenar, un archivo en el disco. Las cookies **NO** son cargadas desde un archivo nombrado hasta que alguno de los métodos `load()` o `revert()` es llamado. Subclases de esta clase son documentadas en la sección *Subclases FileCookieJar y co-operación con navegadores web*.

This should not be initialized directly – use its subclasses below instead.

Distinto en la versión 3.8: El parámetro filename soporta un *path-like object*.

class http.cookiejar.**CookiePolicy**
Esta clase es responsable de decidir si cada cookie debe ser aceptada del servidor, o retornada al mismo.

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')*)
Los argumentos del constructor sólo deben ser dados como argumentos clave. *blocked_domains* es una secuencia de nombres de dominio cuyas cookies nunca deben de ser aceptadas o retornadas. *allowed_domains* y no `None`, es una secuencia de sólo dominios que podemos aceptar y retornar cookies. *secure_protocols* es una secuencia de protocolos para los que se pueden agregar cookies seguras. De manera predeterminada *https* y *wss* (websocket seguro) son considerados protocolos seguros. Para todos los demás argumentos, te recomendamos visitar la documentación para los objetos `CookiePolicy` y `DefaultCookiePolicy`.

`DefaultCookiePolicy` implementa el estándar de reglas aceptar/rechazar para cookies Netscape y **RFC 2965**. De manera predeterminada, las cookies **RFC 2109** (es decir, las cookies que reciben una cabecera `Set-Cookie` con una versión cookie-attribute de 1) son tratadas de acuerdo con las reglas del RFC 2965. Sin embargo, si el manejo del RFC2965 se encuentra apagado o `rfc2109_as_netscape` es `True`, las cookies RFC 2109 son "degradadas" por la instancia `CookieJar` a las cookies Netscape, configurando la `version` del atributo de la instancia `Cookie` como 0. `DefaultCookiePolicy` también provee algunos parámetros que permiten fine-tunning en el policy.

class http.cookiejar.**Cookie**
Esta clase representa Netscape, las cookies **RFC 2109** y **RFC 2965**. No se espera que los usuarios de `http.cookiejar` construyan sus propias instancias de `Cookie`. En vez de ello, si es necesario, llama al método `make_cookies()` de la instancia `CookieJar`.

Ver también:

Módulo `urllib.request` Apertura de URL con manejo automático de cookies.

Módulo `http.cookies` Clases de cookies HTTP, principalmente útiles para código server-side. Los módulos `http.cookiejar` y `http.cookies` no dependen uno del otro.

https://curl.se/rfc/cookie_spec.html La especificación original del protocolo de cookies Netscape. Aunque éste sigue siendo el protocolo dominante, el "protocolo de cookies Netscape" implementado por la mayoría de los navegadores (y el protocolo `http.cookiejar`) sólo tienen in parecido pasajero con el bosquejado en `cookie_spec.html`.

RFC 2109 - Mecanismo de Gestión de Estados HTTP Obsoleto por **RFC 2965**. Usa `Set-Cookie` con versión=1.

RFC 2965 - Mecanismo de Gestión de Estados HTTP El protocolo Netscape con los errores solucionados. Usa *Set-Cookie2* en lugar de *Set-Cookie*. No es ampliamente usado.

http://kristol.org/cookie/errata.html Fe de erratas sin finalizar hasta el **RFC 2965**.

RFC 2964 - Mecanismo de Gestión de Estados HTTP

21.19.1 Objetos CookieJar y FileCookieJar

Los objetos *CookieJar* soportan el protocolo *iterator* para iterar sobre los objetos *Cookie* contenidos.

CookieJar tiene los siguientes métodos:

CookieJar.**add_cookie_header**(*request*)
Añade la cookie correcta a la cabecera del request.

Si el *policy* admite (ie. los atributos `rfc2965` y `hide_cookie2` de la instancia *CookieJar* del *CookiePolicy* son de manera respectiva verdadero y falso), la cabecera *Cookie2* es igualmente añadido cuando es apropiado.

The *request* object (usually a *urllib.request.Request* instance) must support the methods `get_full_url()`, `has_header()`, `get_header()`, `header_items()`, `add_unredirected_header()` and the attributes `host`, `type`, `unverifiable` and `origin_req_host` as documented by *urllib.request*.

Distinto en la versión 3.3: el objeto *request* necesita del atributo `origin_req_host`. La dependencia en el método deprecado `get_origin_req_host()` ha sido removida.

CookieJar.**extract_cookies**(*response*, *request*)
Extrae las cookies del *response* HTTP y las almacena en el *CookieJar*, donde está permitido por la policy.

El *CookieJar* buscará por cabeceras *Set-Cookie* y *Set-Cookie2* permitidos en el argumento *response*, y almacena las cookies cuando es apropiado (sujeto a la aprobación del método *CookiePolicy.set_ok()*).

El objeto *response* (usualmente el resultado de llamar *urllib.request.urlopen()*, o similares) debe de soportar un método `info()` , el cual retorna una instancia *email.message.Message*.

The *request* object (usually a *urllib.request.Request* instance) must support the method `get_full_url()` and the attributes `host`, `unverifiable` and `origin_req_host`, 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: el objeto *request* necesita del atributo `origin_req_host`. La dependencia en el método deprecado `get_origin_req_host()` ha sido removida.

CookieJar.**set_policy**(*policy*)
Establece la instancia *CookiePolicy* para ser usada.

CookieJar.**make_cookies**(*response*, *request*)
Retorna una secuencia de objetos *Cookie* extraída del objeto *response*.

Revisa la documentación *extract_cookies()* para conocer las interfaces necesarias de los argumentos *response* y *request*.

CookieJar.**set_cookie_if_ok**(*cookie*, *request*)
Establece una *Cookie* si la política (*policy*) dice OK para hacerlo.

CookieJar.**set_cookie**(*cookie*)
Establece una *Cookie*, sin consultar con la política (*policy*) para ver si se debe de establecer o no.

CookieJar.**clear**([*domain*[, *path*[, *name*]]])
Borra algunas cookies.

Si se invoca sin argumentos, borra todas las cookies. Si se le da un solo argumento. sólo las cookies que pertenecen a tal *domain* serán removidas. Si se le da dos argumentos, las cookies pertenecientes al especificado

domain y *path* URL serán removidas. Si se le da tres argumentos, entonces la cookie con el especificado *domain*, *path* y *name* será removida.

Lanza *KeyError* si no existe ninguna cookie que coincida.

CookieJar.**clear_session_cookies**()
Descarta todas las cookies de la sesión.

Descarta todas las cookies que tengan un atributo `discard` true (usualmente porque no tienen `max-age` o `expires` atributos'cookie, o un explícito atributo-cookie `discard`). Para los navegadores interactivos, el fin de la sesión usualmente corresponde con el cierre de la ventana del navegador.

Nota que el método `save()` no guardará las cookies de la sesión de todas maneras, a menos que de otra manera preguntes para pasar un argumento *ignore_discard* como true.

FileCookieJar implementa los siguientes métodos adicionales:

FileCookieJar.**save**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Guarda las cookies en un archivo.

Esta clase base genera *NotImplementedError*. Las subclases podrían dejar este método sin implementar.

filename es el nombre del archivo en el que se guardarán las cookies. Si *filename* no es especificado, `self.filename` es utilizado (cuyo valor predeterminado es el valor que será pasado al constructor, si hay alguno); si `self.filename` es *None*, se genera un *ValueError*.

ignore_discard: almacena incluso las cookies configuradas para ser descartadas. *ignore_expires*: almacena las cookies que han caducado

El archivo se sobrescribe si ya existe, borrando así todas las cookies que contiene. Las cookies guardadas se pueden restaurar después utilizando los métodos *load()* o *revert()*.

FileCookieJar.**load**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Carga las cookies desde un archivo.

Las viejas cookies son guardadas a menos que sean sobre escritas por nuevas recién cargadas.

Los argumentos son en cuanto a *save()*.

El archivo nombrado debe estar en un formato entendible para la clase, o se lanzará un *LoadError*. Además, puede generarse un *OSError*, por ejemplo, si el archivo no existe.

Distinto en la versión 3.3: Solía levantarse un *IOError*, pero ahora es un alias de *OSError*.

FileCookieJar.**revert**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Limpia todas las cookies y recarga las cookies de un archivo guardado.

revert() puede levantar las mismas excepciones que *load()*. Si hay un fallo. el estado del objeto no se alterará.

Instancias *FileCookieJar* tienen los siguientes atributos públicos:

FileCookieJar.**filename**
Nombre del archivo predeterminado para el archivo en donde se almacenan las cookies. Este atributo podría ser asignado.

FileCookieJar.**delayload**
Si es true, carga perezosamente las cookies desde el disco. Este atributo no se debería de asignar. Esta es solo una pista, debido a que solo el rendimiento, y no al comportamiento (a menos que las cookies en el disco sean cambiadas). Un objeto *CookieJar* puede ignorarlo. Ninguna de las clases de *FileCookieJar* incluidas en la librería estándar carga las cookies de manera perezosa.

21.19.2 Subclases FileCookieJar y co-operación con navegadores web

Las siguientes subclases *CookieJar* son proveídas para lectura y escritura.

class http.cookiejar.**MozillaCookieJar**(*filename=None*, *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 curl and the Lynx and Netscape browsers).

Nota: Esto puede perder información acerca de cookies **RFC 2965**, y también sobre atributos nuevos o no estándar como `port`.

Advertencia: Realiza una copia de seguridad de tus cookies antes de guardarlas, especialmente si tienes cookies cuya perdida / corrupción puede ser inconveniente (hay algunas sutilezas que pueden conducir a ligeros cambios en el archivo sobre una carga / guardado round-trip).

También toma en cuenta que las cookies guardadas mientras Mozilla se está ejecutando se volverán torpes debido a Mozilla.

class http.cookiejar.**LWPCookieJar**(*filename=None*, *delayload=None*, *policy=None*)
Un *FileCookieJar* que puede cargar y guardar en disco en formato compatible con el formato de archivo `Set-Cookie3` de la librería libwww-perl. Estos es conveniente si tu quieres guardar cookies en un archivo legible para los humanos.

Distinto en la versión 3.8: El parámetro filename soporta un *path-like object*.

21.19.3 Objetos CookiePolicy

Objetos implementando la interfaz *CookiePolicy* tienen los siguientes métodos:

CookiePolicy.**set_ok**(*cookie*, *request*)
Retorna un valor booleano indicando si la cookie debe ser aceptada o no desde el servidor.

cookie es una instancia de *Cookie*. *request* es un objeto implementando la interfaz definida por la documentación de *CookieJar.extract_cookies()*.

CookiePolicy.**return_ok**(*cookie*, *request*)
Retorna un valor booleano indicando si la cookie debe ser retornada o no al servidor.

cookie es una instancia de *Cookie*. *request* es un objeto implementando la interfaz definida por la documentación de *CookieJar.add_cookie_header()*.

CookiePolicy.**domain_return_ok**(*domain*, *request*)
Retorna `False` si las cookies no deben de ser retornadas, a partir del dominio cookie dado.

Este método es una optimización. Elimina la necesidad de verificar cada cookie con un domino en particular (lo cual puede involucrar la lectura de muchos archivos). Retornando true de *domain_return_ok()* y *path_return_ok()* deja todo el trabajo a *return_ok()*.

Si *domain_return_ok()* retorna true para el dominio de cookies, *path_return_ok()* es llamado para el path de cookies. De otra manera, *path_return_ok()* y *return_ok()* no son nunca llamados para ese dominio de cookies. Si *path_return_ok()* retorna true, *return_ok()* es llamado con el objeto *Cookie* en sí para una revisión completa. De otra manera, *return_ok()* es nunca llamado para ese path de cookies.

Ten en cuenta que *domain_return_ok()* es llamado para cada dominio *cookie*, no solamente para el dominio *request*. Por ejemplo, la función podría ser llamada con ambos `".example.com"` y `"www.example.com"` si el dominio request es `"www.example.com"`. Lo mismo se aplica para *path_return_ok()*.

El argumento *request* es tal y como es documentado para *return_ok()*.

CookiePolicy.**path_return_ok**(*path*, *request*)
Retorna `False` si las cookies no deben de ser retornadas, dado el path de las cookies.

Revisa la documentación para *domain_return_ok()*.

Adicionalmente a los métodos implementados anteriormente, las implementaciones de la interfaz *CookiePolicy* también deben de proporcionar los siguientes atributos, indicando cuales protocolos deben de ser utilizados, y como. Todos estos atributos se pueden asignar.

CookiePolicy.**netscape**
Implementa el protocolo Netscape.

CookiePolicy.**rfc2965**
Implementa el protocolo **RFC 2965**.

CookiePolicy.**hide_cookie2**
No añadas la cabecera *Cookie2* a las requests (la presencia de esta cabecera indica al servidor que nosotros entendemos las cookies **RFC 2965**).

El camino más útil para definir una clase *CookiePolicy* es haciendo una subclase de *DefaultCookiePolicy* y sobre escribir algunos o todos los métodos anteriores. *CookiePolicy* en sí misma puede ser utilizada como una "null policy" para permitir establecer y recibir algunas o todas las cookies (esto raramente puede ser útil).

21.19.4 Objetos DefaultCookiePolicy

Implementa las reglas estándar para aceptar y retornar cookies.

Ambas cookies, **RFC 2965** y Netscape son cubiertas. El manejo del RFC 2965 está desactivado de forma predeterminada.

La forma más sencilla de proporcionar tu propia política (*policy*) es sobre escribir esta clase y llamar sus métodos en tus implementaciones modificadas antes de añadir tu propias comprobaciones adicionales:

```
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
```

En adición a las características requeridas para implementar la interfaz *CookiePolicy*, esta clase te permite que bloquees o permitas dominios de establecer y recibir cookies. También hay algunos interruptores estrictos que te permiten ajustar un poco las reglas flojas del protocolo Netscape (a costa de bloquear algunas cookies benignas).

Se proporciona una lista de bloqueo y una lista de permitidos del dominio (ambos desactivados de forma predeterminada). Solo los dominios que no están en la lista de bloqueo y que están presentes en la lista de permisos (si la lista de permisos está activa) participan en la configuración y devolución de cookies. Utilice el argumento del constructor *blocked_domains* y los métodos `blocked_domains()` y `set_blocked_domains()` (y el argumento y los métodos correspondientes para *allowed_domains*). Si configura una lista de permitidos, puede desactivarla nuevamente configurándola en *None*.

Los dominios en listas de bloqueos o permitidos que no comienzan con un punto deben ser iguales al dominio de la cookie para que coincidan. Por ejemplo, `"example.com"` coincide con una entrada de lista de bloqueo de `"example.com"`, pero `"www.example.com"` no. Los dominios que comienzan con un punto también se corresponden con dominios más específicos. Por ejemplo, tanto `"www.example.com"` como `"www.coyote.example.com"` coinciden con `".example.com"` (pero el propio `"example.com"` no lo hace). Las direcciones IP son una excepción y deben coincidir exactamente. Por ejemplo, si block_domains contiene `"192.168.1.2"` y `".168.1.2"`, 192.168.1.2 está bloqueado, pero 193.168.1.2 no.

DefaultCookiePolicy implementa los siguientes métodos adicionales:

DefaultCookiePolicy.**blocked_domains**()
Retorna una secuencia de dominios bloqueados (en forma de tupla).

DefaultCookiePolicy.**set_blocked_domains**(*blocked_domains*)
Establece la secuencia de dominios bloqueados.

DefaultCookiePolicy.**is_blocked**(*domain*)
Retorna si *domain* está en la lista de bloqueo para configurar o recibir cookies.

DefaultCookiePolicy.**allowed_domains**()
Retorna *None*, o la secuencia de los dominios permitidos (en forma de tupla).

DefaultCookiePolicy.**set_allowed_domains**(*allowed_domains*)
Establece la secuencia de los dominios permitidos, o *None*.

DefaultCookiePolicy.**is_not_allowed**(*domain*)
Retorna si *domain* no está en la lista de permitidos para configurar o recibir cookies.

Las instancias *DefaultCookiePolicy* tienen los siguientes atributos, que son todos inicializados desde los argumentos del constructor del mismo nombre, y el cual todos pueden ser asignados a.

DefaultCookiePolicy.**rfc2109_as_netscape**
Si es true, solicita que la instancia *CookieJar* degrade las cookies **RFC 2109** (es decir, las cookies recibidas en una cabecera *Set-Cookie* con un cookie-attribute versión de 1) a cookies Netscape al establecer la versión del atributo de la instancia *Cookie* a 0. El valor predeterminado es *None*, en cuyo caso las cookies RFC 2109 son degradadas si y sólo si el control de **RFC 2965** se encuentra apagado. Por lo tanto, las cookies RFC 2109 son degradadas de forma predeterminada.

Interruptores generales de rigurosidad:

DefaultCookiePolicy.**strict_domain**
No permite que los sitios establezcan dominios de dos componentes con dominios country-code top-level como `.co.uk`, `.gov.uk`, `.co.nz`.etc. ¡Esto está lejos de ser perfecto y no está garantizado a que vaya a funcionar!

Interruptores de rigurosidad del protocolo **RFC 2965**:

DefaultCookiePolicy.**strict_rfc2965_unverifiable**
Siga las reglas :rfc:”2965” sobre transacciones no verificables (normalmente, una transacción no verificable es una del resultado de una redirección o una solicitud de una imagen alojada en otro sitio). Si esto es falso, las cookies *nunca* se bloquean en base a la verificabilidad

Interruptores de rigurosidad del protocolo Netscape:

DefaultCookiePolicy.**strict_ns_unverifiable**
Aplica las reglas del **RFC 2965** a transacciones no verificables incluso a cookies Netscape.

DefaultCookiePolicy.**strict_ns_domain**
Indicadores que señalan que tan estricto deben ser con las reglas de domain-matching para cookies Netscape. Consulte a continuación los valores aceptables.

DefaultCookiePolicy.**strict_ns_set_initial_dollar**
Ignora las cookies en las cabeceras Set-Cookie: que tengan nombres que comienzan con `'$'`.

DefaultCookiePolicy.**strict_ns_set_path**
No permite establecer cookies cuyo path no concuerde con el request URI.

`strict_ns_domain` es una colección de indicadores. Su valor está construido o siendo construido en conjunto (por ejemplo, `DomainStrictNoDots|DomainStrictNonDomain` significa que ambos indicadores se encuentran establecidos).

DefaultCookiePolicy.**DomainStrictNoDots**
Cuando configuras las cookies, el “host prefix” no debe de contener un punto (ejem. `www.foo.bar.com` no puede establecer una cookie para `.bar.com`, porque `www.foo` tiene un punto).

DefaultCookiePolicy.**DomainStrictNonDomain**
Las cookies que no especifican explícitamente un cookie-attribute `domain` sólo se pueden retornar a un dominio igual al que estableció la cookie (ejem. `spam.example.com` no retornará las cookies de `example.com` que no tuvieran un cookie-attribute `domain`)

`DefaultCookiePolicy.`**`DomainRFC2965Match`**
Cuando se establecen las cookies, requiere de un completo emparejamiento de dominio **RFC 2965**.

Los siguientes atributos son proveídos por conveniencia, y son las combinaciones más útiles de los indicadores anteriores:

`DefaultCookiePolicy.`**`DomainLiberal`**
Equivalente a 0 (es decir, todos los anteriores indicadores de rigurosidad de los dominios Netscape están apagados).

`DefaultCookiePolicy.`**`DomainStrict`**
Equivalente a `DomainStrictNoDots|DomainStrictNonDomain`.

21.19.5 Objetos Cookie

Las instancias *`Cookie`* tienen atributos Python mas o menos correspondientes a los estándar cookie-attribute especificados en los diferentes estándares cookie. La correspondencia no es uno a uno, porque existen complicadas reglas para asignar valores predeterminados, debido a que los cookie-attribute `max-age` y `expires` contienen información equivalente, además que las cookies **RFC 2109** pueden estar "downgraded" por *`http.cookiejar`* desde la versión 1 a la 0 (Netscape) cookies.

La asignación a estos atributos no debe de ser necesario excepto en raras circunstancias en un método *`CookiePolicy`*. La clase no aplica consistencia interna, por lo que tienes que saber lo que estás haciendo si lo estas aplicando.

`Cookie.`**`version`**
Entero o *`None`*. Las cookies Netscape tienen *`version`* 0. **RFC 2965** y las cookies **RFC 2109** tienen una `version` del cookie-attribute de 1. Sin embargo, ten en cuenta que *`http.cookiejar`* puede "downgrade" las cookies RFC 2109 a Netscape cookies, en cuyo caso *`version`* es 0.

`Cookie.`**`name`**
Nombre de la cookie (un string).

`Cookie.`**`value`**
El valor de la cookie (una string), o *`None`*.

`Cookie.`**`port`**
String que representa un puerto o un conjunto de puertos (ejem. "80", o "80,8080") o incluso *`None`*.

`Cookie.`**`path`**
Ruta de la cookie (un string, ejem. `'/acme/rocket_launchers'`).

`Cookie.`**`secure`**
`True` si cookie sólo debe de ser retornada sobre una conexión segura.

`Cookie.`**`expires`**
Fecha de caducidad entera en segundos desde la parte temporal, o *`None`*. Véase también el método *`is_expired()`*.

`Cookie.`**`discard`**
`True` si esta es una sesión de cookies.

`Cookie.`**`comment`**
Comentario string del servidor explicando la función de esta cookie, o *`None`*.

`Cookie.`**`comment_url`**
URL que enlaza a un comentario desde un servidor explicando la función de esta cookie, o *`None`*.

`Cookie.`**`rfc2109`**
`True` si esta cookie fue recibida como una cookie **RFC 2109** (es decir, la cookie fue recibida en una cabecera *`Set-Cookie`*, y el valor del cookie-attribute Versión en la cabecera fue de 1). Este atributo es proveído porque *`http.cookiejar`* puede "degradar" cookies RFC 2109 a cookies Netscape, en cuyo caso *`version`* es 0.

Cookie.**port_specified**
: `True` si un puerto o un conjunto de puertos fue explícitamente especificado por el servidor (en la cabecera *Set-Cookie* / *Set-Cookie2*).

Cookie.**domain_specified**
: `True` si un domino fue explícitamente especificado por el servidor.

Cookie.**domain_initial_dot**
: `True` si el dominio explícitamente especificado por el servidor empieza con un punto (`'.'`).

Las cookies pueden tener cookie-attribute no estándar adicionales. Se pueden acceder a estos usando los siguientes métodos:

Cookie.**has_nonstandard_attr**(*name*)
: Retorna `True` si cookie tiene nombrado el cookie-attribute.

Cookie.**get_nonstandard_attr**(*name*, *default=None*)
: Si la cookie tiene nombrado el cookie-attribute, retorna su valor. De otra manera, retorna *default*.

Cookie.**set_nonstandard_attr**(*name*, *value*)
: Retorna el valor del nombrado cookie-attribute.

La clase *Cookie* también define el siguiente método:

Cookie.**is_expired**(*now=None*)
: `True` si la cookie ha excedido el tiempo que el servidor solicitó para que expirara. Si *now* es dado (en segundos desde la parte temporal), retorna si la cookie ha expirado en el momento especificado.

21.19.6 Ejemplos

El primer ejemplo muestra el uso más común de *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/")
```

Este ejemplo ilustra como abrir una URL usando tus cookies Netscape, Mozilla, o Lynx (asume la convención Unix/Netscape para la ubicación del archivo de cookies):

```
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/")
```

El siguiente ejemplo ilustra el uso de *DefaultCookiePolicy*. Activa las cookies **RFC 2965**, es más estricto con respecto a los dominios cuando se establecen o se retornan cookies Netscape, y bloquea algunos dominios para establecer o retornar cookies:

```
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/")
```

21.20 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*

21.21 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 construidos 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.

The following parameters govern the use of the returned proxy instance. If *allow_none* is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a *TypeError*. This is a commonly used extension to the XML-RPC specification, but isn't supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php for a description. The *use_builtin_types* flag can be used to cause date/time values to be presented as *datetime.datetime* objects and binary data to be presented as *bytes* objects; this flag is false by default. *datetime.datetime*, *bytes* and *bytearray* objects may be passed to calls. The *headers* parameter is an optional sequence of HTTP headers to send with each request, expressed as a sequence of 2-tuples representing the header name and value. (e.g. `[('Header-Name', 'value')]`). The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

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`	*bool*
`int`, `i1`, `i2`, `i4`, `i8` or `biginteger`	*int* en el rango de -2147483648 a 2147483647. Los valores obtienen la etiqueta `<int>` .
`double` o `float`	*float*. Los valores obtienen la etiqueta `<double>`.
`string`	*str*
`array`	*list* o *tuple* que contiene elementos determinados. Las matrices se retornan como *lists*.
`struct`	*dict*. 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 *__dict__* .
`dateTime.iso8601`	*DateTime* o *datetime.datetime*. El tipo retornado depende de los valores de los indicadores *use_builtin_types* y *use_datetime* .
`base64`	*Binary*, *bytes* o *bytearray*. El tipo retornado depende del valor de la marca *use_builtin_types*.
`nil`	La constante `None`. Solo se permite pasar si *allow_none* es verdadero.
`bigdecimal`	*decimal.Decimal*. 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: Added support of type tags with prefixes (e.g. `ex:nil`). Added support of unmarshalling additional types used by Apache XML-RPC implementation for numerics: `i1`, `i2`, `i8`, `biginteger`, `float` and `bigdecimal`. See https://ws.apache.org/xmlrpc/types.html for a description.

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.

21.21.1 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)))
```

21.21.2 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"))
```

21.21.3 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*.

Los datos codificados tendrán líneas nuevas cada 76 caracteres según RFC 2045 sección 6.8 **RFC 2045#section-6.8**, que era la especificación estándar de facto base64 cuando se escribió la especificación XML-RPC.

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)
```

21.21.4 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

Un entero que indica el tipo de falla.

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

```

(continué en la próxima página)

(proviene de la página anterior)

```
proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
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)
```

21.21.5 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)
```

21.21.6 Objetos MultiCall

El objeto *MultiCall* proporciona una forma de encapsular múltiples llamadas a un servidor remoto en una sola solicitud[1].

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:

[1] Este enfoque se presentó por primera vez en una discusión en xmlrpc.com.

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

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))
```

21.21.7 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 lanzará 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.

21.21.8 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))
```

21.21.9 Ejemplo de Uso de Cliente y Servidor

Vea *Ejemplo de SimpleXMLRPCServer*.

Pie de notas

21.22 `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 construidos maliciosamente. Si necesita analizar sintácticamente datos no confiables o no autentificados, consulte *Vulnerabilidades XML*.

class xmlrpc.server.**SimpleXMLRPCServer**(*addr*, *requestHandler=SimpleXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_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_builtin_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_builtin_types*.

class xmlrpc.server.**CGIXMLRPCRequestHandler**(*allow_none=False*, *encoding=None*, *use_builtin_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_builtin_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_builtin_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*.

21.22.1 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 caracter 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 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 rota 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
```

The client that interacts with the above server is included in `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
```

21.22.2 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 caracter 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()
```

21.22.3 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*.

21.22.4 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.

21.22.5 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.

21.23 ipaddress — Biblioteca de manipulación IPv4 / IPv6

Source code: Lib/ipaddress.py

---

ipaddress proporciona las capacidades para crear, manipular y operar en redes y direcciones IPv4 e IPv6.

Las funciones y clases de este módulo facilitan el manejo de varias tareas relacionadas con las direcciones IP, incluida la verificación de si dos hosts están en la misma subred, la iteración sobre todos los hosts de una subred en particular, la verificación de si una cadena representa un valor válido o no. Dirección IP o definición de red, etc.

Esta es la referencia completa de la API del módulo; para obtener una descripción general y una introducción, consulte ipaddress-howto.

Nuevo en la versión 3.3.

21.23.1 Funciones de fábrica de conveniencia

El módulo *ipaddress* proporciona funciones de fábrica para crear cómodamente direcciones IP, redes e interfaces:

ipaddress.**ip_address**(*address*)
Retorna un objeto *IPv4Address* o *IPv6Address* según la dirección IP pasada como argumento. Se pueden proporcionar direcciones IPv4 o IPv6; los enteros menores que `2**32` se considerarán IPv4 de forma predeterminada. Se lanza 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* dependiendo de la dirección IP pasada como argumento. *address* es una cadena o entero que representa la red IP. Se pueden suministrar redes IPv4 o IPv6; los enteros menores que `2**32` se considerarán IPv4 de forma predeterminada. *strict* se pasa a *IPv4Network* o *IPv6Network* constructor. Se genera un *ValueError* si *address* no representa una dirección IPv4 o IPv6 válida, o si la red tiene bits de 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* dependiendo de la dirección IP pasada como argumento. *address* es una cadena o entero que representa la dirección IP. Se pueden proporcionar direcciones IPv4 o IPv6; los 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 los formatos IPv4 e IPv6 significa que los mensajes de error brindan información mínima sobre el error exacto, ya que las funciones no saben si se pretendía usar el formato IPv4 o IPv6. Se pueden obtener informes de errores más detallados llamando directamente a los constructores de clases específicos de la versión adecuada.

21.23.2 Direcciones IP

Objetos de dirección

Los objetos *IPv4Address* y *IPv6Address* comparten muchos atributos comunes. Algunos atributos que solo son significativos para direcciones IPv6 también son implementados por objetos *IPv4Address*, a fin de facilitar la escritura de código que maneje ambas versiones de IP correctamente. Los objetos de dirección son *hashable*, por lo que se pueden usar como claves en diccionarios.

class ipaddress.**IPv4Address**(*address*)
Construya una dirección IPv4. Se genera un *AddressValueError* si *address* no es una dirección IPv4 válida.

Lo siguiente constituye una dirección IPv4 válida:

1. Una cadena en notación de punto decimal, que consta de cuatro enteros decimales en el rango inclusivo 0-255, separados por puntos (por ejemplo, `192.168.0.1`). Cada entero representa un octeto (byte) en la dirección. No se toleran ceros iniciales para evitar confusiones con la notación octal.
2. Un número entero que cabe en 32 bits.
3. Un entero empaquetado en un objeto *bytes* de longitud 4 (primero el octeto más significativo).

```
>>> 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: Se toleran los ceros iniciales, incluso en casos ambiguos que parecen notación octal.

Distinto en la versión 3.10: Los ceros iniciales ya no se toleran y se tratan como un error. Las cadenas de direcciones IPv4 ahora se analizan tan estrictamente como glibc *inet_pton()*.

Distinto en la versión 3.9.5: El cambio anterior también se incluyó en Python 3.9 a partir de la versión 3.9.5.

Distinto en la versión 3.8.12: El cambio anterior también se incluyó en Python 3.8 a partir de la versión 3.8.12.

version
El número de versión apropiado: `4` para IPv4, `6` para IPv6.

max_prefixlen
El número total de bits en la representación de direcciones para esta versión: `32` para IPv4, `128` para IPv6.

El prefijo define el número de bits iniciales en una dirección que se comparan para determinar si una dirección es parte de una red o no.

compressed

exploded
Representación de cadena en notación decimal con puntos. Los ceros iniciales nunca se incluyen en la representación.

Como IPv4 no define una notación abreviada para direcciones con octetos establecidos en cero, estos dos atributos son siempre los mismos que `str(addr)` para direcciones IPv4. La exposición de estos atributos facilita la escritura de código de visualización que pueda manejar direcciones IPv4 e IPv6.

packed
La representación binaria de esta dirección: un objeto *bytes* de la longitud adecuada (primero el octeto más significativo). Son 4 bytes para IPv4 y 16 bytes para IPv6.

reverse_pointer
El nombre del registro PTR de DNS inverso para la dirección IP, por ejemplo:

```
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa'
```

Este es el nombre que podría usarse para realizar una búsqueda PTR, no el nombre de host resuelto en sí.

Nuevo en la versión 3.5.

is_multicast
`True` si la dirección está reservada para uso de multidifusión. Consulte **RFC 3171** (para IPv4) o **RFC 2373** (para IPv6).

is_private
`True` si la dirección está asignada para redes privadas. Consulte iana-ipv4-special-registry (para IPv4) o iana-ipv6-special-registry (para IPv6).

is_global
`True` si la dirección está asignada para redes públicas. Consulte iana-ipv4-special-registry (para IPv4) o iana-ipv6-special-registry (para IPv6).

Nuevo en la versión 3.4.

is_unspecified
`True` si la dirección no está especificada. Consulte **RFC 5735** (para IPv4) o **RFC 2373** (para IPv6).

is_reserved
`True` si la dirección está reservada IETF.

is_loopback
`True` si se trata de una dirección de bucle invertido. Consulte **RFC 3330** (para IPv4) o **RFC 2373** (para IPv6).

is_link_local
`True` si se trata de una dirección de bucle invertido. Consulte **RFC 3330** (para IPv4) o **RFC 2373** (para IPv6). `True` si la dirección está reservada para uso local de enlace. Ver **RFC 3927**.

`IPv4Address.__format__`(*fmt*)
Devuelve una representación de cadena de la dirección IP, controlada por una cadena de formato explícito. *fmt* puede ser uno de los siguientes: `'s'`, la opción predeterminada, equivalente a *str()*,‘ *“b”*“ para una cadena binaria con relleno de ceros, `'X'` o `'x'` para una representación hexadecimal en mayúsculas o minúsculas, o `'n'`, que es equivalente a `'b'` para direcciones IPv4 y `'x'` para IPv6. Para representaciones

binarias y hexadecimales, el especificador de formulario `'#'` y la opción de agrupación " "_"" están disponibles. `__format__` es utilizado por `format`, `str.format` y f-strings.

```
>>> format(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> '{:#b}'.format(ipaddress.IPv4Address('192.168.0.1'))
'0b11000000101010000000000000000001'
>>> f'{ipaddress.IPv6Address("2001:db8::1000"):s}'
'2001:db8::1000'
>>> format(ipaddress.IPv6Address('2001:db8::1000'), '_X')
'2001_0DB8_0000_0000_0000_0000_0000_1000'
>>> '{:#_n}'.format(ipaddress.IPv6Address('2001:db8::1000'))
'0x2001_0db8_0000_0000_0000_0000_0000_1000'
```

Nuevo en la versión 3.9.

class ipaddress.**IPv6Address**(*address*)

Construya 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 que consta de ocho grupos de cuatro dígitos hexadecimales, cada grupo representa 16 bits. Los grupos están separados por dos puntos. Esto describe una notación *explotada* (a mano). La cadena también se puede *comprimir* (notación abreviada) por varios medios. Consulte **RFC 4291** para obtener más detalles. Por ejemplo, `"0000:0000:0000:0000:0000:0abc:0007:0def"` se puede comprimir en `"::abc:7:def"`.

   Opcionalmente, la cadena también puede tener un ID de zona de alcance, expresado con un sufijo `%scope_id`. Si está presente, el ID de alcance no debe estar vacío y no puede contener `%`. Consulte **RFC 4007** para obtener más detalles. Por ejemplo, `fe80::1234%1` podría identificar la dirección `fe80::1234` en el primer enlace del nodo.

2. Un número 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')
>>> ipaddress.IPv6Address('ff02::5678%1')
IPv6Address('ff02::5678%1')
```

compressed

La forma corta de la representación de la dirección, con ceros a la izquierda en los grupos omitidos y la secuencia más larga de grupos que consta completamente de ceros colapsó en un solo grupo vacío.

Este también es el valor devuelto por `str(addr)` para las direcciones IPv6.

exploded

La forma larga de la representación de la dirección, con todos los ceros a la izquierda y los grupos que constan completamente de ceros incluidos.

Para los siguientes atributos y métodos, consulte 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 uso local del sitio. Tenga en cuenta que el espacio de direcciones local del sitio ha quedado obsoleto por **RFC 3879**. Utilice *`is_private`* para probar si esta dirección está en el espacio de direcciones locales únicas como se define en **RFC 4193**.

ipv4_mapped
Para las direcciones que parecen ser direcciones IPv4 asignadas (que comienzan con `::FFFF/96`), esta propiedad informará la dirección IPv4 incrustada. Para cualquier otra dirección, esta propiedad será `None`.

scope_id
Para las direcciones de ámbito definido por **RFC 4007**, esta propiedad identifica la zona particular del ámbito de la dirección a la que pertenece la dirección, como una cadena. Cuando no se especifica ninguna zona de alcance, esta propiedad será `None`.

sixtofour
Para direcciones que parecen ser direcciones 6to4 (comenzando con `2002::/16`) según lo definido por **RFC 3056**, esta propiedad informará la dirección IPv4 incrustada. Para cualquier otra dirección, esta propiedad será `None`.

teredo
Para las direcciones que parecen ser direcciones Teredo (que comienzan con `2001::/32`) según lo definido por **RFC 4380**, esta propiedad informará el par de direcciones IP `(servidor, cliente)` incrustado. Para cualquier otra dirección, esta propiedad será `None`.

`IPv6Address.`**`__format__`**(*fmt*)
Consulte la documentación del método correspondiente en *`IPv4Address`*.

Nuevo en la versión 3.9.

Conversión a cadenas y enteros

Para interoperar con interfaces de red como el módulo de socket, las direcciones deben convertirse en cadenas o números enteros. Esto se maneja usando las funciones integradas *`str()`* y *`int()`*

```
>>> 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
```

Tenga en cuenta que las direcciones de ámbito IPv6 se convierten en números enteros sin ID de zona de ámbito.

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 se pueden comparar con el conjunto habitual de operadores de comparación. Las mismas direcciones IPv6 con diferentes ID de zona de alcance no son iguales. 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
>>> IPv6Address('fe80::1234') == IPv6Address('fe80::1234%1')
False
>>> IPv6Address('fe80::1234%1') != IPv6Address('fe80::1234%2')
True
```

Operadores aritméticos

Los números enteros se pueden sumar o restar de los 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
```

21.23.3 Definiciones de red IP

Los objetos *IPv4Network* y *IPv6Network* proporcionan un mecanismo para definir e inspeccionar las definiciones de red IP. Una definición de red consta de 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 están enmascaradas (Y 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` consta de direcciones IP en el rango inclusivo de `192.168.1.0` a `192.168.1.255`.

Prefijo, máscara de red y máscara de host

Hay varias formas equivalentes de especificar máscaras de red IP. Un *prefix* `/<nbits>` es una notación que indica cuántos bits de orden superior se establecen en la máscara de red. Una máscara de red (*mask*) es una dirección IP con una cierta cantidad 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 de host * es el inverso lógico de una * máscara de red * y, a veces, se utiliza (por ejemplo, en las listas de control de acceso de Cisco) para indicar 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 objetos de dirección también son implementados por 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 se documentan para *IPv4Network*. Los objetos de red son *hashable*, por lo que se pueden usar como claves en diccionarios.

class ipaddress.**IPv4Network**(*address*, *strict=True*)

Construya una definición de red IPv4. *address* puede ser una de las siguientes:

1. Una cadena que consta de una dirección IP y una máscara opcional, separadas por una barra (`/`). La dirección IP es la dirección de red y la máscara puede ser un solo número, lo que significa que es un *prefix*, o una representación de cadena de caracteres de una dirección IPv4. Si es la última, la máscara es interpretada como una *net mask* si comienza con un campo no nulo, o como una *host mask* si comienza con un campo cero, con la excepción de una máscara de solo ceros la cual es tratada como una *net mask*. 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 número entero que cabe en 32 bits. Esto es equivalente a una red de una sola dirección, con la dirección de red *address* y la máscara `/32`.

3. Un entero empaquetado en un objeto *bytes* de longitud 4, big-endian. La interpretación es similar a una *address* entera.

4. Dos tuplas de una descripción de dirección y una máscara de red, donde la descripción de la dirección es una cadena, un entero de 32 bits, un entero empaquetado de 4 bytes o un objeto IPv4Address existente; y la máscara de red es un número entero que representa la longitud del prefijo (por ejemplo, `24`) o una cadena que representa la máscara de prefijo (por ejemplo, `255.255.255.0`).

Se genera un *AddressValueError* si *address* no es una dirección IPv4 válida. A *NetmaskValueError* se genera si la máscara no es válida para una dirección IPv4.

Si *strict* es `True` y los bits de host están configurados en la dirección proporcionada, entonces *ValueError* se genera. De lo contrario, los bits del 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 de IP del argumento es incompatible con `self`.

Distinto en la versión 3.5: Se agregó la forma de dos tuplas para el parámetro de constructor *address*.

version

max_prefixlen

Consulte la documentación del atributo correspondiente en *IPv4Address*.

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

Estos atributos son verdaderos para la red en su conjunto si lo son tanto para la dirección de red como para la dirección de transmisión.

network_address

La dirección de red de la red. La dirección de red y la longitud del prefijo juntos definen de forma única una red.

broadcast_address

La dirección de transmisión de la red. Todos los hosts de la red deben recibir los paquetes enviados a la dirección de transmisión.

hostmask
La máscara de host, como un objeto *IPv4Address*.

netmask
La máscara de red, como un objeto *IPv4Address*.

with_prefixlen

compressed

exploded
Una representación de cadena de la red, con la máscara en notación de prefijo.

`with_prefixlen` y `compressed` son siempre lo mismo que `str(red)`. `exploded` utiliza la forma explosionada de la dirección de red.

with_netmask
Una representación de cadena de la red, con la máscara en notación de máscara de red.

with_hostmask
Una representación de cadena 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()
Longitud de Devuelve un iterador sobre los hosts utilizables en la red. Los hosts utilizables son todas las direcciones IP que pertenecen a la red, excepto la dirección de red en sí y la dirección de transmisión de la red. Para redes con una longitud de máscara de 31, la dirección de red y la dirección de transmisión de red también se incluyen en el resultado. Las redes con una máscara de 32 devolverán una lista que contiene la dirección de host única, el prefijo de red, en bits.

```
>>> 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 total o parcialmente contenida en *other* o *other* está completamente contenido en esta red.

address_exclude(*network*)
Calcula las definiciones de red resultantes de eliminar la *network* dada de esta. Devuelve un iterador de objetos de red. Genera *ValueError* si *network* no está completamente contenido 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, según los valores de los argumentos. *prefixlen_diff* es la cantidad en la que se debe aumentar la longitud de nuestro 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*. Devuelve 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, según los valores de los argumentos. *prefixlen_diff* es la cantidad en la que se debe reducir 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*. Devuelve un solo objeto de red.

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> 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*)

Devuelve `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*)

Devuelve `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*)

Compare esta red con *other*. En esta comparación solo se consideran las direcciones de red; los bits de host no lo son. Devuelve `-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*)

Construya una definición de red IPv6. *address* puede ser una de las siguientes:

1. Una cadena que consta de una dirección IP y una longitud de prefijo opcional, separados por una barra (`/`). La dirección IP es la dirección de red y la longitud del prefijo debe ser un solo número, el *prefix*. Si no se proporciona una longitud de prefijo, se considera que es `/128`.

   Tenga en cuenta que las máscaras de red expandidas actualmente no son compatibles. Eso significa que `2001:db00::0/24` es un argumento válido mientras que `2001:db00::0/ffff:ff00::` no lo es.

2. Un número entero que cabe en 128 bits. Esto es equivalente a una red de una sola dirección, con la dirección de red *address* y la máscara `/128`.

3. Un entero empaquetado en un objeto *bytes* de longitud 16, big-endian. La interpretación es similar a una *address* entera.

4. Dos tuplas de una descripción de dirección y una máscara de red, donde la descripción de la dirección es una cadena, un entero de 128 bits, un entero empaquetado de 16 bytes o un objeto IPv6Address existente; y la máscara de red es un número entero que representa la longitud del prefijo.

Se genera un *AddressValueError* si *address* no es una dirección IPv6 válida. A *NetmaskValueError* se genera si la máscara no es válida para una dirección IPv6.

Si *strict* es `True` y los bits de host están configurados en la dirección proporcionada, entonces *ValueError* se genera. De lo contrario, los bits del host se enmascaran para determinar la dirección de red adecuada.

Distinto en la versión 3.5: Se agregó la forma de dos tuplas para el parámetro de constructor *address*.

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()

Devuelve un iterador sobre los hosts utilizables en la red. Los hosts utilizables son todas las direcciones IP que pertenecen a la red, excepto la dirección Anycast del Subnet-Router. Para redes con una longitud de máscara de 127, la dirección anycast del Subnet-Router también se incluye en el resultado. Las redes con una máscara de 128 devolverán una lista que contiene la única dirección de host.

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*)
Consulte la documentación del atributo correspondiente en *IPv4Network*.

is_site_local
Este atributo es verdadero para la red en su conjunto si es cierto tanto para la dirección de red como para la dirección de transmisión.

Operadores

Este atributo es verdadero para la red en su conjunto si es cierto para ambos objetos de red que 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 se pueden comparar con el conjunto habitual de operadores lógicos. Los objetos de red se ordenan primero por dirección de red y luego por máscara de red.

Iteración

Los objetos de red se pueden iterar para enumerar todas las direcciones que pertenecen a la red. Para la iteración, se devuelven *todos* los hosts, incluidos los hosts inutilizables (para hosts utilizables, use 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
```

21.23.4 Objetos de interfaz

Los objetos de interfaz son *hashable*, por lo que se pueden usar como claves en diccionarios.

class ipaddress.**IPv4Interface**(*address*)

Construya una interfaz IPv4. El significado de *address* es como en el constructor de *IPv4Network*, excepto que siempre se aceptan direcciones de host arbitrarias.

IPv4Interface es una subclase de *IPv4Address*, por lo que hereda todos los atributos de esa clase. Además, están disponibles los siguientes atributos:

ip

The address (*IPv4Address*) without network information.

```
>>> 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 de cadena 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 de cadena de la interfaz con la red como 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 cadena de la interfaz con la red como 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*)

Construya una interfaz IPv6. El significado de *address* es como en el constructor de *IPv6Network*, excepto que siempre se aceptan direcciones de host arbitrarias.

IPv6Interface es una subclase de *IPv6Address*, por lo que hereda todos los atributos de esa clase. Además, están disponibles los siguientes atributos:

ip

network

with_prefixlen

with_netmask

with_hostmask

Consulte la documentación del atributo correspondiente en *IPv4Interface*.

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 se pueden comparar con el conjunto habitual de operadores lógicos.

Para la comparación de igualdad (== y !=), Tanto la dirección IP como la red deben ser iguales para que los objetos sean iguales. Una interfaz no se comparará con ninguna dirección u objeto de red.

Para ordenar (<, >, etc) las reglas son diferentes. Los objetos de interfaz y dirección con la misma versión de IP se pueden comparar, y los objetos de dirección siempre se clasificarán antes que los objetos de interfaz. Primero se comparan dos objetos de interfaz por sus redes y, si son iguales, luego por sus direcciones IP.

21.23.5 Otras funciones de nivel de módulo

El módulo también proporciona las siguientes funciones de nivel de módulo:

ipaddress.**v4_int_to_packed**(*address*)

Represente una dirección como 4 bytes empaquetados en orden de red (big-endian). *address* es una representación entera de una dirección IP IPv4. A *ValueError* se genera si el número entero es negativo o demasiado grande para ser una dirección IP 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 entera de una dirección IP IPv6. A *ValueError* se genera si el número entero es negativo o demasiado grande para ser una dirección IP IPv6.

ipaddress.**summarize_address_range**(*first*, *last*)

Devuelve un iterador del rango de red resumido dadas la primera y la última dirección IP. *first* es el primero *IPv4Address* o *IPv6Address* en el rango y *last* es el último *IPv4Address* o *IPv6Address* en el rango. A *TypeError* se genera si *first* o *last* no son direcciones IP o no son de la misma versión. A *ValueError* se genera si *last* no es mayor que *first* o si la *first* versión de la dirección 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*)

Devuelve un iterador de los objetos contraídos *IPv4Network* o *IPv6Network*. *addresses* es un iterador de *IPv4Network* objetos *IPv6Network*. A *TypeError* se genera si *addresses* contiene objetos de versión mixta.

```
>>> [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*)

Devuelve una clave adecuada para clasificar entre redes y direcciones. Los objetos de dirección y red no se pueden ordenar de forma predeterminada; son fundamentalmente diferentes, por lo que la expresión

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

no tiene sentido. Sin embargo, hay ocasiones en las que es posible que desee que *ipaddress* ordene estos de todos modos. Si necesita hacer esto, puede usar esta función como el argumento *key* para *sorted()*.

obj es un objeto de red o de dirección.

21.23.6 Excepciones personalizadas

Para admitir informes de errores más específicos de los constructores de clases, el módulo define las siguientes excepciones:

exception ipaddress.**AddressValueError**(*ValueError*)

Cualquier error de valor relacionado con la dirección.

exception ipaddress.**NetmaskValueError**(*ValueError*)

Cualquier error de valor relacionado con la máscara de red.

CAPÍTULO 22

Servicios Multimedia

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 `wave` — Leer y escribir archivos WAV

Código fuente: Lib/wave.py

The *wave* module provides a convenient interface to the WAV sound format. Only files using `WAVE_FORMAT_PCM` are supported. Note that this does not include files using `WAVE_FORMAT_EXTENSIBLE` even if the subformat is PCM.

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.

`exception` `wave.`**`Error`**

Error que se produce cuando algo es imposible porque viola la especificación WAV o alcanza una deficiencia de implementación.

22.1.1 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.

22.1.2 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*.

22.2 colorsys — Conversiones entre sistemas de color

Código fuente: Lib/colorsys.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)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

CAPÍTULO 23

Internacionalización

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 *`lgettext()`*, *`ldgettext()`*, *`lngettext()`* y *`ldngettext()`*. Si se omite *codeset*, se retorna el enlace (*binding*) actual.

Deprecated since version 3.8, 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, 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*, *localedir=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 *localedir* es como en *bindtextdomain()*. *languages* opcionales es una lista de cadenas, donde cada cadena es un código de idioma.

Si no se proporciona *localedir*, 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:

localedir/*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*, *localedir=None*, *languages=None*, *class_=None*, *fallback=False*, *codeset=None*)

Retorna una instancia de `*Translations` basada en *domain*, *localedir* 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* is 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, 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, 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 operativo en la clase base, este método toma el objeto de archivo *fp* y lee los datos del archivo, inicializando su catálogo de mensajes. Si tiene un formato de archivo de catálogo de mensajes no admitido, debe sobrescribir este método para analizar su formato.

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 *lgettext()*.

Deprecated since version 3.8, 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 *lgettext()* y *lngettext()*.

Deprecated since version 3.8, removed in version 3.10.

set_output_charset(*charset*)
Cambiar la codificación utilizada para retornar mensajes traducidos.

Deprecated since version 3.8, 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.

El conjunto completo de pares clave/valor se colocan en un diccionario y se establece como la variable de instancia «protegida» `_info`.

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*)

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()*.

Advertencia: Estos métodos deben evitarse en Python 3. Consulte la advertencia para la función *lgettext()*.

Deprecated since version 3.8, 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.

There are a few tools to extract the strings meant for translation. The original GNU **gettext** only supported C or C++ source code but its extended version **xgettext** scans code written in a number of languages, including Python, to find strings marked as translatable. Babel is a Python internationalization library that includes a `pybabel` script to extract and compile message catalogs. François Pinard's program called **xpot** does a similar job and is available as part of his po-utils package.

(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.

Agregar configuración regional a su módulo

Si está aplicando configuración regional a su módulo, debe tener cuidado de no realizar cambios globales, por ejemplo, al espacio de nombres integrado. No debe utilizar la API GNU **gettext**, sino la API basada en clases.

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
```

Agregar configuración regional a su aplicación

Si está aplicando configuración regional a su aplicación, puede instalar la función `_()` globalmente en el espacio de nombres incorporado, generalmente en el archivo del controlador principal de su aplicación. Esto permitirá que todos los archivos específicos de su aplicación usen `_('...')` sin tener que instalarlo explícitamente en cada archivo.

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 subprocesos 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
LC_NUMERIC	`'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 *CHAR_MAX*, 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.
LC_MONETARY	`'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.

The function temporarily sets the LC_CTYPE locale to the LC_NUMERIC locale or the LC_MONETARY locale if locales are different and numeric or monetary strings are non-ASCII. This temporary change affects other threads.

Distinto en la versión 3.7: The function now temporarily sets the LC_CTYPE locale to the LC_NUMERIC locale in some cases.

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.

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.

Nota: The regular expressions for *YESEXPR* and *NOEXPR* use syntax suitable for the `regex()` function from the C library, which might differ from the syntax used in *re*.

---

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 *locale encoding* 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 si el *modo Python UTF-8* esta habilitado, siempre retorna 'UTF-8 ', se ignoran la *locale encoding* y el argumento *do_setlocale*.

La pre-inicializacioń de Python configura la configuración regional LC_CTYPE. Véase también la *codificación del sistema de archivos y manejador de errores*.

Distinto en la versión 3.7: La función ahora siempre retorna UTF-8 en Android o si el *Modo UTF-8 Python* 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*)
Formats a number *val* according to the current *LC_NUMERIC* setting. The format follows the conventions of the % operator. For floating point values, the decimal point is modified if appropriate. If *grouping* is True, also takes the grouping into account.

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* .

The returned string includes the currency symbol if *symbol* is true, which is the default. If *grouping* is `True` (which is not the default), grouping is done with the value. If *international* is `True` (which is not the default), the international currency symbol is used.

Nota: This function will not work with the "C" locale, so you have to set a locale via *setlocale()* first.

---

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.**localize**(*string*, *grouping=False*, *monetary=False*)
Convierte una cadena de números normalizada en una cadena de caracteres formateada siguiendo la configuración *LC_NUMERIC*.

Nuevo en la versión 3.10.

locale.**atof**(*string*, *func=float*)
Converts a string to a number, following the *LC_NUMERIC* settings, by calling *func* on the result of calling *delocalize()* on *string*.

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.

This value may not be available on operating systems not conforming to the POSIX standard, most notably Windows.

locale.**LC_NUMERIC**
Categoría de configuración regional para formateo de números. 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\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')    # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')   # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)   # restore saved locale
```

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.

CAPÍTULO 24

Frameworks de programa

Los módulos descritos en este capitulo 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

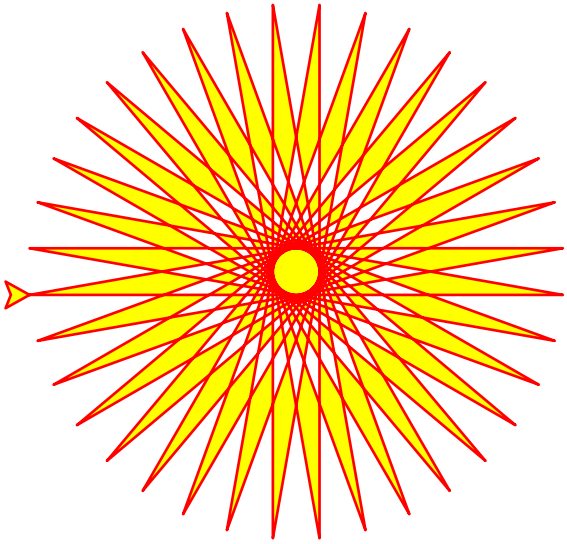
24.1.1 Introducción

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 pixeles 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()
shapesize() | *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()
clearscreen()
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 *ditance* 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 linea. 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 antihorario 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 *fordward/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

Retorna el ángulo entre la línea en la posición de la tortuga a la posición especificada en (x, y), el vector o la otra tortuga. Esto depende de la posición inicial de la tortuga, que depende del modo - «standard»/»world» o «logo».

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.**xcor**()

Devuelve la coordinada *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)
>>> 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 positivo

Establece el grosos 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 muevo 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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 *Screeen* *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)

Escribe texto - la representación de cadena de caracteres de *arg* - en la posición actual de la tortuga de acuerdo con *align* («izquierda», «centro» o «derecha») y con la fuente dada. Si *mov* es verdadero, el lápiz se mueve a la esquina inferior derecha del texto. De forma predeterminada, *move* es `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 *sretchfactor* y *outlinewidth* (contorno), que se establece con la función *`shapesize()`*.
- «*noresize*»: no se adapta la apariencia de la tortuga.

`resizemode("user")` es llamado por la función *`shapesize()`* cuando se usa con argumentos.

```
>>> 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

Return or set the pen's attributes x/y-stretchfactors and/or outline. Set resizemode to «user». If and only if resizemode is set to «user», the turtle will be displayed stretched according to its stretchfactors: *stretch_wid* is stretchfactor perpendicular to its orientation, *stretch_len* is stretchfactor in direction of its orientation, *outline* determines the width of the shape's outline.

```
>>> 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 de da un valor de inclinación: devuelve el factor actual, por ejemplo la tangente del ángulo de inclinación, por el cual se cortan la 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)
- **t12** – un número (opcional)

Establece o devuelve la matriz de transformación actual de la forma de la tortuga.

Si no se proporciona ninguno de los elementos de la matriz, retorna la matriz de transformación como una tupla de 4 elementos. De lo contrario, establezca los elementos dados y transforme la forma de tortuga de acuerdo con la matriz que consta de la primera fila t11, t12 y la segunda fila t21, t22. El determinante t11 * t22 - t12 * t21 no debe ser cero, de lo contrario se genera un error. Modifique el factor de estiramiento, el factor de corte y el ángulo de inclinación de acuerdo con la matriz dada.

```
>>> 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()
>>> 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 si. 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

Establecer o deshabilitar deshacer búfer. Si *size* es un número entero, se instala un búfer de deshacer vacío de un tamaño determinado. *size* da el número máximo de acciones de tortuga que se pueden deshacer mediante el método/función *undo()*. Si *size* es None, el búfer para deshacer está deshabilitado.

```
>>> 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. Add as many components to this object as desired, using the *addcomponent()* method.

   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**()

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.**clearscreen**()
Borra todos los dibujos y todas las tortugas del 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.

turtle.**reset**()

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.**resetscreen**()
Reinicia todas las tortugas de la pantalla a su estado inicial.

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, muevo 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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     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 enésima 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.

Ejemplo de una instancia TurtleScreen llamada `screen` y una instancia Turtle llamada `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 subproceso) - 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
«standard»	hacia la derecha (este)	sentido antihorario
«logo»	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`**`()`
Ata el método `bye()` al click del ratón sobre la pantalla.

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 descriptos 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`

Provides screen oriented methods like *bgcolor()* etc. that are described above.

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:

*type_*	*data*
«polygon»	una tupla para el polígono, por ejemplo: una tupla de par de coordenadas
«image»	una imagen (en esta forma solo se usa ¡internamente!)
«compound»	`None` (una forma compuesta tiene que ser construida usando el método *addcomponent()*)

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:
```

(continué en la próxima página)

(proviene de la página anterior)

```
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)
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.

The built in configuration would correspond to the following `turtle.cfg`:

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

Breve explicación de las entradas seleccionadas:

- The first four lines correspond to the arguments of the *`Screen.setup`* method.
- Line 5 and 6 correspond to the arguments of the method *`Screen.screensize`*.
- *shape* puede ser cualquiera de las formas integradas, por ejemplo: arrow, turtle, etc. Para obtener más información, pruebe con `help(shape)`.
- If you want to use no fill color (i.e. make the turtle transparent), you have to write `fillcolor = ""` (but all nonempty strings must not have quotes in the cfg file).
- 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	`tracer()`, retrasar (*delay*), `update()`
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	`ondrag()`
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 (shape, 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	`onclick()`
paz	elemental	turtle: apariencia y animación
penrose	embaldosado aperiódico con cometas y dardos	`stamp()`
planet_and_moon	simulación de sistema gravitacional	formas compuestas, `Vec2D`
rosette	un patrón del artículo de wikipedia sobre gráficos de tortuga (*turtle*)	`clone()`, `undo()`
round_dance	tortugas bailarinas que giran por parejas en dirección opuesta	formas compuestas, clonar tamaño de forma, tilt, get_shapepoly, update
sorting_animate	demostración visual de diferentes métodos de ordenamiento	alineación simple, aleatorización
árbol	un primer árbol de amplitud (gráfico, usando generadores)	`clone()`
two_canvases	diseño simple	tortugas en dos lienzos (*two_canvases*)
yinyang	otro ejemplo elemental	`circle()`

¡Diviértete!

24.1.8 Cambios desde Python 2.6

- The methods `Turtle.tracer`, `Turtle.window_width` and `Turtle.window_height` have been eliminated. Methods with these names and functionality are now available only as methods of `Screen`. The functions derived from these remain available. (In fact already in Python 2.6 these methods were merely duplications of the corresponding `TurtleScreen`/`Screen` methods.)
- The method `Turtle.fill()` has been eliminated. The behaviour of `begin_fill()` and `end_fill()` have changed slightly: now every filling process must be completed with an `end_fill()` call.
- A method `Turtle.filling` has been added. It returns a boolean value: `True` if a filling process is under way, `False` otherwise. This behaviour corresponds to a `fill()` call without arguments in Python 2.6.

24.1.9 Cambios desde Python 3.0

- The *Turtle* methods *shearfactor()*, *shapetransform()* and *get_shapepoly()* have been added. Thus the full range of regular linear transforms is now available for transforming turtle shapes. *tiltangle()* has been enhanced in functionality: it now can be used to get or set the tilt angle. *settiltangle()* has been deprecated.
- The *Screen* method *onkeypress()* has been added as a complement to *onkey()*. As the latter binds actions to the key release event, an alias: *onkeyrelease()* was also added for it.
- The method *Screen.mainloop* has been added, so there is no longer a need to use the standalone *mainloop()* function when working with *Screen* and *Turtle* objects.
- Two input methods have been added: *Screen.textinput* and *Screen.numinput*. These pop up input dialogs and return strings and numbers respectively.
- 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 linea 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 una reproducción inmediata:

```
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

def parse(arg):
```

(continué en la próxima página)

(proviene de la página anterior)

```
    '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 *turle 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

Current heading is 0
```

(continué en la próxima página)

(proviene de la página anterior)

```
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.

Obsoleto desde la versión 3.9: Pasar `None` para *s* lanzará una excepción en futuras versiones de Python.

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.

Advertencia: El módulo `shlex` **solo está diseñado para los intérpretes de comandos UNIX**.

No se garantiza que la función *quote()* sea correcta en shells no compatibles con POSIX o shells de otros sistemas operativos como Windows. La ejecución de comandos citados por este módulo en tales shells puede abrir la posibilidad de una vulnerabilidad de inyección de comandos.

Considere el uso de funciones que pasen argumentos mediante listas tal como *subprocess.run()* con `shell=False`.

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 ~'
>>> 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 apiladas 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 salteados. 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 `puntuation_chars`, los tokens se dividirán en espacios en blanco además de esos caracteres.

Distinto en la versión 3.8: El atributo `puntuation_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 `whitepace_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*.)

CAPÍTULO 25

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 fina capa orientada a objetos encima de Tcl/Tk. Para usar *tkinter*, no necesita escribir código Tcl, pero deberá consultar la documentación de Tk y, ocasionalmente, la documentación de Tcl. *tkinter* es un conjunto de envoltorios que implementan los widgets Tk como clases de Python.

Las principales virtudes de *tkinter* son que es rápido y que generalmente viene incluido con Python. Aunque su documentación estándar es débil, se dispone de buen material, que incluye: referencias, tutoriales, un libro y otros. *tkinter* también es famoso por tener un aspecto y una sensación obsoletos, que se ha mejorado enormemente en Tk 8.5. Sin embargo, hay muchas otras bibliotecas GUI en las que podría estar interesado. La wiki de Python enumera varias GUI frameworks and tools alternativas.

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 kit de herramientas de 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.

Tkinter soporta un amplio rango de versiones TCL/TK, construidos con o sin soporte de hilo. La versión oficial del binario de python incluye Tcl/Tk 8.6 con subprocesos. Vea el código fuente para el módulo `_tkinter` para mayor información acerca de las versiones soportadas.

Tkinter no es un subproceso liviano pero agrega una buena cantidad de su propia lógica para hacer la experiencia mas pythónico. Esta documentación se concentrará en estas adiciones y cambios, refiriéndose sobre la documentación oficial de Tcl-Tk para los detalles que no han sido cambiados.

Nota: Tcl/Tk 8.5 (2007) introdujo un moderno conjunto de componentes temáticos de interfaz de usuario en conjunto con una nueva API para ser usada. Ambas API tanto la antigua como la nueva están disponibles. La mayor parte de

la documentación que podrá encontrar en línea aún usa la antigua versión de la API y por lo tanto podría estar desactualizada.

Ver también:

- **TkDocs** Es un extenso tutorial sobre como crear interfaces de usuario con Tkinter. Explica conceptos clave y muestra enfoques recomendados para usar la API moderna.
- **Tkinter 8.5 reference: a GUI for Python** Documentación de referencia sobre Tkinter 8.5 donde se detallan clases disponibles, métodos y opciones.

Recursos de Tcl/Tk:

- **Comandos Tk** Referencia exhaustiva para cada uno de los comandos subyacentes utilizados por Tkinter.
- **Tcl/Tk Home Page** Documentación adicional y enlaces al núcleo de desarrollo de Tcl/Tk.

Libros:

- **Modern Tkinter for Busy Python Developers** Escrito por Mark Roseman (ISBN 978-1999149567)
- **Python y la programación de Tkinter** Escrito por Alan Moore (ISBN 978-1788835886)
- **Programming Python** Escrito por Mark Lutz, que cubre los tópicos de Tkinter de forma excelente. (ISBN 978-0596158101)
- **Tcl and the Tk Toolkit** Escrito por John Ousterhout, creador de Tcl/Tk y Kevin Jones, el cual no cubre Tkinter. (ISBN 978-0321336330)

25.1.1 Arquitectura

Tcl/Tk no es una biblioteca única mas bien consiste en unos pocos módulos distintivos entre si. cada uno separados por funcionalidades y que poseen su documentación oficial. Las versiones del binario de Python también incluyen un módulo adicional con esto.

Tcl Tcl es un lenguaje de programación interpretado y dinámico, tal como Python. Aunque puede ser usado en si mismo como un lenguaje de programación de uso general, lo más común es que se utilice en aplicaciones del lenguaje C como un motor de secuencias de comandos o como una interfaz hacia el kit de herramientas de Tk. La librería TCL contiene una interfaz con C que crea y gestiona una o más instancias del intérprete de TCL, corre los comandos en TCL en esas instancias y agrega comandos personalizados tanto en Tcl o C. Cada intérprete tiene su cola de eventos y entrega facilidades para enviar procesos a la cola y procesarlos. A diferencia de Python, el modelo de ejecución en Tcl es diseñado alrededor de cooperar en múltiples asignaciones y Tkinter ayuda a salvar estas diferencias (ver *Threading model* para más detalles).

Tk Tk is a Tcl package implemented in C that adds custom commands to create and manipulate GUI widgets. Each `Tk` object embeds its own Tcl interpreter instance with Tk loaded into it. Tk's widgets are very customizable, though at the cost of a dated appearance. Tk uses Tcl's event queue to generate and process GUI events.

Ttk El nuevo tema Tk (Ttk) es una nueva familia de widgets de Tk que proveen una mejor apariencia en diferentes plataformas mas que varios de los widgets clásicos de Tk. Ttk es distribuido como parte de Tk estando disponible a partir de la versión 8.5. Enlaces para Python son entregados en un módulo aparte, `tkinter.ttk`.

Internamente, Tk y Ttk usan funcionalidades del sistema operativo subyacente, en otras palabras, Xlib en Unix/X11, Cocoa en macOS o GDI en Windows.

Cuando su aplicación de Python una una clase en Tkinter, p.ej, para crear un widget, el módulo `tkinter` ensambla una cadena de comando Tcl/Tk. Entonces se pasa esa cadena de comando Tcl a un módulo binario interno `_tkinter`, que luego llama al intérprete Tcl para evaluarlo. El intérprete de Tcl llamará a los paquetes Tk y o Ttk, que a su vez realizarán llamadas a Xlib, Cocoa o GDI.

25.1.2 Módulos Tkinter

El soporte para Tkinter se distribuye para varios módulos. la mayor parte de las aplicaciones necesitaran el módulo base `tkinter`, así como también el módulo `tkinter.ttk`, el cual entrega el conjunto de widgets temáticos y la API correspondiente:

```
from tkinter import *
from tkinter import ttk
```

class tkinter.**Tk**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=True*, *sync=False*, *use=None*)

Construct a toplevel Tk widget, which is usually the main window of an application, and initialize a Tcl interpreter for this widget. Each instance has its own associated Tcl interpreter.

The `Tk` class is typically instantiated using all default values. However, the following keyword arguments are currently recognized:

screenName When given (as a string), sets the `DISPLAY` environment variable. (X11 only)

baseName Name of the profile file. By default, *baseName* is derived from the program name (`sys.argv[0]`).

className Name of the widget class. Used as a profile file and also as the name with which Tcl is invoked (*argv0* in *interp*).

useTk If `True`, initialize the Tk subsystem. The `tkinter.Tcl()` function sets this to `False`.

sync If `True`, execute all X server commands synchronously, so that errors are reported immediately. Can be used for debugging. (X11 only)

use Specifies the *id* of the window in which to embed the application, instead of it being created as an independent toplevel window. *id* must be specified in the same way as the value for the -use option for toplevel widgets (that is, it has a form like that returned by `winfo_id()`).

Note that on some platforms this will only work correctly if *id* refers to a Tk frame or toplevel that has its -container option enabled.

`Tk` reads and interprets profile files, named `.className.tcl` and `.baseName.tcl`, into the Tcl interpreter and calls `exec()` on the contents of `.className.py` and `.baseName.py`. The path for the profile files is the `HOME` environment variable or, if that isn't defined, then `os.curdir`.

tk

The Tk application object created by instantiating `Tk`. This provides access to the Tcl interpreter. Each widget that is attached the same instance of `Tk` has the same value for its `tk` attribute.

master

The widget object that contains this widget. For `Tk`, the *master* is `None` because it is the main window. The terms *master* and *parent* are similar and sometimes used interchangeably as argument names; however, calling `winfo_parent()` returns a string of the widget name whereas `master` returns the object. *parent*/*child* reflects the tree-like relationship while *master*/*slave* reflects the container structure.

children

The immediate descendants of this widget as a `dict` with the child widget names as the keys and the child instance objects as the values.

tkinter.**Tcl**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=False*)

La función `Tcl()` es una función de fábrica que crea un objeto muy similar al creado por la clase `Tk`, excepto que no inicializa el subsistema Tk. Esto suele ser útil cuando se maneja el intérprete de Tcl en un entorno en el que no se desean crear ventanas de nivel superior extrañas o donde no se puede (como los sistemas Unix/Linux sin un servidor X). Un objeto creado por el objeto `Tcl()` puede tener una ventana *Toplevel* creada (y el subsistema Tk inicializado) llamando a su método `loadtk()`.

Otros módulos que proporcionan soporte a Tk incluyen:

tkinter Módulos de Tkinter.

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.scrolledtext Widget de texto con una barra de desplazamiento vertical integrada.

tkinter.simpledialog Cuadros de diálogo simples y funciones útiles.

tkinter.ttk Un set de widgets temáticos fueron añadidos en Tk 8.5, entregando alternativas modernas para muchos de los widgets clásicos en el módulo principal de *tkinter*.

Módulos adicionales:

_tkinter Un módulo binario que contiene una interfaz de bajo nivel con Tcl/Tk. . Esto es automáticamente importado al módulo principal *tkinter* y nunca debe ser usado directamente por los p programadores de aplicaciones. Esto es usualmente una librería compartida (or DLL), pero podría en ciertos casos ser añadidos de forma estática por el interprete de Python.

idlelib Entorno de aprendizaje y desarrollo integrado de Python (IDLE). Basado en *tkinter*.

tkinter.constants Constantes simbólicas que pueden ser utilizadas en en lugar de cadenas de texto cuando se pasan varios parámetros a las llamadas de Tkinter. Automáticamente son importadas por el módulo principal *tkinter*.

tkinter.dnd De forma experimental está el soporte de arrastrar y soltar (*drag-and-drop*) para *tkinter*. Esto no se mantendrá cuando sea reemplazado por Tk DND.

tkinter.tix (Obsoleto) Un antiguo paquete de terceros para utilizar en Tcl/Tk que agrega un importante número de nuevos widgets. Se pueden encontrar mejores alternativas en el módulo *tkinter.ttk*.

turtle Gráficos de tortuga en una ventana Tk.

25.1.3 Guía de supervivencia de Tkinter

Esta sección no fue pensada en ser un tutorial exhaustivo sobre cada tópico de Tk o Tkinter. Para esto, puede consultar en alguno de los recursos externos que fueron presentados anteriormente. En vez de eso, esta apartado entrega una orientación breve sobre como es ve una aplicación en Tkinter, identifica conceptos claves de Tk y explica como el empaquetador de Tkinter está estructurado.

Es resto de esta sección lo ayudará a identificar las clases, métodos y opciones que necesitará en una aplicación de Tkinter y donde se puede encontrar documentación más detallada sobre esto, incluida en la guía de referencia oficial de Tcl/Tk.

Un programa simple de Hola Mundo

Comenzaremos a explorar Tkinter a través de una simple aplicación «Hola Mundo». Esta no es la aplicación mas pequeña que podríamos escribir, pero es suficiente para ilustrar algunos conceptos claves que necesitará saber.

```
from tkinter import *
from tkinter import ttk
root = Tk()
frm = ttk.Frame(root, padding=10)
frm.grid()
ttk.Label(frm, text="Hello World!").grid(column=0, row=0)
ttk.Button(frm, text="Quit", command=root.destroy).grid(column=1, row=0)
root.mainloop()
```

Después de las declaraciones de import, la línea siguiente será la de crear la instancia de la clase *Tk* la cual iniciativa Tk y crea el interprete asociado a Tcl. Esto también crea una ventana de nivel superior, conocida como la ventana raíz la cual sirve como la ventana principal de la aplicación.

En la siguiente línea se crea el marco del widget, el cual en este caso contendrá la etiqueta y el botón que crearemos después. El marco encaja dentro de la ventana raíz.

La próxima línea crea una etiqueta para el widget que contiene una cadena de texto estática. El método `grid()` es usado para especificar la posición del diseño de la etiqueta que está dentro del marco del widget, similar a como trabajan las tablas en HTML.

Es entonces cuando el botón del widget es creado y dejado a la derecha de la etiqueta. Una vez pulsado llamará al método `destroy()` de la ventana raíz.

Finalmente el método `mainloop()` muestra todo en pantalla y responde a la entrada del usuario hasta que el programa termina.

Conceptos importantes de Tk

Incluso con este sencillo programa se pueden mostrar conceptos claves de Tk:

widgets Una interfaz de usuario de Tkinter está compuesta de varios widgets individuales. Cada widget es representado como un objeto de Python, instanciado desde clases tales como `ttk.Frame`, `ttk.Label` y `ttk.Button`.

jerarquía de los widgets Los widgets se organizan en jerarquías. La etiqueta y el botón estaban dentro del marco el que a su vez estaba contenido dentro del marco raíz. Cuando se crea cada *child* del widget su *parent* es pasado como el primer argumento del constructor del widget.

opciones de configuración Los widgets tienen *configuration options* los cuales modifican su apariencia y comportamiento, como el texto que se despliega en la etiqueta o el botón. Diferentes clases de widgets tendrán diferentes conjuntos de opciones.

gestión del diseño de formularios Los widgets no son añadidos automáticamente a la interfaz de usuario una vez que han sido creados. Un *geometry manager* como controles de `grid` donde la interfaz de usuario es colocada.

ejecución del evento Tkinter reacciona a la entrada del usuario, realiza los cambios del programa e incluso actualiza lo que se muestra en pantalla solo cuando se ejecuta activamente un *event loop*. Si el programa no esta ejecutando el evento, su interfaz de usuario no se actualizará.

Entendiendo como funcionan los empaquetadores de Tcl/Tk

Cuando su aplicación usa las clases y métodos de Tkinter este de forma interna ensambla cadenas de texto representadas como comandos de Tcl/Tk y ejecuta esos comandos en el intérprete de Tcl que esta adjunto a la instancia de su aplicación *Tk*.

Tanto si se está intentando navegar en la documentación de referencia o intentando encontrar el método u opción correcta o adaptando algún código ya existente o depurando su aplicación en Tkinter, hay veces que es útil entender como son esos comandos subyacentes de Tcl/Tk.

Para ilustrar, aquí se tiene un equivalente en Tcl/Tk de la parte principal del script Tkinter anterior.

```
ttk::frame .frm -padding 10
grid .frm
grid [ttk::label .frm.lbl -text "Hello World!"] -column 0 -row 0
grid [ttk::button .frm.btn -text "Quit" -command "destroy ."] -column 1 -row 0
```

La sintaxis es similar a varios lenguajes de línea de comandos, donde la primera palabra es el comando que debe ser ejecutado con los argumentos que le siguen al comando, separados por espacios, Sin entrar en demasiados detalles, note lo siguiente:

- Los comandos usados para crear widgets (tales como `ttk::frame`) corresponden a las clases de widgets en Tkinter.

- Opciones de widget en Tcl (por ejemplo `-text`) corresponden a argumentos clave en Tkinter.
- Los widgets son reverenciados con un *pathname* in Tcl (por ejemplo `.frm.btn`) mientras que Tkinter no usa nombres sino que referencia a objetos.
- El lugar del widget en la jerarquía es codificada (jerárquicamente) en nombre de archivo, usando un `.` (punto) como separador de búsqueda. El nombre de ruta del archivo para la ventana raíz es un `.` (punto), la jerarquía no es definida por la ruta del archivo pero si por la especificación del widget padre cuando se crea cada widget hijo.
- Las operaciones que se implementan como *commands* separados en Tcl (por ejemplo `grid` or `destroy`) son representados como *methods* en objetos de widget de Tkinter. Como se verá en breve, en otras ocasiones, Tcl usa lo que parecen ser llamadas a métodos en objetos de widget, que reflejan más de cerca lo que se usa en Tkinter.

¿Cómo lo hago?, ¿Cómo funciona?

Si no esta seguro sobre como realizar ciertas acciones en Tkinter y no se puede encontrar información de forma rápida en el tutorial o en la documentación de referencias que estas usando, aquí hay varias estrategias que pueden ser de mucha ayuda.

Primero, recordar que los detalles sobre como cada widget trabaja en las diferentes versiones de Tcl/Tk pueden variar según la versión. Si está buscando información, asegúrese de que corresponde a la versión tanto de Python como de Tcl/Tk instaladas en su sistema.

Cuando se busca sobre como usar una API, puede ser muy útil el saber exactamente el nombre de la clase, opción o método que está utilizando. Tener esto en cuenta de antemano, ya sea en un shell de Python interactivo o con *print()*, puede ayudar a identificar lo que necesitas.

Para encontrar que opciones de configuración están disponibles en cada widget, se utiliza la llamada al método `configure()`, el cual retorna un diccionario que contiene una gran variedad de información sobre cada objeto, incluyendo valores por defecto y actuales. Utilice el `keys()` para saber los nombres de cada opción.

```
btn = ttk.Button(frm, ...)
print(btn.configure().keys())
```

La mayoría de los widgets tienen varias opciones de configuraciones en común, puede ser útil averiguar cuales son específicos de una clase de widget en particular. Comparando la lista de opciones de un widget sencillo como por ejemplo un *frame* es una de las formas de hacerlo.

```
print(set(btn.configure().keys()) - set(frm.configure().keys()))
```

De manera similar se pueden encontrar los métodos disponibles para un objeto de un widget utilizando la función standard *dir()*. Si lo intentas podrás ver alrededor de 200 métodos comunes para los widgets por lo que de nuevo, es muy importante identificar las especificaciones de un widget.

```
print(dir(btn))
print(set(dir(btn)) - set(dir(frm)))
```

Navegando en el manual de referencia de Tcl/Tk

Como se ha señalado, el manual oficial de referencia de Tk commands (en las páginas del manual) es a menudo la descripción mas precisa sobre que hacen las operaciones especificas de los widgets. Incluso sabiendo el nombre de la opción o método que necesitas, es posible que puedas encontrar información adicional.

Mientras todas las operaciones en Tkinter son implementadas como llamadas de método sobre objetos del widget, se puede ver cuantas operaciones de Tcl/Tk aparecen como comandos que toman el nombre de ruta del widget como el primer parámetro, seguido de parámetros opcionales, por ejemplo.

```
destroy .
grid .frm.btn -column 0 -row 0
```

Otros, sin embargo, se parecen más a métodos llamados en un objeto del widget (de hecho, cuando se crea un widget en Tcl / Tk, este a su vez crea un comando Tcl con el nombre de la ruta del widget, siendo el primer parámetro de ese comando el nombre de un método para llamar).

```
.frm.btn invoke
.frm.lbl configure -text "Goodbye"
```

En la documentación oficial de referencia para Tcl/Tk, se podrán encontrar mas operaciones que son similares a las llamadas a métodos de la pagina principal de un widget en especifico (p.ej., se podrá encontrar el método `invoke()` en la página principal de ttk::button) mientras que las funciones que toman un widget como un parámetro a menudo tienen su propia página principal (p.ej., grid).

Se podrán encontrar varias opciones en común y metidos en las páginas del manual options o ttk::widget, mientras que otras opciones se encuentran en las páginas del manual de una clase de widget específica.

También se puede encontrar cuantos métodos de Tkinter poseen nombres compuestos, p.ej, `winfo_x()`, `winfo_height()`, `winfo_viewable()`. Usted también encuentra información para todo esto en las páginas del manual en winfo <https://www.tcl.tk/man/tcl8.6/TkCmd/winfo.htm>‘_.

Nota: Puede ser algo confuso, pero también hay métodos en todos los widgets de Tkinter los cuales no operan directamente en el widget pero operan a nivel global, independiente del widget. Ejemplos de esto son los métodos para acceder al portapapeles o a llamados del sistema. (Ellos aparecen al ser implementados como métodos en la clase base `Widget` clase que todos los widgets en Tkinter heredan).

25.1.4 Modelo de subprocesamiento

Python y Tcl/Tk tienen modelos de subprocesamiento muy diferentes, con lo cual el módulo *tkinter* intenta ser un puente entre ambos. Si usted usa hilos, es muy posible que deba ser consciente de esto.

Un interprete de Python puede tener varios hilos asociados. En Tcl, diferentes hilos pueden ser creados, pero cada hilo tiene una instancia del interprete de Tcl asociada por separado. Los hilos también pueden crear mas de una instancia del interprete, aunque cada instancia del intérprete puede ser usada solo por el hilo que ha sido creado.

Cada objeto de la clase *Tk* creado por *tkinter* contiene un interprete para TCL. Esto también mantiene un seguimiento de cada hilo creado por el interprete. Las llamadas al módulo *tkinter* se pueden realizar desde cualquier módulo de Python. De forma interna si una llamada proviene desde un hilo diferente al creado por objeto *Tk*, un evento es publicado por la cola de eventos del interprete y cuando es ejecutado, el resultado es retornado al hilo de llamadas de Python.

Las aplicaciones de Tcl/Tk son normalmente basados en eventos, lo que significa que después de la inicialización, el interprete corre un bucle de eventos (p.ej. `Tk.mainloop()`) que responde a los eventos. Al ser un evento de un solo proceso, los controladores responden rápidamente, en caso contrario ellos bloquearán el procesamiento de otros eventos. Para evitar esto, cualquier procesamiento de larga duración no deben ser procesados en el controlador de eventos, o en caso contrario ser separados en pequeñas piezas usando temporizadores o correr el proceso en otro hilo. Esto es diferente a muchos kits de herramientas de GUI donde GUI se ejecuta en un hilo completamente separado del código de la aplicación, incluido el controlador de eventos.

Si el intérprete de Tcl no se está ejecutando el bucle del evento o procesando eventos, cualquier llamada a *tkinter* realizada desde hilos distintos al que ejecuta el intérprete de Tcl, fallará.

Existen varios casos especiales:

- Las librerías de Tcl/Tk pueden ser construidas para que no sean conscientes de los subprocesos. En este caso, el *tkinter* llama a la librería a partir del hilo originado desde Python, incluso si este es diferente al hilo creado en el intérprete de Tcl. Un cierre global asegura que solo ocurra una llamada a la vez.
- Mientras el *tkinter* permite crear una o mas instancias del objeto de *Tk* (con su propio intérprete), todos los interpretes son parte del mismo hilo compartido en la cola de eventos, lo cual se torna complicado rápidamente. En la practica, no se debe crear mas de una instancia de la clase *Tk* a la vez. De lo contrario, es lo mejor crearlos en hilos separados y asegurarse de que se esta ejecutando un hilo compatible con los subprocesos.

- Bloquear los controladores de eventos no son la única manera de prevenir que el intérprete de Tcl reingrese al bucle de eventos. Es incluso posible ejecutar una cantidad de bucles de eventos anidados o abandonar el bucle de forma completa, Si está realizando alguna tarea complicada cuando se trata de eventos o hilos hay que tener en cuenta estas posibilidades.
- Hay algunas funciones select *tkinter* que ahora solo funcionan cuando se les llama desde el hilo que creó el intérprete Tcl.

25.1.5 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	nombre de la opción para la búsqueda en la base de datos	`'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

El empaquetador es uno de los mecanismos de gestión de geometría de Tk. Los administradores de geometría se utilizan para especificar la posición relativa de los widgets dentro de su contenedor: su *master* mutuo. En contraste con el *placer* más engorroso (que se usa con menos frecuencia, y no cubrimos aquí), el empaquetador toma la especificación de relación cualitativa - *above*, *to the left of*, *filling*, etc - y funciona todo para determinar las coordenadas de ubicación exactas para usted.

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()

#
```

(continué en la próxima página)

(proviene de la página anterior)

```
# 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 Hay ocho nombres de *bitmaps* integrados: `'error'`, `'gray25'`, `'gray50'`, `'hourglass'`, `'info'`, `'questhead'`, `'question'`, `'warning'`. Para especificar un nombre de archivo de mapa de bits de X, indique la ruta completa del archivo, precedida por una `@`, como en `"@/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: `"#RRRGGGBBB"`, or 16 bit: `"#RRRRGGGGBBBB"` ranges, where R,G,B here represent any legal hex digit. See page 160 of Ousterhout's book for details.

cursor 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 Las distancias de pantalla se pueden especificar tanto en píxeles como en distancias absolutas. Los píxeles se dan como números y las distancias absolutas como cadenas con el carácter final que indica unidades: `c` para centímetros, `i` para pulgadas, `m` para milímetros, `p` para puntos de impresora. Por ejemplo, 3.5 pulgadas se expresa como `"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 is a string that denotes the target kind of event. (See the *bind(3tk)* man page, and page 201 of John Ousterhout's book, *Tcl and the Tk Toolkit (2nd edition)*, for details).

func es una función de Python que toma un argumento y se llama cuando ocurre el evento. La instancia del evento se pasa como el argumento mencionado. (Las funciones implementadas de esta manera a menudo se llaman *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)
```

Observe cómo se accede al campo del widget del evento en la devolución de llamada `turn_red()`. Este campo contiene el widget que capturó el evento de X. La siguiente tabla enumera los otros campos de eventos a los que puede acceder y cómo se indican en Tk, lo que puede ser útil cuando se hace referencia a las páginas del manual de Tk.

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 de tipo índice. Esto 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:

The Pillow package adds support for formats such as BMP, JPEG, TIFF, and WebP, among others.

25.1.6 Gestor de archivos

Tk permite registrar y anular el registro de una función de devolución de llamada que se llamará desde el mainloop de Tk cuando la E/S sea posible en un descriptor de archivo. Solo se puede registrar un controlador por descriptor de archivo. Código de ejemplo:

```
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.colorchooser — Diálogo de elección de color

Código fuente: Lib/tkinter/colorchooser.py

El módulo *tkinter.colorchooser* proporciona la clase *Chooser* como una interfaz para el diálogo del selector de color nativo. `Chooser` implementa una ventana de diálogo de elección de color modal. La clase `Chooser` hereda de la clase *Dialog*.

class tkinter.colorchooser.**Chooser**(*master=None*, ***options*)

tkinter.colorchooser.**askcolor**(*color=None*, ***options*)
Cree un cuadro de diálogo de elección de color. Una llamada a este método mostrará la ventana, esperará a que el usuario haga una selección y devolverá el color seleccionado (o `None`) a la persona que llama.

Ver también:

Módulo *tkinter.commondialog* Módulo de diálogo estándar de Tkinter

25.3 tkinter.font — Envoltorio de fuente Tkinter

Código fuente: Lib/tkinter/font.py

El módulo *tkinter.font* proporciona la clase *Font* para crear y usar fuentes con nombre.

Los diferentes pesos e inclinaciones de la fuente son:

tkinter.font.**NORMAL**
tkinter.font.**BOLD**
tkinter.font.**ITALIC**
tkinter.font.**ROMAN**

class tkinter.font.**Font**(*root=None*, *font=None*, *name=None*, *exists=False*, ***options*)
La clase *Font* representa una fuente con nombre. Las instancias *Font* reciben nombres únicos y se pueden especificar por su configuración de familia, tamaño y estilo. Las fuentes con nombre son el método de Tk para crear e identificar fuentes como un solo objeto, en lugar de especificar una fuente por sus atributos con cada aparición.

argumentos:

font - tupla de especificador de fuente (familia, tamaño, opciones)
name - nombre de fuente único
exists - self apunta a la fuente con nombre existente si es verdadera

opciones de palabras clave adicionales (ignoradas si se especifica *font*):

family - familia de la fuente, es decir, Courier, Times
size - tamaño de fuente
Si *size* es positivo, se interpreta como tamaño en puntos.
Si *size* es un número negativo, se trata su valor absoluto como tamaño en píxeles.
weight - énfasis de fuente (NORMAL, BOLD)

slant - ROMAN, ITALIC

underline - subrayado de fuente (0 - ninguno, 1 - subrayado)

overstrike - tachado de fuente (0 - ninguno, 1 - tachado)

actual(*option=None*, *displayof=None*)
Retorna los atributos de la fuente.

cget(*option*)
Recupera un atributo de la fuente.

config(***options*)
Modifica los atributos de la fuente.

copy()
Retorna una nueva instancia de la fuente actual.

measure(*text*, *displayof=None*)
Retorna la cantidad de espacio que ocuparía el texto en la pantalla especificada cuando se formatee en la fuente actual. Si no se especifica ninguna pantalla, se asume la ventana principal de la aplicación.

metrics(**options*, ***kw*)
Retorna datos específicos de la fuente. Las opciones incluyen:

***ascent* - distancia entre la línea de base y el punto más alto que un** el carácter de la fuente puede ocupar

***descent* - distancia entre la línea de base y el punto más bajo que un** el carácter de la fuente puede ocupar

***linespace* - separación vertical mínima necesaria entre dos** caracteres de la fuente que aseguran que no haya superposición vertical entre líneas.

fixed - 1 si la fuente es de ancho fijo en caso contrario 0

tkinter.font.**families**(*root=None*, *displayof=None*)
Retorna las diferentes familias de fuentes.

tkinter.font.**names**(*root=None*)
Retorna los nombres de las fuentes definidas.

tkinter.font.**nametofont**(*name*, *root=None*)
Retorna una representación *Font* de una fuente con nombre tk.

Distinto en la versión 3.10: Se agregó el parámetro *root*.

25.4 Diálogos Tkinter

25.4.1 tkinter.simpledialog —Diálogos de entrada estándar de Tkinter

Código fuente: Lib/tkinter/simpledialog.py

El módulo *tkinter.simpledialog* contiene clases y funciones convenientes para crear diálogos modales simples para obtener un valor de entrada del usuario.

tkinter.simpledialog.**askfloat**(*title*, *prompt*, ***kw*)
tkinter.simpledialog.**askinteger**(*title*, *prompt*, ***kw*)
tkinter.simpledialog.**askstring**(*title*, *prompt*, ***kw*)
Las tres funciones anteriores proporcionan diálogos que piden al usuario que introduzca un valor del tipo deseado.

class tkinter.simpledialog.**Dialog**(*parent*, *title=None*)
La clase base para los diálogos personalizados.

body(*master*)
Sobrescribir para construir la interfaz del dialogo y retornar el widget que debe tener el foco inicial.

buttonbox()
El comportamiento por defecto añade los botones OK y Cancelar. Sobrescribir para diseños de botones personalizados.

25.4.2 Diálogos de selección de archivos

Código fuente: Lib/tkinter/filedialog.py

El módulo *tkinter.filedialog* proporciona clases y funciones de factoría para crear ventanas de selección de archivos/directorios.

Diálogos nativos de carga/guardado

Las siguientes clases y funciones proporcionan ventanas de diálogo de archivos que combinan un aspecto nativo con opciones de configuración para personalizar el comportamiento. Los siguientes argumentos son aplicables a las clases y funciones enumeradas a continuación:

parent - la ventana sobre la que se situará el diálogo

title - el título de la ventana

initialdir - el directorio en el que inicia el diálogo

initialfile - el archivo seleccionado al abrir el diálogo

filetypes - una secuencia de tuplas (label, pattern), se permite el comodín "*"

defaultextension - extensión por defecto para añadir al archivo (diálogos de guardado)

multiple - cuando es verdadero, se permite la selección de múltiples elementos

Funciones estáticas de factoría

Las siguientes funciones, al ser invocadas, crean un diálogo modal de aspecto nativo, esperan la selección del usuario y retornan el(los) valor(es) seleccionado(s) o `Ninguno` a la llamada.

`tkinter.filedialog.`**askopenfile**(*mode='r', **options*)
`tkinter.filedialog.`**askopenfiles**(*mode='r', **options*)
Las dos funciones anteriores crean un diálogo *Open* y retornan el(los) objeto(s) de archivo(s) abierto(s) en modo de sólo lectura.

`tkinter.filedialog.`**asksaveasfile**(*mode='w', **options*)
Crea un diálogo *SaveAs* y retorna un objeto de un archivo abierto en modo de sólo escritura.

`tkinter.filedialog.`**askopenfilename**(**options*)

tkinter.filedialog.**askopenfilenames**(***options*)
Las dos funciones anteriores crean un diálogo *Open* y retornan el(los) nombre(s) de archivo(s) seleccionado(s) que corresponde(n) a un(os) archivo(s) existente(s).

tkinter.filedialog.**asksaveasfilename**(***options*)
Crea un diálogo *SaveAs* y retorna el nombre del archivo seleccionado.

tkinter.filedialog.**askdirectory**(***options*)

Pide al usuario que seleccione un directorio.

Argumento opcional adicional:

mustexist - determina si la selección debe ser un directorio existente.

class tkinter.filedialog.**Open**(*master=None*, ***options*)
class tkinter.filedialog.**SaveAs**(*master=None*, ***options*)
Las dos clases anteriores proporcionan ventanas de diálogo nativas para guardar y cargar archivos.

Clases de conveniencia

Las siguientes clases se utilizan para crear ventanas de archivos/directorios desde cero. Estas no emulan el aspecto nativo de la plataforma.

class tkinter.filedialog.**Directory**(*master=None*, ***options*)
Crear un diálogo que solicite al usuario que seleccione un directorio.

Nota: La clase *FileDialog* debe ser subclasificada para el manejo de eventos y comportamiento personalizados.

---

class tkinter.filedialog.**FileDialog**(*master*, *title=None*)
Crear un diálogo básico de selección de archivos.

cancel_command(*event=None*)
Activa la terminación de la ventana de diálogo.

dirs_double_event(*event*)
Manejador de eventos para el evento de doble clic sobre un directorio.

dirs_select_event(*event*)
Manejador de eventos para el evento de clic sobre un directorio.

files_double_event(*event*)
Manejador de eventos para el evento de doble clic sobre un archivo.

files_select_event(*event*)
Manejador de eventos para el evento de un solo clic sobre un archivo.

filter_command(*event=None*)
Filtra los archivos por directorio.

get_filter()
Recupera el filtro de archivos que se está utilizando actualmente.

get_selection()
Recupera el elemento seleccionado actualmente.

go(*dir_or_file=os.curdir*, *pattern='*'*, *default=''*, *key=None*)
Renderiza el diálogo e inicia el bucle de eventos.

ok_event(*event*)
Salir del diálogo retornando la selección actual.

quit(*how=None*)
Salir del diálogo retornando el nombre del archivo, si lo hay.

set_filter(*dir*, *pat*)
Establece el filtro de archivos.

set_selection(*file*)
Actualiza la selección de archivos actual a *archivo*.

class tkinter.filedialog.**LoadFileDialog**(*master*, *title=None*)
Una subclase de FileDialog que crea una ventana de diálogo para seleccionar un archivo existente.

ok_command()
Comprueba que se proporciona un archivo y que la selección indica un archivo ya existente.

class tkinter.filedialog.**SaveFileDialog**(*master*, *title=None*)
Una subclase de FileDialog que crea una ventana de diálogo para seleccionar un archivo de destino.

ok_command()
Comprueba si la selección apunta a un archivo válido que no es un directorio. Se requiere confirmación si se selecciona un archivo ya existente.

25.4.3 tkinter.commondialog — Plantillas de ventanas de diálogo

Código fuente: Lib/tkinter/commondialog.py

El módulo *tkinter.commondialog* proporciona la clase *Dialog* que es la clase base para los diálogos definidos en otros módulos de soporte.

class tkinter.commondialog.**Dialog**(*master=None*, ***options*)

show(*color=None*, ***options*)
Renderiza la ventana de diálogo.

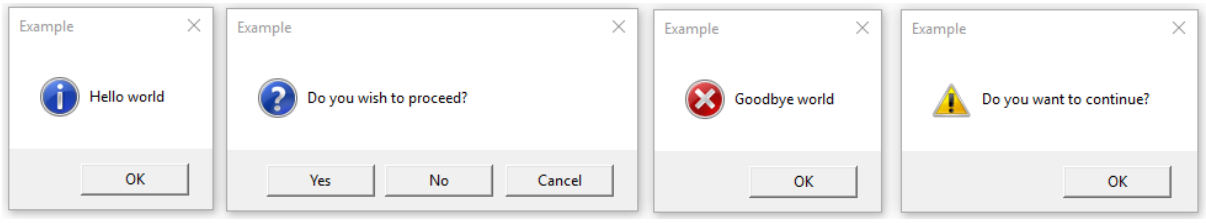
Ver también:

Módulos *tkinter.messagebox*, tut-files

25.5 tkinter.messagebox — Indicadores de mensajes de Tkinter

Código fuente: Lib/tkinter/messagebox.py

El módulo *tkinter.messagebox* proporciona una clase base de plantilla, así como una variedad de métodos convenientes para configuraciones de uso común. Los cuadros de mensaje son modales y devolverán un subconjunto de (Verdadero, Falso, OK, Ninguno, Sí, No) según la selección del usuario. Los estilos y diseños de cuadros de mensajes comunes incluyen, entre otros:



class tkinter.messagebox.**Message**(*master=None*, ***options*)
Crea un cuadro de mensaje de información predeterminado.

Cuadro de mensaje de información

tkinter.messagebox.**showinfo**(*title=None*, *message=None*, ***options*)

Cuadros de mensaje de advertencia

tkinter.messagebox.**showwarning**(*title=None*, *message=None*, ***options*)
tkinter.messagebox.**showerror**(*title=None*, *message=None*, ***options*)

Cuadros de mensajes de preguntas

tkinter.messagebox.**askquestion**(*title=None*, *message=None*, ***options*)
tkinter.messagebox.**askokcancel**(*title=None*, *message=None*, ***options*)
tkinter.messagebox.**askretrycancel**(*title=None*, *message=None*, ***options*)
tkinter.messagebox.**askyesno**(*title=None*, *message=None*, ***options*)
tkinter.messagebox.**askyesnocancel**(*title=None*, *message=None*, ***options*)

25.6 tkinter.scrolledtext — Widget de texto desplazado

Código fuente: Lib/tkinter/scrolledtext.py

El módulo *tkinter.scrolledtext* proporciona una clase del mismo nombre que implementa un widget de texto básico que tiene una barra de desplazamiento vertical configurada 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 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:

class tkinter.scrolledtext.**ScrolledText**(*master=None*, ***kw*)

frame
El marco que rodea el texto y los widgets de la barra de desplazamiento.

vbar
El widget de la barra de desplazamiento.

25.7 tkinter.dnd — Soporte de arrastrar y soltar

Código fuente: Lib/tkinter/dnd.py

Nota: Esto es experimental y quedará obsoleto cuando se sustituya por el Tk DND.

El módulo *tkinter.dnd* proporciona soporte para arrastrar y soltar objetos dentro de una sola aplicación, dentro de la misma ventana o entre ventanas. Para permitir que se arrastre un objeto, debe crear un enlace de evento para él que inicie el proceso de arrastrar y soltar. Por lo general, vincula un evento ButtonPress a una función de devolución de llamada que escribe (consulte *Enlaces y eventos*). La función debe llamar a *dnd_start()*, donde "fuente" es el objeto que se va a arrastrar y "evento" es el evento que invoca la llamada (el argumento de la función de devolución de llamada).

La selección de un objeto de destino ocurre de la siguiente manera:

1. Búsqueda de arriba hacia abajo del área debajo del mouse para el widget de destino

- El widget de destino debe tener un atributo *dnd_accept* invocable
- Si *dnd_accept* no está presente o devuelve None, la búsqueda se mueve al widget principal

- Si no se encuentra ningún widget de destino, el objeto de destino es None

2. Llama a *<old_target>.dnd_leave(source, event)*
3. Llama a *<new_target>.dnd_enter(source, event)*
4. Llama a *<target>.dnd_commit(source, event)* para notificar la caída
5. Llama a *<source>.dnd_end(target, event)* para señalar el final de arrastrar y soltar

class tkinter.dnd.**DndHandler**(*source*, *event*)
La clase *DndHandler* maneja los eventos de arrastrar y soltar que rastrean los eventos Motion y ButtonRelease en la raíz del widget de eventos.

cancel(*event=None*)
Cancela el proceso de arrastrar y soltar.

finish(*event*, *commit=0*)
Ejecuta el fin de las funciones de arrastrar y soltar.

on_motion(*event*)
Inspecciona el área debajo del mouse en busca de objetos de destino mientras se realiza el arrastre.

on_release(*event*)
Señal de fin de arrastre cuando se activa el patrón de liberación.

tkinter.dnd.**dnd_start**(*source*, *event*)
Función de fábrica para el proceso de arrastrar y soltar.

Ver también:

Enlaces y eventos

25.8 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

25.8.1 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:

Converting existing applications to use Tile widgets Una monografía (utilizando la terminología Tcl) sobre las diferencias que normalmente se encuentran al modificar aplicaciones para usar los nuevos widgets.

25.8.2 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*.

25.8.3 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
class	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 (*bindtags*) 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.
cursor	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.
takefocus	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.
style	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 `Scrollbar.set()` 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 statespec/value según lo definido por `Style.map()`, 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: • 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 `Widget.state()` 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 *background* 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* 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*)
Modify or inquire widget state. If *statespec* is specified, sets the widget state according to it and return a new *statespec* indicating which flags were changed. If *statespec* is not specified, returns the currently enabled state flags.

statespec es generalmente una lista o una tupla.

25.8.4 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
exportselection	Valor booleano. Si se establece, la selección del widget está vinculada a la selección del Administrador de ventanas (que se puede retornar invocando Misc.selection_get, por ejemplo).
justify	Especifica cómo el texto se alinea en el widget. Uno de «left», «center» o «right».
height	Especifica el largo/altura del cuadro de la lista desplegable, en filas.
postcommand	Un script (posiblemente registrado con Misc.register) que se llama inmediatamente antes de mostrar los valores. Puede especificar qué valores mostrar.
state	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.
textvariable	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 `tkinter.StringVar`.
values	Especifica la lista de valores que se mostrarán en el cuadro de lista desplegable.
width	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*.

25.8.5 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
from	Valor flotante. Si se establece, este es el valor mínimo al que se reducirá el botón de disminución. Debe escribirse como `from_` cuando se usa como argumento, ya que `from` es una palabra clave de Python.
to	Valor flotante. Si se establece, este es el valor máximo al que se incrementará el botón de incremento.
increment	Valor flotante. Especifica la cantidad por la cual los botones de incremento/disminución cambian el valor. Por defecto la cantidad es de 1.0.
values	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.
wrap	Valor booleano. Si es `True`, los botones de incremento y disminución pasarán del valor `to` al valor `from` o del valor `from` al valor `to`, respectivamente.
format	Valor de cadena. Especifica el formato de los números establecidos por los botones de incremento/disminución. Debe tener la forma «%W.Pf», donde W es el ancho de relleno del valor, P es la precisión, y "%" y "f" son literales.
command	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*.

25.8.6 Notebook

Ttk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently displayed window.

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. *Value* 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 `grid()`.
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 *`Notebook.enable_traversal()`*.

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
- The literal string «current», which identifies the currently selected tab
- 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.

The associated child window will be displayed, and the previously selected window (if different) is unmapped. If *tab_id* is omitted, returns the widget name of the currently selected pane.

tab(*tab_id*, *option=None*, ***kw*)
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.

25.8.7 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
orient	Puede ser «horizontal» o «vertical». Especifica la orientación de la barra de progreso.
length	Especifica la longitud del eje largo de la barra de progreso (ancho si horizontal, alto si es vertical).
mode	Puede ser «determinate» o «indeterminate».
maximum	Número que especifica el valor máximo. Por defecto el valor es 100.
value	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 *maximum*, es decir, la barra de progreso completa un «ciclo» cuando su valor aumenta en *maximum*.
variable	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.
phase	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.

25.8.8 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.

25.8.9 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, las ventanas de nivel superior incluyen automáticamente un control de tamaño integrado de forma predeterminada. Agregar un `Sizegrip` es inofensivo, ya que el agarre integrado solo enmascara el 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.

25.8.10 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
columns	Una lista de identificadores de columna, que especifican el número de columnas y sus nombres.
displaycolumns	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».
height	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.
padding	Especifica el relleno interno del widget. El relleno es una lista de hasta cuatro especificaciones de longitud.
selectmode	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.
show	Una lista que contiene cero o más de los siguientes valores, especificando qué elementos del árbol se van a mostrar. • tree: muestra las etiquetas del árbol en la columna #0. • headings: 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 show=»tree».

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 `True`/`False` 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 enésima columna de datos.
- Una cadena de la forma #n, donde n es un entero, especificando la enésima 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 open=True.
«TreeviewClose»	Generado justo después de establecer el elemento de resalto a open=False.

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 *columna* 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.

25.8.11 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:

Tcl'2004 conference presentation 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()

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()
```

(continué en la próxima página)

(proviene de la página anterior)

```
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*, *layoutspec=None*)
Define el diseño del widget para un *estilo* dado. Si se omite *layoutspec*, retorna la especificación de diseño para un estilo determinado.

layoutspec, 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

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 themename, 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 le tema actual.

element_options(*elementname*)
Retorna la lista de opciones de *elementname*.

theme_create(*themename*, *parent=None*, *settings=None*)
Crea un tema nuevo.

Es un error si *themename* 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(*themename*, *settings*)
Establece temporalmente el tema actual en *themename*, 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*.

25.9 `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.

25.9.1 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')
```

25.9.2 Widgets de Tix

Tix introduces over 40 widget classes to the *tkinter* repertoire.

Widgets Básicos

class tkinter.tix.**Balloon**
A Balloon that pops up over a widget to provide help. When the user moves the cursor inside a widget to which a Balloon widget has been bound, a small pop-up window with a descriptive message will be shown on the screen.

class tkinter.tix.**ButtonBox**
The ButtonBox widget creates a box of buttons, such as is commonly used for `Ok Cancel`.

class tkinter.tix.**ComboBox**
The ComboBox widget is similar to the combo box control in MS Windows. The user can select a choice by either typing in the entry subwidget or selecting from the listbox subwidget.

class tkinter.tix.**Control**
The Control widget is also known as the `SpinBox` widget. The user can adjust the value by pressing the two arrow buttons or by entering the value directly into the entry. The new value will be checked against the user-defined upper and lower limits.

class tkinter.tix.**LabelEntry**
The LabelEntry widget packages an entry widget and a label into one mega widget. It can be used to simplify the creation of «entry-form» type of interface.

class tkinter.tix.**LabelFrame**
The LabelFrame widget packages a frame widget and a label into one mega widget. To create widgets inside a LabelFrame widget, one creates the new widgets relative to the `frame` subwidget and manage them inside the `frame` subwidget.

class tkinter.tix.**Meter**
The Meter widget can be used to show the progress of a background job which may take a long time to execute.

class tkinter.tix.**OptionMenu**
The OptionMenu creates a menu button of options.

class tkinter.tix.**PopupMenu**
The PopupMenu widget can be used as a replacement of the `tk_popup` command. The advantage of the `Tix` *PopupMenu* widget is it requires less application code to manipulate.

class tkinter.tix.**Select**
The Select widget is a container of button subwidgets. It can be used to provide radio-box or check-box style of selection options for the user.

class tkinter.tix.**StdButtonBox**
The StdButtonBox widget is a group of standard buttons for Motif-like dialog boxes.

Selectores de Archivos

class tkinter.tix.**DirList**
The DirList widget displays a list view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class tkinter.tix.**DirTree**
The DirTree widget displays a tree view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class tkinter.tix.**DirSelectDialog**
The DirSelectDialog widget presents the directories in the file system in a dialog window. The user can use this dialog window to navigate through the file system to select the desired directory.

class 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**
The ExFileSelectBox widget is usually embedded in a tixExFileSelectDialog widget. It provides a convenient method for the user to select files. The style of the *ExFileSelectBox* widget is very similar to the standard file dialog on MS Windows 3.1.

class tkinter.tix.**FileSelectBox**
The FileSelectBox is similar to the standard Motif(TM) file-selection box. It is generally used for the user to choose a file. FileSelectBox stores the files mostly recently selected into a *ComboBox* widget so that they can be quickly selected again.

class tkinter.tix.**FileEntry**
The FileEntry widget can be used to input a filename. The user can type in the filename manually. Alternatively, the user can press the button widget that sits next to the entry, which will bring up a file selection dialog.

ListBox jerárquico

class tkinter.tix.**HList**
The HList widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierarchy.

class tkinter.tix.**CheckList**
The CheckList widget displays a list of items to be selected by the user. CheckList acts similarly to the Tk checkbutton or radiobutton widgets, except it is capable of handling many more items than checkbuttons or radiobuttons.

class tkinter.tix.**Tree**
The Tree widget can be used to display hierarchical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.

ListBox Tabular

class tkinter.tix.**TList**
The TList widget can be used to display data in a tabular format. The list entries of a *TList* widget are similar to the entries in the Tk listbox widget. The main differences are (1) the *TList* widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.

Gestores de Widgets

class tkinter.tix.**PanedWindow**
The PanedWindow widget allows the user to interactively manipulate the sizes of several panes. The panes can be arranged either vertically or horizontally. The user changes the sizes of the panes by dragging the resize handle between two panes.

class tkinter.tix.**ListNoteBook**
The ListNoteBook widget is very similar to the `TixNoteBook` widget: it can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages (windows). At one time only one of these pages can be shown. The user can navigate through these pages by choosing the name of the desired page in the `hlist` subwidget.

class tkinter.tix.**NoteBook**
The NoteBook widget can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages. At one time only one of these pages can be shown. The user can navigate through these pages by choosing the visual «tabs» at the top of the NoteBook widget.

Tipos de Imágenes

El módulo *tkinter.tix* añade:

- pixmap capabilities to all *tkinter.tix* and *tkinter* widgets to create color images from XPM files.
- Compound image types can be used to create images that consists of multiple horizontal lines; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from left to right. For example, a compound image can be used to display a bitmap and a text string simultaneously in a Tk `Button` widget.

Widgets Varios

class tkinter.tix.**InputOnly**
The InputOnly widgets are to accept inputs from the user, which can be done with the `bind` command (Unix only).

Gestor de Geometría de Formulario

Además, *tkinter.tix* mejora a *tkinter* al proporcionar:

class tkinter.tix.**Form**
The Form geometry manager based on attachment rules for all Tk widgets.

25.9.3 Comandos Tix

class tkinter.tix.**tixCommand**
The tix commands provide access to miscellaneous elements of `Tix`'s internal state and the `Tix` application context. Most of the information manipulated by these methods pertains to the application as a whole, or to a screen or display, rather than to a particular window.

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 `tex_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()*.

25.10 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

25.10.1 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.

Abrir módulo… Abre un módulo existente (buscar en sys.path).

Archivos recientes Abre una lista de archivos recientes. Haga click en alguno para abrirlo.

Module Browser 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… Save the current window with a Save As dialog. The file saved becomes the new associated file for the window. (If your file namager is set to hide extensions, the current extension will be omitted in the file name box. If the new filename has no “.”, “.py” and “.txt” will be added for Python and text files, except that on macOS Aqua,”.py” is added for all files.)

Guardar copia como… Save the current window to different file without changing the associated file. (See Save As note above about filename extensions.)

Imprimir ventana Imprime la ventana actual en la impresora predeterminada.

Close Window Close the current window (if an unsaved editor, ask to save; if an unsaved Shell, ask to quit execution). Calling `exit()` or `close()` in the Shell window also closes Shell. If this is the only window, also exit IDLE.

Exit IDLE Close all windows and quit IDLE (ask to save unsaved edit windows).

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.

Seleccionar todo Selecciona el contenido completo de 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.

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 Abre una lista navegable que permite la selección de valores existentes. Consultar *Terminaciones* en la sección Edición y Navegación a continuación.

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.

Show Call Tip 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.

Show Surrounding Parens Resalta los paréntesis.

Menú de formato (solo ventana del Editor)

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.

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 predeterminado aceptado por la comunidad de Python es de 4 espacios.

Strip Trailing Chitespace 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 Reinicia el shell para limpiar el entorno.

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.

Context menus

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*".

25.10.2 Editing and Navigation

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*.

Search and Replace

Any selection becomes a search target. However, only selections within a line work because searches are only performed within lines with the terminal newline removed. If `[x] Regular expresion` is checked, the target is interpreted according to the Python re module.

Terminaciones

Cuando se solicitan y están disponibles, se suministran complementos para los nombres de los módulos, los atributos de las clases o las funciones, o los nombres de los archivos. Cada método de solicitud muestra un cuadro para completar con los nombres existentes. (Para cualquier cuadro, cambie el nombre que se está completando y el elemento resaltado en el cuadro escribiendo y borrando caracteres; pulsando las teclas `Up`, `Down`, `PageUp`, `PageDown`, `Home`, y `End`; y con un solo clic dentro del cuadro. Cierra la caja con las teclas `Escape`, `Enter` y doble `Tab` o con clics fuera de la caja. Un doble clic dentro de la caja selecciona y cierra.

Una forma de abrir una caja es escribir un carácter clave y esperar un intervalo predefinido. Este intervalo es por defecto de 2 segundos; se puede modificar en el diálogo de configuración. (Para evitar las ventanas emergentes automáticas, establezca el retardo a un número grande de milisegundos, como 100000000). Para nombres de módulos importados o atributos de clases o funciones, escriba ".". Para nombres de archivos en el directorio raíz, escriba *`os.sep`* o *`os.altsep`* inmediatamente después de una comilla de apertura. (En Windows, se puede especificar una unidad de disco primero.) Muévase a los subdirectorios escribiendo un nombre de directorio y un separador.

En lugar de esperar, o después de cerrar una caja, abra una caja de finalización inmediatamente con Mostrar finalizaciones en el menú Edición. La tecla rápida por defecto es `C-espacio`. Si se teclea un prefijo para el nombre deseado antes de abrir el cuadro, la primera coincidencia o casi coincidencia se hace visible. El resultado es el mismo que si se introduce un prefijo después de mostrar la caja. Mostrar las terminaciones después de una cita completa de los nombres de archivo en el directorio actual en lugar de un directorio raíz.

Pulsar `Tab` después de un prefijo suele tener el mismo efecto que el de mostrar las terminaciones. (Sin prefijo, se indenta.) Sin embargo, si solo hay una coincidencia con el prefijo, esa coincidencia se añade inmediatamente al texto del editor sin abrir una caja.

Al invocar "Mostrar las terminaciones", o al pulsar `Tab` después de un prefijo, fuera de una cadena y sin un "." precedente, se abre un cuadro con las palabras clave, los nombres incorporados y los nombres disponibles a nivel de módulo.

Cuando se edita el código en un editor (a diferencia de Shell), aumentar los nombres disponibles a nivel de módulo mediante la ejecución de su código y no reiniciar el Shell después. Esto es especialmente útil después de añadir importaciones en la parte superior de un archivo. Esto también aumenta las posibles terminaciones de atributos.

Los cuadros de excluyen inicialmente los valores que empiezan por "_" o, en el caso de los módulos, no incluidos en "__all__". Se puede acceder a los nombres ocultos escribiendo "_" después de ".", ya sea antes o después de abrir la caja.

Sugerencias de llamada

Se muestra una sugerencia de llamada cuando se escribe `(` después del nombre de una función *accesible*. Una expresión de nombre puede incluir puntos y subíndices. Una sugerencia permanece hasta que se hace click, el cursor se mueve fuera del área de argumento o es escrito `)`. Cada vez que el cursor está en la parte del argumento de una definición, el menú o acceso directo muestra una sugerencia de llamada.

La sugerencia de llamada consiste en la firma de la función y el docstring hasta la primera línea en blanco de este último o la quinta línea no en blanco. (Algunas funciones incorporadas carecen de una firma accesible.) Un "/" o "*" en la firma indica que los argumentos anteriores o posteriores se pasan solo por posición o por nombre (palabra clave). Los detalles están sujetos a cambios.

En Shell, las funciones accesibles dependen de los módulos que se hayan importado en el proceso del usuario, incluidos los importados por el propio Idle, y de las definiciones que se hayan ejecutado, todo ello desde el último reinicio.

Por ejemplo, reinicie el Shell e ingrese `itertools.count(`. Una sugerencia de llamada aparece porque el Idle importa itertools en el proceso del usuario para su propio uso. (Esto puede cambiar.) Al ingresar `turtle.write(` nada aparece. El Idle no importa turtle. El menú o el acceso directo tampoco hace nada. Al ingresar `import turtle` y luego `turtle.write(` funcionará.

En un editor, las declaraciones de importación no tienen efecto hasta que se ejecuta el archivo. Es posible que desee ejecutar un archivo después de escribir las declaraciones de importación en la parte superior o ejecutar inmediatamente un archivo existente antes de editarlo.

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 window

In IDLE's Shell, enter, edit, and recall complete statements. (Most consoles and terminals only work with a single physical line at a time).

Submit a single-line statement for execution by hitting `Return` with the cursor anywhere on the line. If a line is extended with Backslash (`\`), the cursor must be on the last physical line. Submit a multi-line compound statement by entering a blank line after the statement.

When one pastes code into Shell, it is not compiled and possibly executed until one hits `Return`, as specified above. One may edit pasted code first. If one pastes more than one statement into Shell, the result will be a *`SyntaxError`* when multiple statements are compiled as if they were one.

Lines containing `RESTART` mean that the user execution process has been re-started. This occurs when the user execution process has crashed, when one requests a restart on the Shell menu, or when one runs code in an editor window.

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` while the cursor is on any previous command retrieves that command

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.

IDLE también resalta las palabras claves ligeras `match`, `case`, y _ en sentencias de patrones coincidentes. Sin embargo, este resaltado no es perfecto y será incorrecto en algunos raras ocasiones, incluyendo algunos _-s en patrones `case`.

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.

25.10.3 Startup and Code Execution

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

El IDLE usa un socket para comunicarse entre el proceso del GUI de IDLE y el proceso de ejecución del código de usuario. Se debe establecer una conexión cada vez que el Shell se inicia o reinicia. (Esto último se indica mediante una línea divisoria que dice "REINICIAR" (*RESTART*)). Si el proceso del usuario no se conecta al proceso de la GUI, muestra un cuadro de error `Tk` con un mensaje de "no se puede conectar" que dirige al usuario aquí. Y después se detiene.

Una falla de conexión específica en sistemas Unix es el resultado de una mala configuración de las reglas de enmascarado en algún lugar de la configuración del sistema de red. Cuando el IDLE inicia desde la terminal, puede verse un mensaje que comienza con **"** Invalid host: `. The valid value is ``127.0.0.1 (idlelib.rpc.LOCALHOST)``**. Se puede diagnosticar con ``tcpconnect -irv 127.0.0.1 6543` en una ventana de la terminal y `tcplisten <same args>` en otra.

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.

En sistemas basados en Unix con tcl/tk más antiguos que `8.6.11` (ver `About IDLE`) ciertos caracteres de ciertas fuentes pueden causar un fallo a tk con un mensaje hacia la terminal. Esto puede pasar cuando se inicia el IDLE para editar un archivo con dicho carácter o luego cuando se ingresa ese carácter. Si no se puede actualizar tcl/tk, entonces re configurar el IDLE para usar una fuente que funcione mejor

Ejecutando código del usuario

Con raras excepciones, el resultado de ejecutar el código de Python con el IDLE se supone es el mismo que al ejecutar el mismo código por el método predeterminado, directamente con Python en una consola de sistema en modo de texto o una ventana de terminal. Sin embargo, las diferentes interfaces y operaciones ocasionalmente afectan los resultados visibles. Por ejemplo, `sys.modules` comienza con más entradas y `threading.activeCount()` retorna 2 en lugar de 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`.

El envío de la salida de impresión de un proceso a un widget de texto en otro es más lento que la impresión a un terminal del sistema en el mismo proceso. Esto tiene el mayor efecto cuando se imprimen múltiples argumentos, ya que la cadena de cada argumento, cada separador y la nueva línea se envían por separado. Para el desarrollo, esto no suele ser un problema, pero si uno quiere imprimir más rápido en IDLE, formatea y une todo lo que quiere mostrar junto y luego imprime una sola cadena. Tanto las cadenas de formato como *`str.join()`* pueden ayudar a combinar campos y líneas.

IDLE's standard stream replacements are not inherited by subprocesses created in the execution process, whether directly by user code or by modules such as multiprocessing. If such subprocess use `input` from sys.stdin or `print` or `write` to sys.stdout or sys.stderr, IDLE should be started in a command line window. (On Windows, use `python` or `py` rather than `pythonw` or `pyw`.) The secondary subprocess will then be attached to that window for input and output.

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\a<\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 subproceso

Por defecto, el IDLE ejecuta el código de usuario en un subproceso 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 subproceso que ejecuta el servidor de ejecución de Python RPC. Esto puede ser útil si Python no puede crear el subproceso 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 subproceso predeterminado si es posible.

Obsoleto desde la versión 3.4.

25.10.4 Help and Preferences

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 zzdummy, un ejemplo que también se utiliza para realizar pruebas.

25.10.5 idlelib

Source code: Lib/idlelib

The Lib/idlelib package implements the IDLE application. See the rest of this page for how to use IDLE.

The files in idlelib are described in idlelib/README.txt. Access it either in idlelib or click Help => About IDLE on the IDLE menu. This file also maps IDLE menu items to the code that implements the item. Except for files listed under "Startup", the idlelib code is "private" in sense that feature changes can be backported (see **PEP 434**).

CAPÍTULO 26

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.

This module provides runtime support for type hints. The most fundamental support consists of the types *Any*, *Union*, *Callable*, *TypeVar*, and *Generic*. For a full specification, please see **PEP 484**. For a simplified introduction to type hints, see **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.

New features are frequently added to the `typing` module. The typing_extensions package provides backports of these new features to older versions of Python.

Ver también:

For a quick overview of type hints, refer to this cheat sheet.

The «Type System Reference» section of https://mypy.readthedocs.io/ – since the Python typing system is standardised via PEPs, this reference should broadly apply to most Python type checkers, although some parts may still be specific to mypy.

The documentation at https://typing.readthedocs.io/ serves as useful reference for type system features, useful typing related tools and typing best practices.

26.1.1 Relevant PEPs

Since the initial introduction of type hints in **PEP 484** and **PEP 483**, a number of PEPs have modified and enhanced Python's framework for type annotations. These include:

- **PEP 526: Syntax for Variable Annotations** *Introducing* syntax for annotating variables outside of function definitions, and `ClassVar`
- **PEP 544: Protocols: Structural subtyping (static duck typing)** *Introducing* `Protocol` and the `@runtime_checkable` decorator
- **PEP 585: Type Hinting Generics In Standard Collections** *Introducing* `types.GenericAlias` and the ability to use standard library classes as *generic types*
- **PEP 586: Literal Types** *Introducing* `Literal`
- **PEP 589: TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys** *Introducing* `TypedDict`
- **PEP 591: Adding a final qualifier to typing** *Introducing* `Final` and the `@final` decorator
- **PEP 593: Flexible function and variable annotations** *Introducing* `Annotated`
- **PEP 604: Allow writing union types as `X | Y`** *Introducing* `types.UnionType` and the ability to use the binary-or operator `|` to signify a *union of types*
- **PEP 612: Parameter Specification Variables** *Introducing* `ParamSpec` and `Concatenate`
- **PEP 613: Explicit Type Aliases** *Introducing* `TypeAlias`
- **PEP 647: User-Defined Type Guards** *Introducing* `TypeGuard`

26.1.2 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:

```
Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# passes type checking; 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 collections.abc import 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
# 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)`.

26.1.3 NewType

Use the *NewType* helper to create distinct types:

```
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:
    ...

# passes type checking
user_a = get_user_name(UserId(42351))

# fails type checking; 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)
```

Note that these checks are enforced only by the static type checker. At runtime, the statement `Derived = NewType('Derived', Base)` will make `Derived` a callable that immediately returns whatever parameter you pass it. That means the expression `Derived(some_value)` does not create a new class or introduce much overhead beyond that of a regular function call.

Más concretamente, la expresión `some_value is Derived(some_value)` será siempre verdadera en tiempo de ejecución.

No es válido crear un subtipo de `Derived`:

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not pass type checking
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.

Distinto en la versión 3.10: `NewType` es ahora una clase en lugar de una función. Existe un costo de tiempo de ejecución adicional cuando se llama a `NewType` a través de una función normal. Sin embargo, este costo se reducirá en 3.11.0.

26.1.4 Callable

Entidades que esperen llamadas a funciones con interfaces específicas puede ser anotadas usando `Callable[[Arg1Type, Arg2Type], ReturnType]`.

Por ejemplo:

```
from collections.abc 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

async def on_update(value: str) -> None:
    # Body
callback: Callable[[str], Awaitable[None]] = on_update
```

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]`.

Los invocables que toman otros invocables como argumentos pueden indicar que sus tipos de parámetros dependen unos de otros utilizando *ParamSpec*. Además, si ese invocable agrega o elimina argumentos de otros invocables, se puede utilizar el operador *Concatenate*. Toman la forma `Callable[ParamSpecVariable, ReturnType]` y `Callable[Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable], ReturnType]` respectivamente.

Distinto en la versión 3.10: `Callable` now supports *ParamSpec* and *Concatenate*. See **PEP 612** for more details.

Ver también:

The documentation for *ParamSpec* and *Concatenate* provides examples of usage in `Callable`.

26.1.5 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 collections.abc import Mapping, Sequence

def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

Generics can be parameterized by using a factory available in typing called *TypeVar*.

```
from collections.abc import Sequence
from typing import TypeVar

T = TypeVar('T')      # Declare type variable

def first(l: Sequence[T]) -> T:   # Generic function
    return l[0]
```

26.1.6 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.

The *Generic* base class defines `__class_getitem__()` so that `LoggedVar[T]` is valid as a type:

```
from collections.abc import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

A generic type can have any number of type variables. All varieties of *TypeVar* are permissible as parameters for a generic type:

```
from typing import TypeVar, Generic, Sequence

T = TypeVar('T', contravariant=True)
B = TypeVar('B', bound=Sequence[bytes], covariant=True)
S = TypeVar('S', int, str)

class WeirdTrio(Generic[T, B, 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 collections.abc import Sized
from typing import TypeVar, Generic

T = TypeVar('T')

class LinkedList(Sized, Generic[T]):
    ...
```

Cuando se hereda de clases genéricas, se pueden fijar algunas variables de tipo:

```
from collections.abc import Mapping
from typing import TypeVar

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 collections.abc import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
```

Son posibles los alias de tipos genéricos definidos por el usuario. Ejemplos:

```
from collections.abc import Iterable
from typing import TypeVar
S = TypeVar('S')
Response = Iterable[S] | int

# Return type here is same as Iterable[str] | int
def response(query: str) -> Response[str]:
    ...

T = TypeVar('T', int, float, complex)
Vec = Iterable[tuple[T, T]]
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 metaclase personalizable.

Los genéricos definidos por el usuario para expresiones de parámetros también se admiten a través de variables de especificación de parámetros con el formato `Generic[P]`. El comportamiento es coherente con las variables de tipo descritas anteriormente, ya que el módulo typing trata las variables de especificación de parámetros como una variable de tipo especializada. La única excepción a esto es que se puede usar una lista de tipos para sustituir un *ParamSpec*:

```
>>> from typing import Generic, ParamSpec, TypeVar

>>> T = TypeVar('T')
>>> P = ParamSpec('P')

>>> class Z(Generic[T, P]): ...
...
>>> Z[int, [dict, float]]
__main__.Z[int, (<class 'dict'>, <class 'float'>)]
```

Furthermore, a generic with only one parameter specification variable will accept parameter lists in the forms `X[[Type1, Type2, ...]]` and also `X[Type1, Type2, ...]` for aesthetic reasons. Internally, the latter is converted to the former, so the following are equivalent:

```
>>> class X(Generic[P]): ...
...
>>> X[int, str]
__main__.X[(<class 'int'>, <class 'str'>)]
>>> X[[int, str]]
__main__.X[(<class 'int'>, <class 'str'>)]
```

Tenga en cuenta que los genéricos con *ParamSpec* pueden no tener el `__parameters__` correcto después de la sustitución en algunos casos porque están destinados principalmente a la verificación de tipos estáticos.

Distinto en la versión 3.10: *Generic* ahora se puede parametrizar sobre expresiones de parámetros. Consulte *ParamSpec* y **PEP 612** para obtener más detalles.

A user-defined generic class can have ABCs as base classes without a metaclass conflict. Generic metaclasses are not supported. The outcome of parameterizing generics is cached, and most types in the typing module are *hashable* and comparable for equality.

26.1.7 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.

Esto significa que es posible realizar cualquier operación o llamada a un método en un valor de tipo *Any* y asignarlo a cualquier variable:

```
from typing import Any

a: Any = None
a = []          # OK
a = 2           # OK

s: str = ''
s = a           # OK

def foo(item: Any) -> int:
```

(continué en la próxima página)

(proviene de la página anterior)

```
    # Passes type checking; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

Notice that no type checking is performed when assigning a value of type *Any* to a more precise type. For example, the static type checker did not report an error when assigning `a` to `s` even though `s` was declared to be of type *str* and receives an *int* value at runtime!

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 type checking; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Passes type checking
    item.magic()
    ...

# Passes type checking, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")

# Passes type checking, 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.

26.1.8 Subtipado nominal vs estructural

Initially **PEP 484** defined the Python static type system as using *nominal subtyping*. This means that a class `A` is allowed where a class `B` is expected if and only if `A` is a subclass of `B`.

Este requisito también se aplicaba anteriormente a clases base abstractas (ABC), tales como *`Iterable`*. El problema con esta estrategia es que una clase debía de ser marcada explícitamente para proporcionar tal funcionalidad, lo que resulta poco *pythónico* (idiomático) y poco ajustado a lo que uno normalmente haría en un código Python tipado dinámicamente. Por ejemplo, esto sí se ajusta al **PEP 484**:

```
from collections.abc 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 collections.abc 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).

26.1.9 Contenido del módulo

El módulo define las siguientes clases, funciones y decoradores.

Nota: Este módulo define algunos tipos que son subclases de clases que ya existen en la librería estándar, y que además extienden *`Generic`* para soportar variables de tipo dentro de `[]`. Estos tipos se vuelven redundantes en Python 3.9 ya que las clases correspondientes fueron mejoradas para soportar `[]`.

Los tipos redundantes están descontinuados con Python 3.9 pero el intérprete no mostrará ninguna advertencia. Se espera que los verificadores de tipo marquen estos tipos como obsoletos cuando el programa a verificar apunte a Python 3.9 o superior.

Los tipos obsoletos serán removidos del módulo *`Generic`* en la primera versión de Python que sea lanzada 5 años después del lanzamiento de Python 3.9.0. Véase los detalles en **PEP 585** – *Sugerencias de tipo genéricas en las Colecciones Estándar*.

Primitivos especiales de tipado

Tipos especiales

Estos pueden ser usados como tipos en anotaciones y no soportan `[]`.

`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.`**`TypeAlias`**
Anotación especial para declarar explícitamente un *alias de tipo*. Por ejemplo:

```
from typing import TypeAlias

Factors: TypeAlias = list[int]
```

Consulte **PEP 613** para obtener más detalles sobre los alias de tipos explícitos.

Nuevo en la versión 3.10.

Formas especiales

Estas se pueden usar como anotaciones de tipo usando `[]`, cada cual tiene una sintaxis única.

`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 un 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*.

Obsoleto desde la versión 3.9: *builtins.tuple* now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

`typing.`**`Union`**
Tipo de unión; `Union[X, Y]` es equivalente a `X | Y` y significa X o Y.

To define a union, use e.g. `Union[int, str]` or the shorthand `int | str`. Using that shorthand is recommended. Details:

- 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] == int | str
```

- Cuando se comparan uniones, el orden de los argumentos se ignoran, p. ej.:

```
Union[int, str] == Union[str, int]
```

- No puede crear una subclase o instanciar un `Union`.
- No es posible escribir `Union[X][Y]`.

Distinto en la versión 3.7: No elimina subclases explícitas de una unión en tiempo de ejecución.

Distinto en la versión 3.10: Las uniones ahora se pueden escribir como `X | Y`. Consulte *union type expressions*.

typing.**Optional**

Tipo Optional.

`Optional[X]` es equivalente a `X | None` (o `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:
    ...
```

Distinto en la versión 3.10: Optional ahora se puede escribir como `X | None`. Consulte *union type expressions*.

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`*.

Los invocables que toman otros invocables como argumentos pueden indicar que sus tipos de parámetros dependen unos de otros utilizando *`ParamSpec`*. Además, si ese invocable agrega o elimina argumentos de otros invocables, se puede utilizar el operador *`Concatenate`*. Toman la forma `Callable[ParamSpecVariable, ReturnType]` y `Callable[Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable], ReturnType]` respectivamente.

Obsoleto desde la versión 3.9: *`collections.abc.Callable`* now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

Distinto en la versión 3.10: `Callable` now supports *ParamSpec* and *Concatenate*. See **PEP 612** for more details.

Ver también:

La documentación de *ParamSpec* y *Concatenate* proporciona ejemplos de uso con `Callable`.

`typing.`**`Concatenate`**

Se utiliza con *Callable* y *ParamSpec* para escribir anotar un invocable de orden superior que agrega, elimina o transforma parámetros de otro invocable. El uso tiene el formato `Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable]`. Actualmente, `Concatenate` solo es válido cuando se utiliza como primer argumento de un *Callable*. El último parámetro de `Concatenate` debe ser un *ParamSpec*.

Por ejemplo, para anotar un decorador `with_lock` que proporciona un *threading.Lock* a la función decorada, `Concatenate` puede usarse para indicar que `with_lock` espera un invocable que toma un `Lock` como primer argumento y retorna un invocable con un tipo de firma diferente. En este caso, el *ParamSpec* indica que los tipos de parámetros de los invocables retornados dependen de los tipos de parámetros de los invocables que se pasan en

```
from collections.abc import Callable
from threading import Lock
from typing import Concatenate, ParamSpec, TypeVar

P = ParamSpec('P')
R = TypeVar('R')

# Use this lock to ensure that only one thread is executing a function
# at any time.
my_lock = Lock()

def with_lock(f: Callable[Concatenate[Lock, P], R]) -> Callable[P, R]:
    '''A type-safe decorator which provides a lock.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> R:
        # Provide the lock as the first argument.
        return f(my_lock, *args, **kwargs)
    return inner

@with_lock
def sum_threadsafe(lock: Lock, numbers: list[float]) -> float:
    '''Add a list of numbers together in a thread-safe manner.'''
    with lock:
        return sum(numbers)

# We don't need to pass in the lock ourselves thanks to the decorator.
sum_threadsafe([1.1, 2.2, 3.3])
```

Nuevo en la versión 3.10.

Ver también:

- **PEP 612** - Variables de especificación de parámetros (el PEP que introdujo `ParamSpec` y `Concatenate`).
- *ParamSpec* y *Callable*.

`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 *objeto 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[BasicUser | ProUser]): ...
```

`Type[Any]` es equivalente a `Type`, que a su vez es equivalente a `type`, que es la raíz de la jerarquía de metaclases de Python.

Nuevo en la versión 3.5.2.

Obsoleto desde la versión 3.9: *builtins.type* now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

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.

Distinto en la versión 3.9.1: `Literal` ahora elimina los parámetros duplicados. Las comparaciones de igualdad de los objetos `Literal` ya no dependen del orden. Los objetos `Literal` ahora lanzarán una excepción *TypeError* durante las comparaciones de igualdad si uno de sus parámetros no es *hashable*.

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.**Annotated**

A type, introduced in **PEP 593** (`Flexible function and variable annotations`), to decorate existing types with context-specific metadata (possibly multiple pieces of it, as `Annotated` is variadic). Specifically, a type `T` can be annotated with metadata `x` via the typehint `Annotated[T, x]`. This metadata can be used for either static analysis or at runtime. If a library (or tool) encounters a typehint `Annotated[T, x]` and has no special logic for metadata `x`, it should ignore it and simply treat the type as `T`. Unlike the `no_type_check` functionality that currently exists in the `typing` module which completely disables typechecking annotations on a function or a class, the `Annotated` type allows for both static typechecking of `T` (which can safely ignore `x`) together with runtime access to `x` within a specific application.

En última instancia, la responsabilidad de cómo interpretar las anotaciones (si es que la hay) es de la herramienta o librería que encuentra el tipo `Annotated`. Una herramienta o librería que encuentra un tipo `Annotated` puede escanear las anotaciones para determinar si son de interés. (por ejemplo, usando `isinstance()`).

Cuando una herramienta o librería no soporta anotaciones o encuentra una anotación desconocida, simplemente debe ignorarla o tratar la anotación como el tipo subyacente.

Depende de la herramienta que consume las anotaciones decidir si el cliente puede tener varias anotaciones en un tipo y cómo combinar esas anotaciones.

Dado que el tipo `Annotated` permite colocar varias anotaciones del mismo (o diferente) tipo(s) en cualquier nodo, las herramientas o librerías que consumen dichas anotaciones están a cargo de ocuparse de potenciales duplicados. Por ejemplo, si se está realizando un análisis de rango, esto se debería permitir:

```
T1 = Annotated[int, ValueRange(-10, 5)]
T2 = Annotated[T1, ValueRange(-20, 3)]
```

Pasar `include_extras=True` a *get_type_hints()* permite acceder a las anotaciones extra en tiempo de ejecución.

Los detalles de la sintaxis:

- El primer argumento en `Annotated` debe ser un tipo válido
- Se permiten varias anotaciones de tipo (`Annotated` admite argumentos variádicos):

```
Annotated[int, ValueRange(3, 10), ctype("char")]
```

- `Annotated` debe ser llamado con al menos dos argumentos (`Annotated[int]` no es válido)
- Se mantiene el orden de las anotaciones y se toma en cuenta para chequeos de igualdad:

```
Annotated[int, ValueRange(3, 10), ctype("char")] != Annotated[
    int, ctype("char"), ValueRange(3, 10)
]
```

- Los tipos `Annotated` anidados son aplanados con los metadatos ordenados empezando por la anotación más interna:

```
Annotated[Annotated[int, ValueRange(3, 10)], ctype("char")] == Annotated[
    int, ValueRange(3, 10), ctype("char")
]
```

- Anotaciones duplicadas no son removidas:

```
Annotated[int, ValueRange(3, 10)] != Annotated[
    int, ValueRange(3, 10), ValueRange(3, 10)
]
```

- `Anotated` puede ser usado con alias anidados y genéricos:

```
T = TypeVar('T')
Vec = Annotated[list[tuple[T, T]], MaxLen(10)]
V = Vec[int]

V == Annotated[list[tuple[int, int]], MaxLen(10)]
```

Nuevo en la versión 3.9.

`typing.`**`TypeGuard`**

Formulario de mecanografía especial utilizado para anotar el tipo de retorno de una función de protección de tipo definida por el usuario. `TypeGuard` solo acepta un argumento de tipo único. En tiempo de ejecución, las funciones marcadas de esta manera deberían retornar un booleano.

`TypeGuard` tiene como objetivo beneficiar a *type narrowing*, una técnica utilizada por los verificadores de tipo estático para determinar un tipo más preciso de una expresión dentro del flujo de código de un programa. Por lo general, el estrechamiento de tipos se realiza analizando el flujo de código condicional y aplicando el estrechamiento a un bloque de código. La expresión condicional aquí a veces se denomina «protección de tipo»:

```
def is_str(val: str | float):
    # "isinstance" type guard
    if isinstance(val, str):
        # Type of ``val`` is narrowed to ``str``
        ...
    else:
        # Else, type of ``val`` is narrowed to ``float``.
        ...
```

A veces sería conveniente utilizar una función booleana definida por el usuario como protección de tipos. Dicha función debería usar `TypeGuard[...]` como su tipo de retorno para alertar a los verificadores de tipo estático sobre esta intención.

El uso de `-> TypeGuard` le dice al verificador de tipo estático que para una función determinada:

1. El valor de retorno es un booleano.
2. Si el valor de retorno es `True`, el tipo de su argumento es el tipo dentro de `TypeGuard`.

Por ejemplo:

```
def is_str_list(val: List[object]) -> TypeGuard[List[str]]:
    '''Determines whether all objects in the list are strings'''
    return all(isinstance(x, str) for x in val)

def func1(val: List[object]):
    if is_str_list(val):
        # Type of ``val`` is narrowed to ``List[str]``.
        print(" ".join(val))
    else:
        # Type of ``val`` remains as ``List[object]``.
        print("Not a list of strings!")
```

Si `is_str_list` es un método de clase o instancia, entonces el tipo en `TypeGuard` se asigna al tipo del segundo parámetro después de `cls` o `self`.

En resumen, la forma `def foo(arg: TypeA) -> TypeGuard[TypeB]: ...` significa que si `foo(arg)` retorna `True`, entonces `arg` se estrecha de `TypeA` a `TypeB`.

Nota: No es necesario que `TypeB` sea una forma más estrecha de `TypeA`; incluso puede ser una forma más amplia. La razón principal es permitir cosas como reducir `List[object]` a `List[str]` aunque este último no sea un subtipo del primero, ya que `List` es invariante. La responsabilidad de escribir protecciones de tipo seguro se deja al usuario.

`TypeGuard` also works with type variables. See **PEP 647** for more details.

Nuevo en la versión 3.10.

Tipos de construcción de genéricos

Estos no son utilizados en anotaciones. Son utilizados como bloques para crear tipos genéricos.

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.**TypeVar**
Variable de tipo.

Uso:

```
T = TypeVar('T')  # Can be anything
S = TypeVar('S', bound=str)  # Can be any subtype of str
A = TypeVar('A', str, bytes)  # Must be exactly 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 *Generic* para más información sobre tipos genéricos. Las funciones genéricas funcionan de la siguiente manera:

```
def repeat(x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n


def print_capitalized(x: S) -> S:
    """Print x capitalized, and return x."""
    print(x.capitalize())
    return x


def concatenate(x: A, y: A) -> A:
    """Add two strings or bytes objects together."""
    return x + y
```

Note that type variables can be *bound*, *constrained*, or neither, but cannot be both bound *and* constrained.

Constrained type variables and bound type variables have different semantics in several important ways. Using a *constrained* type variable means that the `TypeVar` can only ever be solved as being exactly one of the constraints given:

```
a = concatenate('one', 'two')  # Ok, variable 'a' has type 'str'
b = concatenate(StringSubclass('one'), StringSubclass('two'))  # Inferred type␣
↪of variable 'b' is 'str',
                                                               # despite
↪'StringSubclass' being passed in
c = concatenate('one', b'two')  # error: type variable 'A' can be either 'str'␣
↪or 'bytes' in a function call, but not both
```

Using a *bound* type variable, however, means that the `TypeVar` will be solved using the most specific type possible:

```
print_capitalized('a string')  # Ok, output has type 'str'

class StringSubclass(str):
    pass

print_capitalized(StringSubclass('another string'))  # Ok, output has type
↪'StringSubclass'
print_capitalized(45)  # error: int is not a subtype of str
```

Type variables can be bound to concrete types, abstract types (ABCs or protocols), and even unions of types:

```
U = TypeVar('U', bound=str|bytes)  # Can be any subtype of the union str|bytes
V = TypeVar('V', bound=SupportsAbs)  # Can be anything with an __abs__ method
```

Bound type variables are particularly useful for annotating *classmethods* that serve as alternative constructors. In the following example (by Raymond Hettinger), the type variable `C` is bound to the `Circle` class through the use of a forward reference. Using this type variable to annotate the `with_circumference` classmethod, rather than hardcoding the return type as `Circle`, means that a type checker can correctly infer the return type even if the method is called on a subclass:

```
import math

C = TypeVar('C', bound='Circle')

class Circle:
    """An abstract circle"""

    def __init__(self, radius: float) -> None:
        self.radius = radius

    # Use a type variable to show that the return type
    # will always be an instance of whatever ``cls`` is
    @classmethod
    def with_circumference(cls: type[C], circumference: float) -> C:
        """Create a circle with the specified circumference"""
        radius = circumference / (math.pi * 2)
        return cls(radius)


class Tire(Circle):
    """A specialised circle (made out of rubber)"""

    MATERIAL = 'rubber'


c = Circle.with_circumference(3)  # Ok, variable 'c' has type 'Circle'
t = Tire.with_circumference(4)  # Ok, variable 't' has type 'Tire' (not 'Circle
↪')
```

En tiempo de ejecución, `isinstance(x, T)` lanzará una excepción *TypeError*. En general, *isinstance()* y *issubclass()* no se deben usar con variables de tipo.

Type variables may be marked covariant or contravariant by passing `covariant=True` or `contravariant=True`. See **PEP 484** for more details. By default, type variables are invariant.

class typing.**ParamSpec**(*name*, *, *bound=None*, *covariant=False*, *contravariant=False*)

Variable de especificación de parámetros. Una versión especializada de *type variables*.

Uso:

```
P = ParamSpec('P')
```

Las variables de especificación de parámetros existen principalmente para el beneficio de los verificadores de tipo estático. Se utilizan para reenviar los tipos de parámetros de un invocable a otro invocable, un patrón que se encuentra comúnmente en funciones y decoradores de orden superior. Solo son válidos cuando se utilizan en `Concatenate`, o como primer argumento de `Callable`, o como parámetros para genéricos definidos por el usuario. Consulte *Generic* para obtener más información sobre tipos genéricos.

Por ejemplo, para agregar un registro básico a una función, se puede crear un decorador `add_logging` para registrar llamadas a funciones. La variable de especificación de parámetros le dice al verificador de tipo que el invocable pasado al decorador y el nuevo invocable retornado por él tienen parámetros de tipo interdependientes:

```
from collections.abc import Callable
from typing import TypeVar, ParamSpec
import logging

T = TypeVar('T')
P = ParamSpec('P')

def add_logging(f: Callable[P, T]) -> Callable[P, T]:
    '''A type-safe decorator to add logging to a function.'''
```

(continué en la próxima página)

(proviene de la página anterior)

```
    def inner(*args: P.args, **kwargs: P.kwargs) -> T:
        logging.info(f'{f.__name__} was called')
        return f(*args, **kwargs)
    return inner

@add_logging
def add_two(x: float, y: float) -> float:
    '''Add two numbers together.'''
    return x + y
```

Sin `ParamSpec`, la forma más sencilla de anotar esto anteriormente era usar un *`TypeVar`* con `Callable[..., Any]` enlazado. Sin embargo, esto causa dos problemas:

1. El verificador de tipo no puede verificar la función `inner` porque `*args` y `**kwargs` deben escribirse *`Any`*.
2. Es posible que se requiera *`cast()`* en el cuerpo del decorador `add_logging` al retornar la función `inner`, o se debe indicar al verificador de tipo estático que ignore el `return inner`.

args

kwargs

Dado que `ParamSpec` captura tanto parámetros posicionales como de palabras clave, `P.args` y `P.kwargs` se pueden utilizar para dividir un `ParamSpec` en sus componentes. `P.args` representa la tupla de parámetros posicionales en una llamada determinada y solo debe usarse para anotar `*args`. `P.kwargs` representa la asignación de parámetros de palabras clave a sus valores en una llamada determinada y solo debe usarse para anotar `**kwargs`. Ambos atributos requieren que el parámetro anotado esté dentro del alcance. En tiempo de ejecución, `P.args` y `P.kwargs` son instancias respectivamente de *`ParamSpecArgs`* y *`ParamSpecKwargs`*.

Las variables de especificación de parámetros creadas con `covariant=True` o `contravariant=True` se pueden utilizar para declarar tipos genéricos covariantes o contravariantes. También se acepta el argumento `bound`, similar a *`TypeVar`*. Sin embargo, la semántica real de estas palabras clave aún no se ha decidido.

Nuevo en la versión 3.10.

Nota: Solo las variables de especificación de parámetros definidas en el ámbito global pueden ser serializadas.

Ver también:

- **PEP 612** - Variables de especificación de parámetros (el PEP que introdujo `ParamSpec` y `Concatenate`).
- *`Callable`* y *`Concatenate`*.

typing.**ParamSpecArgs**

typing.**ParamSpecKwargs**

Argumentos y atributos de argumentos de palabras clave de un *`ParamSpec`*. El atributo `P.args` de un `ParamSpec` es una instancia de `ParamSpecArgs` y `P.kwargs` es una instancia de `ParamSpecKwargs`. Están pensados para la introspección en tiempo de ejecución y no tienen un significado especial para los verificadores de tipo estático.

Llamar a *`get_origin()`* en cualquiera de estos objetos retornará el `ParamSpec` original:

```
P = ParamSpec("P")
get_origin(P.args)   # returns P
get_origin(P.kwargs)   # returns P
```

Nuevo en la versión 3.10.

typing.**AnyStr**
AnyStr is a *constrained type variable* defined as 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
```

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()

func(C())  # Passes static type check
```

See **PEP 544** for more details. Protocol classes decorated with *runtime_checkable()* (described later) act as simple-minded runtime protocols that check only the presence of given attributes, ignoring their type signatures.

Las clases protocolo pueden ser genéricas, por ejemplo:

```
class GenProto(Protocol[T]):
    def meth(self) -> T:
        ...
```

Nuevo en la versión 3.8.

@typing.**runtime_checkable**
Marca una clase protocolo como aplicable en tiempo de ejecución (lo convierte en un *runtime protocol*).

Tal protocolo se puede usar con *isinstance()* y *issubclass()*. Esto lanzará una excepción *TypeError* cuando se aplique a una clase que no es un protocolo. Esto permite una comprobación estructural simple, muy semejante a «one trick ponies» en *collections.abc* con *Iterable*. Por ejemplo:

```
@runtime_checkable
class Closable(Protocol):
    def close(self): ...

assert isinstance(open('/some/file'), Closable)

@runtime_checkable
class Named(Protocol):
    name: str
```

(continué en la próxima página)

(proviene de la página anterior)

```
import threading
assert isinstance(threading.Thread(name='Bob'), Named)
```

Nota: `runtime_checkable()` will check only the presence of the required methods or attributes, not their type signatures or types. For example, *ssl.SSLObject* is a class, therefore it passes an *issubclass()* check against *Callable*. However, the `ssl.SSLObject.__init__` method exists only to raise a *TypeError* with a more informative message, therefore making it impossible to call (instantiate) *ssl.SSLObject*.

Nota: An *isinstance()* check against a runtime-checkable protocol can be surprisingly slow compared to an `isinstance()` check against a non-protocol class. Consider using alternative idioms such as *hasattr()* calls for structural checks in performance-sensitive code.

Nuevo en la versión 3.8.

Otras directivas especiales

Estos no son utilizados en anotaciones. Son utilizados como bloques para crear tipos genéricos.

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.

The resulting class has an extra attribute `__annotations__` giving a dict that maps the field names to the field types. (The field names are in the `_fields` attribute and the default values are in the `_field_defaults` attribute, both of which are part of the *namedtuple()* API.)

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*.

Distinto en la versión 3.8: Los atributos `_field_types` y `__annotations__` son simples diccionarios en vez de instancias de `OrderedDict`.

Distinto en la versión 3.9: Se remueve el atributo `_field_types` en favor del atributo más estándar `__annotations__` que tiene la misma información.

class typing.**NewType**(*name*, *tp*)

Una clase auxiliar para indicar un tipo diferenciado a un comprobador de tipos, consulte *NewType*. En tiempo de ejecución, retorna un objeto que retorna su argumento cuando se llama. Uso:

```
UserId = NewType('UserId', int)
first_user = UserId(1)
```

Nuevo en la versión 3.5.2.

Distinto en la versión 3.10: `NewType` es ahora una clase en lugar de una función.

class typing.**TypedDict**(*dict*)

Es una construcción especial para añadir indicadores de tipo a un diccionario. En tiempo de ejecución es un *dict* simple.

`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 aplicada 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')
```

To allow using this feature with older versions of Python that do not support **PEP 526**, `TypedDict` supports two additional equivalent syntactic forms:

```
Point2D = TypedDict('Point2D', x=int, y=int, label=str)
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': str})
```

The functional syntax should also be used when any of the keys are not valid identifiers, for example because they are keywords or contain hyphens. Example:

```
# raises SyntaxError
class Point2D(TypedDict):
    in: int  # 'in' is a keyword
    x-y: int  # name with hyphens

# OK, functional syntax
Point2D = TypedDict('Point2D', {'in': int, 'x-y': int})
```

De forma predeterminada, todas las claves deben estar presentes en un `TypedDict`. Es posible anular esto especificando la totalidad. Uso:

```
class Point2D(TypedDict, total=False):
    x: int
    y: int
```

Esto significa que un `Point2D TypedDict` puede tener cualquiera de las claves omitidas. Solo se espera que un verificador de tipo admita un `False` literal o `True` como valor del argumento `total`. `True` es el predeterminado y hace que todos los elementos definidos en el cuerpo de la clase sean obligatorios.

It is possible for a `TypedDict` type to inherit from one or more other `TypedDict` types using the class-based syntax. Usage:

```
class Point3D(Point2D):
    z: int
```

`Point3D` has three items: `x`, `y` and `z`. It is equivalent to this definition:

```
class Point3D(TypedDict):
    x: int
    y: int
    z: int
```

A `TypedDict` cannot inherit from a non-`TypedDict` class, notably including *Generic*. For example:

```
class X(TypedDict):
    x: int

class Y(TypedDict):
    y: int

class Z(object): pass  # A non-TypedDict class

class XY(X, Y): pass  # OK

class XZ(X, Z): pass  # raises TypeError

T = TypeVar('T')
class XT(X, Generic[T]): pass  # raises TypeError
```

A `TypedDict` can be introspected via annotations dicts (see annotations-howto for more information on annotations best practices), *__total__*, *__required_keys__*, and *__optional_keys__*.

__total__

`Point2D.__total__` gives the value of the `total` argument. Example:

```
>>> from typing import TypedDict
>>> class Point2D(TypedDict): pass
>>> Point2D.__total__
True
>>> class Point2D(TypedDict, total=False): pass
>>> Point2D.__total__
False
>>> class Point3D(Point2D): pass
>>> Point3D.__total__
True
```

__required_keys__

Nuevo en la versión 3.9.

__optional_keys__

`Point2D.__required_keys__` and `Point2D.__optional_keys__` return *frozenset* objects containing required and non-required keys, respectively. Currently the only way to declare both required and non-required keys in the same `TypedDict` is mixed inheritance, declaring a `TypedDict`

with one value for the `total` argument and then inheriting it from another `TypedDict` with a different value for `total`. Usage:

```
>>> class Point2D(TypedDict, total=False):
...     x: int
...     y: int
...
>>> class Point3D(Point2D):
...     z: int
...
>>> Point3D.__required_keys__ == frozenset({'z'})
True
>>> Point3D.__optional_keys__ == frozenset({'x', 'y'})
True
```

Nuevo en la versión 3.9.

Véase **PEP 589** para más ejemplos y reglas detalladas del uso de `TypedDict`.

Nuevo en la versión 3.8.

Colecciones genéricas concretas

Correspondientes a tipos integrados

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]:
    ...
```

Obsoleto desde la versión 3.9: *`builtins.dict`* now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

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]
```

Obsoleto desde la versión 3.9: *`builtins.list`* now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

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`*.

Obsoleto desde la versión 3.9: *`builtins.set`* now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

class `typing.`**FrozenSet**(*frozenset, AbstractSet[T_co]*)
Una versión genérica de *`builtins.frozenset`*.

Obsoleto desde la versión 3.9: `builtins.frozenset` now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

Nota: `Tuple` es una forma especial.

Correspondiente a tipos en `collections`

class typing.**DefaultDict**(*collections.defaultdict, MutableMapping[KT, VT]*)
Una versión genérica de `collections.defaultdict`.

Nuevo en la versión 3.5.2.

Obsoleto desde la versión 3.9: `collections.defaultdict` now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**OrderedDict**(*collections.OrderedDict, MutableMapping[KT, VT]*)
Una versión genérica de `collections.OrderedDict`.

Nuevo en la versión 3.7.2.

Obsoleto desde la versión 3.9: `collections.OrderedDict` now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

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.

Obsoleto desde la versión 3.9: `collections.ChainMap` now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

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.

Obsoleto desde la versión 3.9: `collections.Counter` now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

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.

Obsoleto desde la versión 3.9: `collections.deque` now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

Otros tipos concretos

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 los tipos de flujos de E/S como los retornados por *open()*.

Obsoleto desde la versión 3.8, se eliminará en la versión 3.13: El espacio de nombres `typing.io` está obsoleto y se eliminará. En su lugar, estos tipos deben importarse directamente desde `typing`.

class typing.**Pattern**
class typing.**Match**
Estos alias de tipo corresponden a los tipos retornados de *re.compile()* y *re.match()*. Estos tipos (y las funciones correspondientes) son genéricos en `AnyStr` y se pueden hacer específicos escribiendo `Pattern[str]`, `Pattern[bytes]`, `Match[str]` o `Match[bytes]`.

Obsoleto desde la versión 3.8, se eliminará en la versión 3.13: El espacio de nombres `typing.re` está obsoleto y se eliminará. En su lugar, estos tipos deben importarse directamente desde `typing`.

Obsoleto desde la versión 3.9: Las clases `Pattern` y `Match` de *re* ahora soportan `[]`. Véase **PEP 585** y *Tipo Alias Genérico*.

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.

Clase base abstracta para tipos genéricos

Correspondientes a las colecciones en `collections.abc`

class typing.**AbstractSet**(*Collection[T_co]*)
Una versión genérica de *collections.abc.Set*.

Obsoleto desde la versión 3.9: *collections.abc.Set* now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**ByteString**(*Sequence[int]*)
Una versión genérica de *collections.abc.ByteString*.

Este tipo representa a los tipos *bytes*, *bytearray*, y *memoryview* de secuencias de bytes.

Como abreviación para este tipo, *bytes* se puede usar para anotar argumentos de cualquiera de los tipos mencionados arriba.

Obsoleto desde la versión 3.9: *collections.abc.ByteString* now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**Collection**(*Sized, Iterable[T_co], Container[T_co]*)
Una versión genérica de *collections.abc.Collection*

Nuevo en la versión 3.6.0.

Obsoleto desde la versión 3.9: *collections.abc.Collection* now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**Container**(*Generic[T_co]*)
Una versión genérica de `collections.abc.Container`.

Obsoleto desde la versión 3.9: `collections.abc.Container` now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**ItemsView**(*MappingView, AbstractSet[tuple[KT_co, VT_co]]*)
Una versión genérica de `collections.abc.ItemsView`.

Obsoleto desde la versión 3.9: `collections.abc.ItemsView` now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**KeysView**(*MappingView, AbstractSet[KT_co]*)
Una versión genérica de `collections.abc.KeysView`.

Obsoleto desde la versión 3.9: `collections.abc.KeysView` now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**Mapping**(*Collection[KT], Generic[KT, 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]
```

Obsoleto desde la versión 3.9: `collections.abc.Mapping` now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**MappingView**(*Sized*)
Una versión genérica de `collections.abc.MappingView`.

Obsoleto desde la versión 3.9: `collections.abc.MappingView` now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**MutableMapping**(*Mapping[KT, VT]*)
Una versión genérica de `collections.abc.MutableMapping`.

Obsoleto desde la versión 3.9: `collections.abc.MutableMapping` now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**MutableSequence**(*Sequence[T]*)
Una versión genérica de `collections.abc.MutableSequence`.

Obsoleto desde la versión 3.9: `collections.abc.MutableSequence` now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**MutableSet**(*AbstractSet[T]*)
Una versión genérica de `collections.abc.MutableSet`.

Obsoleto desde la versión 3.9: `collections.abc.MutableSet` now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**Sequence**(*Reversible[T_co], Collection[T_co]*)
Una versión genérica de `collections.abc.Sequence`.

Obsoleto desde la versión 3.9: `collections.abc.Sequence` now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**ValuesView**(*MappingView, Collection[_VT_co]*)
Una versión genérica de `collections.abc.ValuesView`.

Obsoleto desde la versión 3.9: `collections.abc.ValuesView` now supports subscripting (`[]`). See **PEP 585** and *Tipo Alias Genérico*.

Correspondiente a otros tipos en `collections.abc`

class typing.**Iterable**(*Generic[T_co]*)
Una versión genérica de *collections.abc.Iterable*.

Obsoleto desde la versión 3.9: *collections.abc.Iterable* now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**Iterator**(*Iterable[T_co]*)
Una versión genérica de *collections.abc.Iterator*.

Obsoleto desde la versión 3.9: *collections.abc.Iterator* now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

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
```

Obsoleto desde la versión 3.9: *collections.abc.Generator* now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**Hashable**
An alias to *collections.abc.Hashable*.

class typing.**Reversible**(*Iterable[T_co]*)
Una versión genérica de *collections.abc.Reversible*.

Obsoleto desde la versión 3.9: *collections.abc.Reversible* now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**Sized**
An alias to *collections.abc.Sized*.

Programación asíncrona

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 collections.abc import Coroutine
c: Coroutine[list[str], str, int]  # Some coroutine defined elsewhere
x = c.send('hi')                   # Inferred type of 'x' is list[str]
async def bar() -> None:
    y = await c                    # Inferred type of 'y' is int
```

Nuevo en la versión 3.5.3.

Obsoleto desde la versión 3.9: *collections.abc.Coroutine* now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

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"ReturnType". 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.

Obsoleto desde la versión 3.9: *collections.abc.AsyncGenerator* now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**AsyncIterable**(*Generic[T_co]*)
Una versión genérica de *collections.abc.AsyncIterable*.

Nuevo en la versión 3.5.2.

Obsoleto desde la versión 3.9: *collections.abc.AsyncIterable* now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**AsyncIterator**(*AsyncIterable[T_co]*)
Una versión genérica de *collections.abc.AsyncIterator*.

Nuevo en la versión 3.5.2.

Obsoleto desde la versión 3.9: `collections.abc.AsyncIterator` now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

class typing.**Awaitable**(*Generic[T_co]*)
Una versión genérica de `collections.abc.Awaitable`.

Nuevo en la versión 3.5.2.

Obsoleto desde la versión 3.9: `collections.abc.Awaitable` now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

Tipos del administrador de contextos

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.

Obsoleto desde la versión 3.9: `contextlib.AbstractContextManager` now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

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.

Obsoleto desde la versión 3.9: `contextlib.AbstractAsyncContextManager` now supports subscripting ([]). See **PEP 585** and *Tipo Alias Genérico*.

Protocolos

Estos protocolos se decoran con `runtime_checkable()`.

class typing.**SupportsAbs**
Una ABC con un método abstracto `__abs__` que es covariante en su tipo retornado.

class typing.**SupportsBytes**
Una ABC con un método abstracto `__bytes__`.

class typing.**SupportsComplex**
Una ABC con un método abstracto `__complex__`.

class typing.**SupportsFloat**
Una ABC con un método abstracto `__float__`.

class typing.**SupportsIndex**
Una ABC con un método abstracto `__index__`.

Nuevo en la versión 3.8.

class typing.**SupportsInt**
Una ABC con un método abstracto `__int__`.

class typing.**SupportsRound**
Una ABC con un método abstracto `__round__` que es covariantes en su tipo retornado.

Funciones y decoradores

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.**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>
```

See **PEP 484** for more details and comparison with other typing semantics.

@typing.**final**
Un decorador que indica a los validadores de tipos que el método decorado no se puede sobreescribir, o que la clase decorada no se puede derivar (*subclassed*). 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 lo 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.

Ayudas de introspección

typing.**get_type_hints**(*obj*, *globalns=None*, *localns=None*, *include_extras=False*)
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.

La función reemplaza todos los `Annotated[T, ...]` con `T` de manera recursiva, a menos que `include_extras` se defina como `True` (véase *Annotated* para más información). Por ejemplo:

```
class Student(NamedTuple):
    name: Annotated[str, 'some marker']

get_type_hints(Student) == {'name': str}
get_type_hints(Student, include_extras=False) == {'name': str}
get_type_hints(Student, include_extras=True) == {
    'name': Annotated[str, 'some marker']
}
```

Nota: *get_type_hints()* no funciona con *alias de tipo* importados que incluyen referencias hacia adelante. Habilitar la evaluación pospuesta de anotaciones (**PEP 563**) puede eliminar la necesidad de la mayoría de las referencias futuras.

Distinto en la versión 3.9: Se agregan los parámetros `include_extras` como parte de **PEP 593**.

typing.**get_args**(*tp*)

typing.**get_origin**(*tp*)
Provee introspección básica para tipos genéricos y construcciones especiales de tipado.

Para un objeto de escritura de la forma `X[Y, Z, ...]`, estas funciones retornan `X` y `(Y, Z, ...)`. Si `X` es un alias genérico para una clase incorporada o *collections*, se normaliza a la clase original. Si `X` es una unión o *Literal* contenido en otro tipo genérico, el orden de `(Y, Z, ...)` puede ser diferente del orden de los argumentos originales `[Y, Z, ...]` debido al tipo de almacenamiento en caché. Para objetos no admitidos, retorna `None` y `()` correspondientemente. Ejemplos:

```
assert get_origin(Dict[str, int]) is dict
assert get_args(Dict[int, str]) == (int, str)
```

(continué en la próxima página)

(proviene de la página anterior)

```
assert get_origin(Union[int, str]) is Union
assert get_args(Union[int, str]) == (int, str)
```

Nuevo en la versión 3.8.

typing.**is_typeddict**(*tp*)
Compruebe si un tipo es *TypedDict*.

Por ejemplo:

```
class Film(TypedDict):
    title: str
    year: int

is_typeddict(Film)  # => True
is_typeddict(list | str)  # => False
```

Nuevo en la versión 3.10.

class typing.**ForwardRef**
Una clase utilizada para la representación de escritura interna de referencias de cadena hacia adelante. Por ejemplo, `List["SomeClass"]` se transforma implícitamente en `List[ForwardRef("SomeClass")]`. Esta clase no debe ser instanciada por un usuario, pero puede ser utilizada por herramientas de introspección.

Nota: Los tipos genéricos de **PEP 585**, como `list["SomeClass"]`, no se transformarán implícitamente en `list[ForwardRef("SomeClass")]` y, por lo tanto, no se resolverán automáticamente en `list[SomeClass]`.

Nuevo en la versión 3.7.4.

Constantes

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.

Nota: If `from __future__ import annotations` is used, annotations are not evaluated at function definition time. Instead, they are stored as strings in `__annotations__`. This makes it unnecessary to use quotes around the annotation (see **PEP 563**).

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 a partir de módulos de Python. La documentación puede presentarse como páginas de texto en la consola, enviarse a un navegador web o guardarse 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

```
python -m 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.

You can also use **pydoc** to start an HTTP server on the local machine that will serve documentation to visiting web browsers. **python -m pydoc -p 1234** will start a HTTP server on port 1234, allowing you to browse the documentation at `http://localhost:1234/` in your preferred web browser. Specifying `0` as the port number will select an arbitrary unused port.

python -m pydoc -n <hostname> will start the server listening at the given hostname. By default the hostname is "localhost" but if you want the server to be reached from other machines, you may want to change the host name that the server responds to. During development this is especially useful if you want to run pydoc from within a container.

python -m pydoc -b will start the server and additionally open a web browser to a module index page. Each served page has a navigation bar at the top where you can *Get* help on an individual item, *Search* all modules with a keyword in their synopsis line, and go to the *Module index*, *Topics* and *Keywords* pages.

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 del 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 `PYTHONDOCS` 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 Modo de desarrollo de Python

Nuevo en la versión 3.7.

El modo de desarrollo de Python introduce comprobaciones adicionales en tiempo de ejecución que son muy costosas para ser activadas por defecto. No debería ser más verboso que el predeterminado si el código es correcto; sólo se emiten nuevas advertencias cuando se detecta un problema.

Puede activarse mediante la opción de línea de comandos `-X dev` o estableciendo la variable de entorno `PYTHONDEVMODE` en `1`.

Consulte también Compilación de depuración de Python.

26.4 Efectos del modo de desarrollo de Python

Activar el modo de desarrollo de Python es similar al siguiente comando, pero con efectos adicionales que se describen a continuación:

```
PYTHONMALLOC=debug PYTHONASYNCIODEBUG=1 python3 -W default -X faulthandler
```

Efectos del modo de desarrollo de Python:

- Añadir `default` *filtro de avisos*.Se muestran las siguientes advertencias:
  - *DeprecationWarning*
  - *ImportWarning*
  - *PendingDeprecationWarning*
  - *ResourceWarning*

  Normalmente, los advertencias son filtradas por defecto *warning filters*.

  Se comporta como si se utilizara la opción de línea de comandos `-W default`.

  Utilice la opción de línea de comandos `-W error` o establezca la variable de entorno `PYTHONWARNINGS` en `error` para tratar las advertencias como errores.
- Instalar hooks(enganches) de depuración en los asignadores de memoria para comprobar:
  - Desbordamiento del búfer
  - Sobrecarga del búfer
  - Violación de la API del asignador de memoria
  - Uso inseguro del GIL

Ver la función en C `PyMem_SetupDebugHooks()`.

Se comporta como si la variable de entorno `PYTHONMALLOC` estuviera establecida en `debug`.

Para activar el modo de desarrollo de Python sin instalar ganchos de depuración en los asignadores de memoria, establezca la variable de entorno `PYTHONMALLOC` a `default`.

- Llama a *`faulthandler.enable()`* al inicio de Python para instalar los handlers(manejadores) de las señales `SIGSEGV`, `SIGFPE`, `SIGABRT`, `SIGBUS` y `SIGILL` para volcar la traza de Python en caso de fallo.

  Se comporta como si se utilizara la opción de línea de comandos `-X faulthandler` o si la variable de entorno `PYTHONFAULTHANDLER` se establece en `1`.

- Habilitar *asyncio debug mode*. Por ejemplo, *`asyncio`* comprueba las corutinas que no fueron esperadas y las registra.

  Se comporta como si la variable de entorno `PYTHONASYNCIODEBUG` estuviera establecida en `1`.

- Comprueba los argumentos *encoding* y *errors* para las operaciones de codificación y decodificación de cadenas. Ejemplos: *`open()`*, *`str.encode()`* y *`bytes.decode()`*.

  Por defecto, para un mejor rendimiento, el argumento *errors* sólo se comprueba en el primer error de codificación/decodificación y el argumento *encoding* a veces se ignora para las cadenas vacías.

- El destructor de *`io.IOBase`* registra las excepciones `close()`.

- Establece el atributo `dev_mode` de *`sys.flags`* a `True`.

El Modo de Desarrollo de Python no habilita el módulo *`tracemalloc`* por defecto, porque el costo de la sobrecarga (para el rendimiento y la memoria) sería demasiado grande. Activar el módulo *`tracemalloc`* proporciona información adicional sobre el origen de algunos errores. Por ejemplo, *`ResourceWarning`* registra la traza donde se asignó el recurso, y un error de desbordamiento de búfer registra la traza donde se asignó el bloque de memoria.

El modo de desarrollo de Python no impide que la opción de línea de comandos `-O` elimine las declaraciones `assert` ni que establezca *`__debug__`* a `False`.

El modo de desarrollo de Python solo se puede habilitar en el inicio de Python. Su valor se puede leer en *`sys.flags.dev_mode`*.

Distinto en la versión 3.8: El destructor de *`io.IOBase`* ahora registra las excepciones `close()`.

Distinto en la versión 3.9: Los argumentos *enconding* y *errors* se comprueban ahora para las operaciones de codificación y descodificación de cadenas.

26.5 Ejemplo de ResourceWarning

Ejemplo de un script que cuenta el número de líneas del archivo de texto especificado en la línea de comandos:

```
import sys

def main():
    fp = open(sys.argv[1])
    nlines = len(fp.readlines())
    print(nlines)
    # The file is closed implicitly

if __name__ == "__main__":
    main()
```

El script no cierra el archivo explícitamente. Por defecto, Python no emite ninguna advertencia. Ejemplo usando README.txt, que tiene 269 líneas:

```
$ python3 script.py README.txt
269
```

Al activar el modo de desarrollo de Python aparece una advertencia *ResourceWarning*:

```
$ python3 -X dev script.py README.txt
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst'␣
↪mode='r' encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
```

Además, al activar *tracemalloc* se muestra la línea en la que se abrió el archivo:

```
$ python3 -X dev -X tracemalloc=5 script.py README.rst
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst'␣
↪mode='r' encoding='UTF-8'>
  main()
Object allocated at (most recent call last):
  File "script.py", lineno 10
    main()
  File "script.py", lineno 4
    fp = open(sys.argv[1])
```

La solución es cerrar explícitamente el archivo. Ejemplo utilizando un gestor de contexto:

```
def main():
    # Close the file explicitly when exiting the with block
    with open(sys.argv[1]) as fp:
        nlines = len(fp.readlines())
    print(nlines)
```

No cerrar un recurso explícitamente puede dejar un recurso abierto durante mucho más tiempo del estimado; puede causar graves problemas al salir de Python. Es malo en CPython, pero es aún peor en PyPy. Cerrar los recursos explícitamente hace que una aplicación sea más detallista y más fiable.

26.6 Ejemplo de error de descriptor de archivo incorrecto

Script que muestra la primera línea de sí mismo:

```
import os

def main():
    fp = open(__file__)
    firstline = fp.readline()
    print(firstline.rstrip())
    os.close(fp.fileno())
    # The file is closed implicitly

main()
```

Por defecto, Python no emite ninguna advertencia:

```
$ python3 script.py
import os
```

El modo de desarrollo de Python muestra un *ResourceWarning* y registra un error «Bad file descriptor» cuando termina el objeto archivo:

```
$ python3 script.py
import os
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='script.py'␣
↪mode='r' encoding='UTF-8'>
```

(continué en la próxima página)

(proviene de la página anterior)

```
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
Exception ignored in: <_io.TextIOWrapper name='script.py' mode='r' encoding='UTF-8
↪'>
Traceback (most recent call last):
  File "script.py", line 10, in <module>
    main()
OSError: [Errno 9] Bad file descriptor
```

`os.close(fp.fileno())` cierra el descriptor de archivo. Cuando el finalizador de objetos de archivo intenta cerrar el descriptor de archivo de nuevo, falla con el error `Bad file descriptor`. Un descriptor de archivo debe cerrarse sólo una vez. En el peor de los casos, cerrarlo dos veces puede provocar un fallo (ver bpo-18748 para un ejemplo).

La solución es eliminar la línea `os.close(fp.fileno())`, o abrir el archivo con `closefd=False`.

26.7 `doctest` – Prueba ejemplos interactivos de Python

Código fuente: Lib/doctest.py

El módulo *doctest* busca pedazos de texto que lucen como sesiones interactivas de Python, y entonces ejecuta esas sesiones para verificar que funcionen exactamente como son mostradas. Hay varias maneras de usar doctest:

- Para revisar que los docstrings de un módulo están al día al verificar que todos los ejemplos interactivos todavía trabajan como está documentado.
- Para realizar pruebas de regresión al verificar que los ejemplos interactivos de un archivo de pruebas o un objeto de pruebas trabaje como sea esperado.
- Para escribir documentación de tutorial para un paquete, generosamente ilustrado con ejemplos de entrada-salida. Dependiendo si los ejemplos o el texto expositivo son enfatizados, tiene el sabor de «pruebas literarias» o «documentación ejecutable».

Aquí hay un módulo de ejemplo completo pero pequeño:

```
"""
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.

    >>> [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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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()
```

Si tu ejecutas `example.py` directamente desde la línea de comandos, *doctest* hará su magia:

```
$ python example.py
$
```

¡No hay salida! Eso es normal, y significa que todos los ejemplos funcionaron. Pasa `-v` al script, y *doctest* imprime un registro detallado de lo que está intentando, e imprime un resumen al final:

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

Y demás, eventualmente terminando con:

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
```

(continué en la próxima página)

(proviene de la página anterior)

```
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.
$
```

¡Eso es todo lo que necesitas saber para empezar a hacer uso productivo de *doctest*! Lánzate. Las siguientes secciones proporcionan detalles completos. Note que hay muchos ejemplos de doctests en el conjunto de pruebas estándar de Python y bibliotecas. Especialmente ejemplos útiles se pueden encontrar en el archivo de pruebas estándar `Lib/test/test_doctest.py`.

26.7.1 Uso simple: verificar ejemplos en docstrings

La forma más simple para empezar a usar doctest (pero no necesariamente la forma que continuarás usándolo) es terminar cada módulo `M` con:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

doctest entonces examina docstrings en el módulo `M`.

Ejecutar el módulo como un script causa que los ejemplos en los docstrings se ejecuten y verifiquen:

```
python M.py
```

No mostrará nada a menos que un ejemplo falle, en cuyo caso el ejemplo fallido o ejemplos fallidos y sus causas de la falla son imprimidas a stdout, y la línea final de salida es `***Test Failed*** N failures.`, donde *N* es el número de ejemplos que fallaron.

Ejecútalo con el modificador `-v` en su lugar:

```
python M.py -v
```

y un reporte detallado de todos los ejemplos intentados es impreso a la salida estándar, junto con resúmenes clasificados al final.

Puedes forzar el modo verboso al pasar `verbose=True` a *testmod()*, o prohibirlo al pasarlo `verbose=False`. En cualquiera de estas casos, `sys.argv` no es examinado por *testmod()* (por lo que pasar o no `-v`, no tiene efecto).

También hay un atajo de línea de comandos para ejecutar *testmod()*. Puedes instruir al intérprete de Python para ejecutar el módulo doctest directamente de la biblioteca estándar y pasar los nombres del módulo en la línea de comandos:

```
python -m doctest -v example.py
```

Esto importará `example.py` como un módulo independiente y ejecutará a *testmod()*. Note que esto puede no funcionar correctamente si el archivo es parte de un paquete e importa otros submódulos de ese paquete.

Para más información de *testmod()*, véase la sección *API básica*.

26.7.2 Uso Simple: Verificar ejemplos en un Archivo de Texto

Otra simple aplicación de doctest es probar ejemplos interactivos en un archivo de texto. Esto puede ser hecho con la función *testfile()*:

```
import doctest
doctest.testfile("example.txt")
```

Este script corto ejecuta y verifica cualquier ejemplo interactivo de Python contenido en el archivo `example.txt`. El contenido del archivo es tratado como si fuese un solo gran docstring; ¡el archivo no necesita contener un programa de Python! Por ejemplo, tal vez `example.txt` contenga esto:

```
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
```

Ejecutar `doctest.testfile("example.txt")` entonces encuentra el error en esta documentación:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

Como con *testmod()*, *testfile()* no mostrará nada a menos que un ejemplo falle. Si un ejemplo no falla, entonces los ejemplos fallidos y sus causas son impresas a stdout, usando el mismo formato como *testmod()*.

Por defecto, *testfile()* busca archivos en el directorio del módulo al que se llama. Véase la sección *API básica* para una descripción de los argumentos opcionales que pueden ser usados para decirle que busque archivos en otros lugares.

Como *testmod()*, la verbosidad de *testfile()* puede ser establecida con el modificador de la línea de comandos `-v` o con el argumento por palabra clave opcional *verbose*.

También hay un atajo de línea de comandos para ejecutar *testfile()*. Puedes indicar al intérprete de Python para correr el módulo doctest directamente desde la biblioteca estándar y pasar el nombre de los archivos en la línea de comandos:

```
python -m doctest -v example.txt
```

Porque el nombre del archivo no termina con `.py`, *doctest* infiere que se debe ejecutar con *testfile()*, no *testmod()*.

Para más información en *testfile()*, véase la sección *API básica*.

26.7.3 Cómo funciona

Esta sección examina en detalle cómo funciona doctest: qué docstrings revisa, cómo encuentra ejemplos interactivos, qué contexto de ejecución usa, cómo maneja las excepciones, y cómo las banderas pueden ser usadas para controlar su comportamiento. Esta es la información que necesitas saber para escribir ejemplos de doctest; para información sobre ejecutar doctest en estos ejemplos, véase las siguientes secciones.

¿Qué docstrings son examinados?

Se busca en el docstring del módulo, y todos los docstrings de las funciones, clases, y métodos. Los objetos importados en el módulo no se buscan.

Además, si `M.__test__` existe y «es verdaderos», debe ser un diccionario, y cada entrada mapea un nombre (cadena de caracteres) a un objeto de función, objeto de clase, o cadena de caracteres. Se buscan los docstrings de los objetos de función o de clase encontrados de `M.__test__`, y las cadenas de caracteres son tratadas como si fueran docstrings. En la salida, una clave `K` en `M.__test__` aparece con el nombre:

```
<name of M>.__test__.K
```

Todas las clases encontradas se buscan recursivamente de manera similar, para probar docstrings en sus métodos contenidos y clases anidadas.

¿Cómo se reconocen los ejemplos de docstring?

En la mayoría de los casos un copiar y pegar de una sesión de consola interactiva funciona bien, pero doctest no está intentando hacer una emulación exacta de ningún shell específico de Python.

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
...
no
NO
NO!!!
>>>
```

Cualquier salida esperada debe seguir inmediatamente el final de la línea `'>>>'` o `'...'` conteniendo el código, y la salida esperada (si la hubiera) se extiende hasta el siguiente `'>>>'` o la línea en blanco.

La letra pequeña:

- La salida esperada no puede contener una línea de espacios en blanco, ya que ese tipo de línea se toma para indicar el fin de la salida esperada. Si la salida esperada de verdad contiene una línea en blanco, pon `<BLANKLINE>` en tu ejemplo de doctest en cada lugar donde una línea en blanco sea esperada.
- Todos los caracteres de tabulación se expanden a espacios, usando paradas de tabulación de 8 -columnas. Las tabulaciones generadas por el código en pruebas no son modificadas. Ya que todas las tabulaciones en la salida de prueba *son* expandidas, significa que si el código de salida incluye tabulaciones, la única manera de que el doctest pueda pasar es si la opción *NORMALIZE_WHITESPACE* o *directive* está en efecto. Alternativamente, la prueba puede ser reescrita para capturar la salida y compararla a un valor esperado como parte de la prueba. Se llegó a este tratamiento de tabulaciones en la fuente a través de prueba y error, y ha demostrado ser la manera menos propensa a errores de manejarlos. Es posible usar un algoritmo diferente para manejar tabulaciones al escribir una clase *DocTestParser* personalizada.

- La salida a stdout es capturada, pero no la salida a stderr (los rastreos de la excepción son capturados a través de maneras diferentes).
- Si continuas una línea poniendo una barra invertida en una sesión interactiva, o por cualquier otra razón usas una barra invertida, debes usar un docstring crudo, que preservará tus barras invertidas exactamente como las escribes:

```
>>> def f(x):
...     r'''Backslashes in a raw docstring: m\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

  De otra manera, la barra invertida será interpretada como parte de una cadena. Por ejemplo, el `\n` arriba sería interpretado como un carácter de nueva línea. Alternativamente, puedes duplicar cada barra invertida en la versión de doctest (y no usar una cadena de caracteres cruda):

```
>>> def f(x):
...     '''Backslashes in a raw docstring: m\\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

- La columna inicial no importa:

```
>>> assert "Easy!"
      >>> import math
          >>> math.floor(1.9)
          1
```

  y tantos espacios en blanco al principio se eliminan de la salida esperada como aparece en la línea `'>>>'` inicial que empezó el ejemplo.

¿Cuál es el contexto de ejecución?

Por defecto, cada vez que un *doctest* encuentre un docstring para probar, usa una *copia superficial* de los globales de `M`, por lo que ejecutar pruebas no cambia los globales reales del módulo, y por lo que una prueba en `M` no puede dejar atrás migajas que permitan a otras pruebas trabajar. Significa que los ejemplos pueden usar libremente cualquier nombre definido en el nivel superior en `M`, y nombres definidos más temprano en los docstrings siendo ejecutados. Los ejemplos no pueden ver nombres definidos en otros docstrings.

Puedes forzar el uso de tus propios diccionarios como contexto de ejecución al pasar `globs=your_dict` a *testmod()* o *testfile()* en su lugar.

¿Y las excepciones?

No hay problema, siempre que el rastreo sea la única salida producida por el ejemplo: sólo copia el rastreo.[1] Ya que los rastreos contienen detalles que probablemente cambien rápidamente (por ejemplo, rutas de archivos exactas y números de línea), este es un caso donde doctest trabaja duro para ser flexible en lo que acepta.

Ejemplo simple:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

El doctest tiene éxito si se lanza *ValueError*, con el detalle `list.remove(x): x not in list` como se muestra.

[1] No se admiten los ejemplos que contienen una salida esperada y una excepción. Intentar adivinar dónde una termina y la otra empieza es muy propenso a errores, y da lugar a una prueba confusa.

La salida esperada para una excepción debe empezar con una cabecera de rastreo, que puede ser una de las siguientes dos líneas, con el mismo sangrado de la primera línea del ejemplo:

```
Traceback (most recent call last):
Traceback (innermost last):
```

La cabecera de rastreo es seguida por una pila de rastreo opcional, cuyo contenido es ignorado por doctest. La pila de rastreo es típicamente omitida, o copiada palabra por palabra de una sesión interactiva.

La pila de rastreo es seguida por la parte más interesante: la línea o líneas conteniendo el tipo de excepción y detalle. Esto es usualmente la última línea de un rastreo, pero se puede extender a través de múltiples líneas si la excepción tiene un detalle de varias líneas:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

Las últimas tres líneas (empezando con *ValueError*) son comparados con el tipo de excepción y detalle, y el resto es ignorado.

La mejor práctica es omitir la pila de rastreo, a menos que añada valor de documentación significante al ejemplo. Por lo que el último ejemplo es probablemente mejor como:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

Note que los rastreos son tratados de manera especial. En particular, en el ejemplo reescrito, el uso de `...` es independiente de la opción *ELLIPSIS* de doctest. Se pueden excluir los puntos suspensivos en ese ejemplo, así como también pueden haber tres (o trescientas) comas o dígitos, o una transcripción sangrada de un *sketch* de Monty Python.

Algunos detalles que debes leer una vez, pero no necesitarás recordar:

- Doctest no puede adivinar si tu salida esperada vino de una excepción de rastreo o de una impresión ordinaria. Así que, un ejemplo que espera `ValueError: 42 is prime` pasará, ya sea si de hecho se lance *ValueError* o si el ejemplo simplemente imprime ese texto de rastreo. En la práctica, la salida ordinaria raramente comienza con una línea de cabecera de rastreo, por lo que esto no crea problemas reales.
- Cada línea de la pila de rastreo (si se presenta) debe estar más sangrada que la primera línea del ejemplo, *o* empezar con un carácter no alfanumérico. la primera línea que sigue a la cabecera de rastreo sangrada de igual forma y empezando con un alfanumérico es considerado el inicio del detalle de la excepción. Por supuesto que esto es lo correcto para rastreos genuinos.
- Cuando se especifica la opción *IGNORE_EXCEPTION_DETAIL* de doctest. todo lo que sigue a los dos puntos más a la izquierda y cualquier otra información del módulo en el nombre de la excepción se ignora.
- El shell interactivo omite la línea de la cabecera de rastreo para algunos *SyntaxError*. Pero doctest usa la línea de la cabecera de rastreo para distinguir excepciones de los que no son. Así que en algunos casos raros donde necesitas probar un *SyntaxError* que omite la cabecera de rastreo, necesitarás poner manualmente la línea de cabecera de rastreo en tu ejemplo de prueba.
- Para algunos *SyntaxError*, Python muestra la posición del carácter del error de sintaxis, usando un marcador ^:

  ```
  >>> 1 1
    File "<stdin>", line 1
  ```

(continué en la próxima página)

(proviene de la página anterior)

```
    1 1
      ^
SyntaxError: invalid syntax
```

Ya que las líneas mostrando la posición del error vienen antes del tipo de excepción y detalle, no son revisadas por doctest. Por ejemplo, el siguiente test pasaría, a pesar de que pone el marcador ^ en la posición equivocada:

```
>>> 1 1
  File "<stdin>", line 1
    1 1
    ^
SyntaxError: invalid syntax
```

Banderas de Opción

Varias banderas de opción controlan diversos aspectos del comportamiento de doctest. Los nombres simbólicos para las banderas son proporcionados como constantes del módulo, que se pueden conectar mediante *OR* bit a bit y pasar a varias funciones. Los nombres también pueden ser usados en las *directivas de doctest*, y se pueden pasar a la interfaz de la línea de comandos de doctest a través de la opción `-o`.

Nuevo en la versión 3.4: La opción de la línea de comandos `-o`.

El primer grupo de opciones definen las semánticas de la prueba, controlando aspectos de cómo doctest decide si la salida de hecho concuerda con la salida esperada del ejemplo:

`doctest.`**`DONT_ACCEPT_TRUE_FOR_1`**
: Por defecto, si un bloque de salida esperada contiene sólo `1`, se considera igual a un bloque de salida real conteniendo sólo `1` o `true`, y similarmente para `0` contra `False`. Cuando se especifica `DONT_ACCEPT_TRUE_FOR_1`, no se permite ninguna sustitución. El comportamiento por defecto atiende a que Python cambió el tipo de retorno de muchas funciones de enteros a booleanos; los doctest esperando salidas «de pequeño enteros» todavía trabajan en estos casos. Esta opción probablemente se vaya, pero no por muchos años.

`doctest.`**`DONT_ACCEPT_BLANKLINE`**
: Por defecto, si un bloque de salida esperada contiene una línea que sólo tiene la cadena de caracteres `<BLANKLINE>`, entonces esa línea corresponderá a una línea en blanco en la salida real. Ya que una línea en blanca auténtica delimita la salida esperada, esta es la única manera de comunicar que una línea en blanco es esperada. Cuando se especifica *`DONT_ACCEPT_BLANKLINE`*, esta substitución no se permite.

`doctest.`**`NORMALIZE_WHITESPACE`**
: Cuando se especifica, todas las secuencias de espacios en blanco (vacías y nuevas líneas) son tratadas como iguales. Cualquier secuencia de espacios en blanco dentro de la salida esperada corresponderá a cualquier secuencia de espacios en blanco dentro de la salida real. Por defecto, los espacios en blanco deben corresponderse exactamente. *`NORMALIZE_WHITESPACE`* es especialmente útil cuando una línea de la salida esperada es muy larga, y quieres envolverla a través de múltiples líneas en tu código fuente.

`doctest.`**`ELLIPSIS`**
: Cuando se especifica, un marcador de puntos suspensivos (`...`) en la salida esperada puede corresponder a cualquier cadena de caracteres en la salida real. Esto incluye las cadenas de caracteres que abarcan límites de líneas, y cadenas de caracteres vacías, por lo que es mejor mantener su uso simple. Usos complicados pueden conducir a los mismo tipos de sorpresa de «ups, coincidió demasiado» que `.*` es propenso a hacer en expresiones regulares.

`doctest.`**`IGNORE_EXCEPTION_DETAIL`**
: When specified, doctests expecting exceptions pass so long as an exception of the expected type is raised, even if the details (message and fully qualified exception name) don't match.

  For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3*14`, but will fail if, say, a *`TypeError`* is raised instead. It will also ignore any fully qualified name included before the exception class, which can vary between implementations and versions of Python and the code/libraries in use. Hence, all three of these variations will work with the flag specified:

```
>>> raise Exception('message')
Traceback (most recent call last):
Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
builtins.Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
__main__.Exception: 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 the module name is present or matches exactly.

Distinto en la versión 3.2: *IGNORE_EXCEPTION_DETAIL* también ignora cualquier información relacionada al módulo conteniendo la excepción bajo prueba.

doctest.**SKIP**
Cuando se especifica, no ejecuta el ejemplo del todo. Puede ser útil en contextos donde los ejemplos de doctest sirven como documentación y casos de prueba a la vez, y un ejemplo debe ser incluido para propósitos de documentación, pero no debe ser revisado. P. ej., la salida del ejemplo puede ser aleatoria, o el ejemplo puede depender de recursos que no estarían disponibles para el controlador de pruebas.

La bandera *SKIP* también se puede usar para temporalmente «quitar» ejemplos.

doctest.**COMPARISON_FLAGS**
Una máscara de bits o juntadas lógicamente todas las banderas de arriba.

El segundo grupo de opciones controla cómo las fallas de las pruebas son reportadas:

doctest.**REPORT_UDIFF**
Cuando se especifica, las fallas que involucran salidas multilínea esperadas y reales son mostradas usando una diferencia (*diff*) unificada.

doctest.**REPORT_CDIFF**
Cuando se especifica, las fallas que involucran salidas multilínea esperadas y reales se mostrarán usando una diferencia (*diff*) contextual.

doctest.**REPORT_NDIFF**
Cuando se especifica, las diferencias son computadas por `difflib.Differ`, usando el mismo algoritmo que la popular utilidad `ndiff.py`. Este es el único método que marca diferencias dentro de líneas también como a través de líneas. Por ejemplo, si una línea de salida esperada contiene el dígito `1` donde la salida actual contiene la letra `l`, se inserta una línea con una marca de inserción marcando la posición de las columnas que no coinciden.

doctest.**REPORT_ONLY_FIRST_FAILURE**
Cuando se especifica, muestra el primer ejemplo fallido en cada doctest, pero suprime la salida para todos ejemplos restantes. Esto evitará que doctest reporte los ejemplos correctos que se rompen por causa de fallos tempranos; pero también puede esconder ejemplos incorrectos que fallen independientemente de la primera falla. Cuando se especifica *REPORT_ONLY_FIRST_FAILURE*, los ejemplos restantes aún se ejecutan, y aún cuentan para el número total de fallas reportadas, sólo se suprime la salida.

doctest.**FAIL_FAST**
Cuando se especifica, sale después del primer ejemplo fallido y no intenta ejecutar los ejemplos restantes. Por consiguiente, el número de fallas reportadas será como mucho 1. Esta bandera puede ser útil durante la depuración, ya que los ejemplos después de la primera falla ni siquiera producirán salida de depuración.

La línea de comandos de doctest acepta la opción `-f` como un atajo para `-o FAIL_FAST`.

Nuevo en la versión 3.4.

doctest.**REPORTING_FLAGS**
Una máscara de bits o todas las banderas de reporte arriba combinadas.

También hay una manera de registrar nombres de nuevas opciones de banderas, aunque esto no es útil a menos que intentes extender *doctest* a través de herencia:

doctest.**register_optionflag**(*name*)

Crea una nueva bandera de opción con un nombre dado, y retorna el valor entero de la nueva bandera. se puede usar *register_optionflag()* cuando se hereda *OutputChecker* o *DocTestRunner* para crear nuevas opciones que sean compatibles con tus clases heredadas. *register_optionflag()* siempre debe ser llamado usando la siguiente expresión:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

Directivas

Se pueden usar las directivas de doctest para modificar las *banderas de opción* para un ejemplo individual. Las directivas de doctest son comentarios de Python especiales que siguen el código fuente de un ejemplo:

```
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" | ...
```

No se permite el espacio en blanco entre el + o − y el nombre de la opción de directiva (*directive option name*). El nombre de la opción de directiva puede ser cualquiera de los nombres de las banderas de opciones explicadas arriba.

Las directivas de doctest de un ejemplo modifican el comportamiento de doctest para ese único ejemplo. Usa + para habilitar el comportamiento nombrado, o − para deshabilitarlo.

For example, this test passes:

```
>>> print(list(range(20)))  # doctest: +NORMALIZE_WHITESPACE
[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)))  # doctest: +ELLIPSIS
[0, 1, ..., 18, 19]
```

Multiple directives can be used on a single physical line, separated by commas:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

If multiple directive comments are used for a single example, then they are combined:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS
...                         # doctest: +NORMALIZE_WHITESPACE
[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)))
... # doctest: +ELLIPSIS
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

Tenga en cuenta que ya que todas las opciones están deshabilitadas por defecto, y las directivas sólo aplican a los ejemplos en los que aparecen, habilitarlas (a través de + en la directiva) usualmente es la única opción significativa.

Sin embargo, las banderas de opciones también pueden ser pasadas a funciones que ejecutan doctests, estableciendo valores por defecto diferentes. En tales casos, deshabilitar una opción a través de - en una directiva puede ser útil.

Advertencias

doctest es serio acerca de requerir coincidencias exactas en la salida esperada. Si incluso un solo carácter no coincide, el test falla. Esto probablemente te sorprenderá algunas veces, mientras aprendes exactamente lo que Python asegura y no asegura sobre la salida. Por ejemplo, cuando se imprime un conjunto, Python no asegura que el elemento sea impreso en ningún orden particular, por lo que una prueba como

```
>>> foo()
{"Hermione", "Harry"}
```

¡es vulnerable! Una solución puede ser hacer

```
>>> foo() == {"Hermione", "Harry"}
True
```

es su lugar. Otra es hacer

```
>>> d = sorted(foo())
>>> d
['Harry', 'Hermione']
```

Existen otros casos, pero ya captas la 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
<C object at 0x00AC18F0>
```

The *ELLIPSIS* directive gives a nice approach for the last example:

```
>>> C()  # doctest: +ELLIPSIS
<C object at 0x...>
```

Los números de coma flotante también son sujetos a pequeñas variaciones de la salida a través de las plataformas, porque Python defiere a la librería C de la plataforma para el formato de flotantes, y las librerías de C varían extensamente en calidad aquí.

```
>>> 1./7  # risky
0.14285714285714285
>>> print(1./7) # safer
0.142857142857
>>> print(round(1./7, 6)) # much safer
0.142857
```

Números de la forma `I/2.**J` son seguros a lo largo de todas las plataformas, y yo frecuentemente planeo ejemplos de doctest para producir números de esa forma:

```
>>> 3./4  # utterly safe
0.75
```

Las facciones simples también son más fáciles de entender para las personas, y eso conduce a una mejor documentación.

26.7.4 API básica

Las funciones `testmod()` y `testfile()` proporcionan una interfaz simple para doctest que debe ser suficiente para la mayoría de los usos básicos. Para una introducción menos formal a estas funciones, véase las secciones *Uso simple: verificar ejemplos en docstrings* y *Uso Simple: Verificar ejemplos en un Archivo de Texto*.

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*)

Todos los argumentos excepto *filename* son opcionales, y deben ser especificados en forma de palabras claves.

Prueba los ejemplos en el archivo con nombre *filename*. Retorna `(failure_count, test_count)`.

El argumento opcional *module_relative* especifica cómo el nombre de archivo debe ser interpretado:

- Si *module_relative* es `True` (el valor por defecto), entonces *filename* especifica una ruta relativa al módulo que es independiente del SO. Por defecto, esta ruta es relativa al directorio del módulo que lo invoca; pero si el argumento *package* es especificado, entonces es relativo a ese paquete. Para asegurar la independencia del SO, *filename* debe usar caracteres `/` para separar segmentos, y no puede ser una ruta absoluta (por ejemplo., no puede empezar con `/`).
- Si *module_relative* es `False`, entonces *filename* especifica una ruta especifica del SO. La ruta puede ser absoluta o relativa; las rutas relativas son resueltas con respecto al directorio de trabajo actual.

El argumento opcional *name* proporciona el nombre de la prueba; por defecto, o si es `None`, se usa `os.path.basename(filename)`.

El argumento opcional *package* es un paquete de Python o el nombre de una paquete de Python cuyo directorio debe ser usado como el directorio base para un nombre de archivo relativo al módulo. Si no se especifica ningún paquete, entonces el directorio del módulo que invoca se usa como el directorio base para los nombres de archivos relativos al módulo. Es un error especificar *package* si *module_relative* es `False`.

El argumento opcional *globs* proporciona un diccionario a ser usado como los globales cuando se ejecuten los ejemplos. Se crea una nueva copia superficial de este diccionario para el doctest, por lo que sus ejemplos empiezan con una pizarra en blanco. Por defecto, o si es `None`, se usa un nuevo diccionario vacío.

El argumento opcional *extraglobs* proporciona un diccionario mezclado con los globales usados para ejecutar ejemplos. Funciona como `dict.update()`: si *globs* y *extraglobs* tienen una clave en común, el valor asociado en *extraglobs* aparece en el diccionario combinado. Por defecto, o si es `None`, no se usa ninguna variable global. Es una característica avanzada que permite la parametrización de doctests. Por ejemplo, un doctest puede ser escribo para una clase base, usando un nombre genérico para la clase, y luego reusado para probar cualquier número de clases heredadas al pasar un diccionario de *extraglobs* mapeando el nombre genérico a la clase heredada para ser probada.

El argumento opcional *verbose* imprime un montón de cosas si es verdadero, e imprime sólo las fallas si es falso; por defecto, o si es `None`, es verdadero si y sólo si `'-v'` está en `sys.argv`.

El argumento opcional *report* imprime un resumen al final cuando es verdadero; si no, no imprime nada al final. En modo verboso (*verbose*), el resumen es detallado; si no, el resumen es muy corto (de hecho, vacío si todos las pruebas pasan).

El argumento opcional *optionflags* (valor por defecto 0) toma las banderas de opciones juntadas lógicamente por un OR. Véase la sección *Banderas de Opción*.

El argumento opcional *raise_on_error* tiene como valor por defecto *false*. Si es *true*, se lanza una excepción sobre la primera falla o excepción no esperada en un ejemplo. Esto permite que los fallos sean depurados en un análisis a posteriori. El comportamiento por defecto es continuar corriendo los ejemplos.

El argumento opcional *parser* especifica un `DocTestParser` (o subclase) que debe ser usado para extraer las pruebas de los archivos. Su valor por defecto es un analizador sintáctico normal (i.e., `DocTestParser()`).

El argumento opcional *encoding* especifica una codificación que debe ser usada para convertir el archivo a *unicode*.

doctest.**testmod**(*m=None*, *name=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *exclude_empty=False*)

Todos los argumentos son opcionales, y todos excepto por *m* deben ser especificados en forma de palabras claves.

Prueba los ejemplos en los docstring de las funciones y clases alcanzables desde el módulo *m* (o desde el módulo *__main__* si *m* no es proporcionado o es `None`), empezando con `m.__doc__`.

También prueba los ejemplos alcanzables desde el diccionario de `m.__test__`, si existe y no es `None`. `m.__test__` mapea los nombres (cadenas de caracteres) a funciones, clases y cadenas de caracteres; se buscan los ejemplos de las funciones y clases; se buscan las cadenas de caracteres directamente como si fueran docstrings.

Sólo se buscan los docstrings anexados a los objetos pertenecientes al módulo *m*.

Retorna `(failure_count, test_count)`.

El argumento opcional *name* proporciona el nombre del módulo; por defecto, o si es `None`, se usa `m.__name__`.

El argumento opcional *exclude_empty* es por defecto *false*. Si es verdadero, se excluyen los objetos por los cuales no se encuentren doctest. El valor por defecto es un *hack* de compatibilidad hacia atrás, por lo que el código que use `doctest.master.summarize()` en conjunto con *testmod()* continua obteniendo la salida para objetos sin pruebas. El argumento *exclude_empty* para el más nuevo constructor *DocTestFinder* es por defecto verdadero.

Los argumentos opcionales *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error*, y *globs* son los mismos en cuanto a la función *testfile()* arriba, excepto que *globs* es por defecto `m.__dict__`.

doctest.**run_docstring_examples**(*f*, *globs*, *verbose=False*, *name='NoName'*, *compileflags=None*, *optionflags=0*)

Prueba los ejemplos asociados con el objeto *f*; por ejemplo, *f* puede ser una cadena de caracteres, un módulo, una función, o un objeto clase.

Una copia superficial del diccionario del argumento *globs* se usa para la ejecución del contexto.

Se usa el argumento opcional *name* en mensajes de fallos, y por defecto es `"NoName"`.

Si el argumento opcional *verbose* es verdadero, la salida se genera incluso si no hay fallas. Por defecto, la salida se genera sólo en caso de la falla de un ejemplo.

El argumento opcional *compileflags* proporciona el conjunto de banderas que se deben usar por el compilador de Python cuando se corran los ejemplos. Por defecto, o si es `None`, las banderas se deducen correspondiendo al conjunto de características futuras encontradas en *globs*.

El argumento opcional *optionflags* trabaja con respecto a la función *testfile()* de arriba.

26.7.5 API de unittest

Mientras crece tu colección de módulos probados con doctest, vas a querer una forma de ejecutar todos sus doctests sistemáticamente. *doctest* proporciona dos funciones que se pueden usar para crear un banco de pruebas (*test suite*) de *unittest* desde módulos y archivos de texto que contienen doctests. Para integrarse con el descubrimiento de pruebas de *unittest* , incluye una función `load_tests()` en tu módulo de pruebas:

```
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
```

Hay dos funciones principales para crear instancias de *unittest.TestSuite* desde los archivos de texto y módulos con doctests:

doctest.**DocFileSuite**(**paths*, *module_relative=True*, *package=None*, *setUp=None*, *tearDown=None*, *globs=None*, *optionflags=0*, *parser=DocTestParser()*, *encoding=None*)

Convierte las pruebas de doctest de uno o más archivos de texto a una *unittest.TestSuite*.

El *unittest.TestSuite* que se retorne será ejecutado por el framework de unittest y ejecuta los ejemplos interactivos en cada archivo. Si un ejemplo en cualquier archivo falla, entonces la prueba unitaria sintetizada falla, y una excepción `failureException` se lanza mostrando el nombre del archivo conteniendo la prueba y un número de línea (algunas veces aproximado).

Pasa una o más rutas (como cadenas de caracteres) a archivos de texto para ser examinados.

Se pueden proporcionar las opciones como argumentos por palabra clave:

El argumento opcional *module_relative* especifica cómo los nombres de archivos en *paths* se deben interpretar:

- Si *module_relative* is `True` (el valor por defecto), entonces cada archivo de nombre en *paths* especifica una ruta relativa al módulo independiente del SO. Por defecto, esta ruta es relativa al directorio del módulo lo está invocando; pero si se especifica el argumento *package*, entonces es relativo a ese paquete. Para asegurar la independencia del SO, cada nombre de archivo debe usar caracteres `/` para separar los segmentos de rutas, y puede no ser una ruta absoluta (i.e., puede no empezar con `/`).
- Si *module_relative* es `False`, entonces cada archivo de nombre en *paths* especifica una ruta especifica al SO. La ruta puede ser absoluta o relativa; las rutas relativas son resueltas con respecto a directorio de trabajo actual.

El argumento opcional *package* es un paquete de Python o el nombre de un paquete de Python cuyo directorio debe ser usado como el directorio base para los nombres de archivos relativos al módulo en *paths*. Si no se especifica ningún paquete, entonces el directorio del módulo que lo está invocando se usa como el directorio base para los archivos de nombres relativos al módulo. Es un error especificar *package* si *module_relative* es `False`.

El argumento opcional *setUp* especifica una función de configuración para el banco de pruebas. Es invocado antes de ejecutar las pruebas en cada archivo. La función *setUp* se pasará a un objeto *DocTest*. La función *setUp* puede acceder a las variables globales de prueba como el atributo *globs* de la prueba pasada.

El argumento opcional *tearDown* especifica una función de destrucción para el banco de pruebas. Es invocado después de ejecutar las pruebas en cada archivo. Se pasará un objeto *DocTest* a la función *tearDown*. La función *setUp* de configuración puede acceder a los globales de la prueba como el atributo *globs* de la prueba pasada.

El argumento opcional *globs* es un diccionario que contiene las variables globales iniciales para las pruebas. Se crea una nueva copia de este diccionario para cada prueba. Por defecto, *globs* es un nuevo diccionario vacío.

El argumento opcional *optionflags* especifica las opciones de doctest por defecto para las pruebas, creado al juntar lógicamente las opciones de bandera individuales. Véase la sección *Banderas de Opción*. Véase la función *set_unittest_reportflags()* abajo para una mejor manera de definir las opciones de informe.

El argumento opcional *parser* especifica un *DocTestParser* (o subclase) que debe ser usado para extraer las pruebas de los archivos. Su valor por defecto es un analizador sintáctico normal (i.e., `DocTestParser()`).

El argumento opcional *encoding* especifica una codificación que debe ser usada para convertir el archivo a *unicode*.

Se añade el global `__file__` a los globales proporcionados a los doctests cargados desde un archivo de texto usando *DocFileSuite()*.

doctest.**DocTestSuite**(*module=None*, *globs=None*, *extraglobs=None*, *test_finder=None*, *setUp=None*, *tearDown=None*, *checker=None*)

Convierte las pruebas de doctest para un módulo a un *unittest.TestSuite*.

El *unittest.TestSuite* que se retorne será ejecutado por el framework de unittest y corre cada doctest en el módulo. Si cualquiera de los doctests falla, entonces la prueba unitaria combinada falla, y se lanza una excepción `failureException` mostrando el nombre del archivo que contiene la prueba y un número de línea (a veces aproximado).

El argumento opcional *module* proporciona el módulo a probar. Puede ser un objeto de módulo o un nombre (posiblemente punteado) de módulo. Si no se especifica, se usa el módulo que invoca esta función.

El argumento opcional *globs* es un diccionario que contiene las variables globales iniciales para las pruebas. Se crea una nueva copia de este diccionario para cada prueba. Por defecto, *globs* es un nuevo diccionario vacío.

El argumento opcional *extraglobs* especifica un conjunto de variables globales adicionales que son mezcladas con *globs*. Por defecto, no se usa ningún global adicional.

El argumento opcional *test_finder* es el objeto *DocTestFinder* (o un reemplazo directo) que se usa para extraer doctests desde el módulo.

Los argumentos opcionales *setUp*, *tearDown*, y *optionflags* son lo mismo con respecto a la función *DocFileSuite()* arriba.

Esta función usa la misma técnica de búsqueda que *testmod()*.

Distinto en la versión 3.5: *DocTestSuite()* retorna un *unittest.TestSuite* vacío si *module* no contiene ningún docstring en vez de lanzar un *ValueError*.

Por detrás, *DocTestSuite()* crea un *unittest.TestSuite* de las instancias de `doctest.DocTestCase`, y `DocTestCase` es una subclase de *unittest.TestCase*. `DocTestCase` no está documentado aquí (es un detalle interno), pero estudiar su código puede responder preguntas sobre los detalles exactos de la integración de *unittest*.

De manera similar, *DocFileSuite()* crea un *unittest.TestSuite* de las instancias de `doctest.DocFileCase`, y `DocFileCase` es una subclase de `DocTestCase`.

Por lo que ambas formas de crear un *unittest.TestSuite* ejecutan instancias de `DocTestCase`. Esto es importante por una razón sutil: cuando ejecutas las funciones de *doctest* por ti mismo, puedes controlar las opciones de *doctest* en uso directamente, al pasar las banderas de opciones a las funciones de *doctest*. Sin embargo, si estás escribiendo un framework de *unittest*, básicamente *unittest* controla cuándo y cómo se ejecutan las pruebas. El autor del framework típicamente, quiere controlar las opciones de reporte de *doctest* (quizás, p.ej., especificadas por las opciones de la línea de comandos), pero no hay forma de pasar opciones a través de *unittest* al probador de ejecución (*test runner*) de *doctest*.

Por esta razón, *doctest* también admite una noción de banderas de informe de *doctest* específicas para la compatibilidad con *unittest*, a través de esta función:

`doctest.`**`set_unittest_reportflags`**(*flags*)

Establece las banderas de informe de *doctest* a usar.

El argumento *flags* toma la combinación por el operador OR de las banderas de opciones. Véase la sección *Banderas de Opción*. Sólo se pueden usar las «banderas de informe».

Esta es una configuración global del módulo, y afecta a todos los doctests futuros a ejecutar por *unittest*: el método `runTest()` de `DocTestCase` revisa las banderas de opciones especificadas para el caso de prueba cuando la instancia de `DocTestCase` fue construida. Si no se especificó ninguna bandera de informe (que es el caso típico y esperado), las banderas de informe -pertenecientes a *doctest*- de *unittest* son combinadas por la operación Or en las banderas de opciones, y las banderas de opciones aumentadas se pasan a la instancia de *DocTestRunner* creada para ejecutar los doctest. Si se especificó alguna bandera de informe cuando el `DocTestCase` fue construido, se ignoran las banderas de informe -pertenecientes a *doctest*- de *unittest*.

La función retorna el valor de las banderas de informe de *unittest* en efecto antes de que la función fuera invocada.

26.7.6 API avanzada

La API básica es un simple envoltorio que sirve para hacer los doctest fáciles de usar. Es bastante flexible, y debe cumplir las necesidades de la mayoría de los usuarios; si requieres un control más preciso en las pruebas, o deseas extender las capacidades de doctest, entonces debes usar la API avanzada.

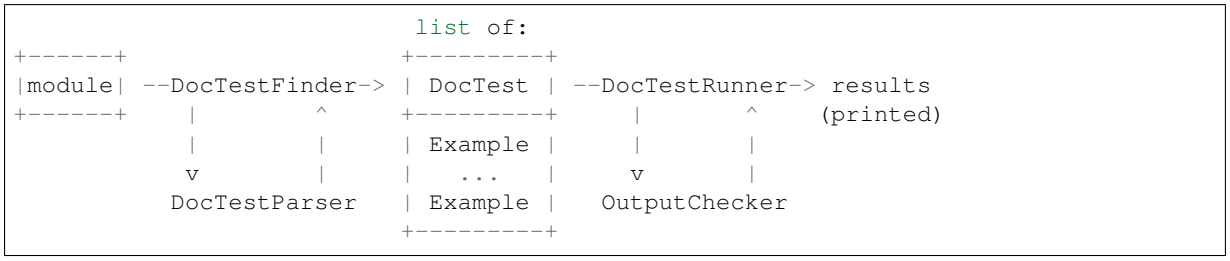
La API avanzada gira en torno a dos clases contenedoras, que se usan para guardar los ejemplos interactivos extraídos de los casos doctest:

- `Example`: Un *statement* de Python, emparejado con su salida esperada.
- `DocTest`: Una colección de clases `Example`, típicamente extraídos de un sólo docstring o archivo de texto.

Se definen clases de procesamiento adicionales para encontrar, analizar sintácticamente, y ejecutar, y comprobar ejemplos de doctest:

- `DocTestFinder`: Encuentra todos los docstrings en un módulo dado, y usa un `DocTestParser` para crear un `DocTest` de cada docstring que contiene ejemplos interactivos.
- `DocTestParser`: Crea un objeto `DocTest` de una cadena de caracteres (tal como un docstring de un objeto).
- `DocTestRunner`: Ejecuta los ejemplos en un `DocTest`, y usa un `OutputChecker` para verificar su salida.
- `OutputChecker`: Compara la salida real de un ejemplo de doctest con la salida esperada, y decide si coinciden.

Las relaciones entre estas clases de procesamiento se resumen en el siguiente diagrama:



Objetos DocTest

class doctest.**DocTest**(*examples*, *globs*, *name*, *filename*, *lineno*, *docstring*)

Una colección de ejemplos de doctest que deben ejecutarse en un sólo nombre de espacios. Se usan los argumentos del constructor para inicializar los atributos de los mismos nombres.

`DocTest` define los siguientes atributos. Son inicializados por el constructor, y no deben ser modificados directamente.

examples

Una lista de objetos `Example` codificando los ejemplos interactivos de Python individuales que esta prueba debe ejecutar.

globs

El nombre de espacios (alias *globals*) en que los ejemplos se deben ejecutar. Este es un diccionario que mapea nombres a valores. Cualquier cambio al nombre de espacios hecho por los ejemplos (tal como juntar nuevas variables) se reflejará en `globs` después de que se ejecute la prueba.

name

Un nombre de cadena de caracteres que identifica el `DocTest`. Normalmente, este es el nombre del objeto o archivo del que se extrajo la prueba.

filename

El nombre del archivo del que se extrajo este `DocTest`; o `None` si el nombre del archivo se desconoce, o si `DocTest` no se extrajo de un archivo.

lineno
El número de línea dentro de *filename* donde este *DocTest* comienza, o None si el número de línea no está disponible. Este número de línea es comienza en 0 con respecto al comienzo del archivo.

docstring
La cadena de caracteres del que se extrajo la cadena, o None si la cadena no está disponible, o si la prueba no se extrajo de una cadena de caracteres.

Objetos *Example*

class doctest.**Example**(*source*, *want*, *exc_msg=None*, *lineno=0*, *indent=0*, *options=None*)
Un sólo ejemplo interactivo, que consta de una sentencia de Python y su salida esperada. Los argumentos del constructor se usan para inicializar los atributos del mismo nombre.

La clase *Example* define los siguientes atributos. Son inicializados por el constructor, y no deben ser modificados directamente.

source
Una cadena de caracteres que contiene el código fuente del ejemplo. Este código fuente consiste de una sola sentencia Python, y siempre termina en una nueva línea; el constructor añade una nueva línea cuando sea necesario.

want
La salida esperada de ejecutar el código fuente del ejemplo (o desde la salida estándar, o un seguimiento en caso de una excepción). wants termina con una nueva línea a menos que no se espera ninguna salida, en cuyo caso es una cadena vacía. El constructor añade una nueva línea cuando sea necesario.

exc_msg
El mensaje de excepción que el ejemplo genera, si se espera que el ejemplo genere una excepción; o None si no se espera que genere una excepción. Se compara este mensaje de excepción con el valor de retorno de *traceback.format_exception_only()*. *exc_msg* termina con una nueva línea a menos que sea None. El constructor añade una nueva línea si se necesita.

lineno
El número de línea dentro de la cadena de caracteres que contiene este ejemplo donde el ejemplo comienza. Este número de línea comienza en 0 con respecto al comienzo de la cadena que lo contiene.

indent
La sangría del ejemplo en la cadena que lo contiene; i.e., el número de caracteres de espacio que preceden la primera entrada del ejemplo.

options
Un diccionario que mapea de las banderas de opciones a True o False, que se usa para anular las opciones por defecto para este ejemplo. Cualquier bandera de opción que no contiene este diccionario se deja con su valor por defecto (como se especifica por los optionflags de *DocTestRunner*). Por defecto, no se establece ninguna opción.

Objetos *DocTestFinder*

class doctest.**DocTestFinder**(*verbose=False*, *parser=DocTestParser()*, *recurse=True*, *exclude_empty=True*)
Una clase de procesamiento que se usa para extraer los *DocTest* que son relevantes para un objeto dado, desde su docstring y los docstring de sus objetos contenidos. Se puede extraer los *DocTest* de los módulos, clases, funciones, métodos, métodos estáticos, métodos de clase, y propiedades.

Se puede usar el argumento opcional *verbose* para mostrar los objetos buscados por *finder*. Su valor por defecto es False (ninguna salida).

El argumento opcional *parser* especifica el objeto *DocTestParser* (o un reemplazo directo) que se usa para extraer doctests desde docstrings.

Si el argumento opcional *recurse* es falso, entonces el método `DocTestFinder.find()` sólo examinará el objeto dado, y no cualquier objeto contenido.

Si el argumento opcional *exclude_empty* es falso, entonces `DocTestFinder.find()` incluirá pruebas para objetos con docstrings vacíos.

`DocTestFinder` define los siguientes métodos:

find(*obj[, name][, module][, globs][, extraglobs]*)
Retorna una lista de los `Doctest` que se definen por el docstring de *obj*, o por cualquiera de los docstring de sus objetos contenidos.

El argumento opcional *name* especifica el nombre del objeto; este nombre será usado para construir los nombres de los `DocTest` retornados. Si *name* no se especifica, entonces se usa `obj.__name__`.

El parámetro opcional *module* es el módulo que contiene el objeto dado. Si no se especifica el módulo o si es `None`, entonces el buscador de pruebas tratará de determinar automáticamente el módulo correcto. Se usa el módulo del objeto:

- Como un espacio de nombres por defecto, si no se especifica *globs*.
- Para evitar que *DocTestFinder* extraiga DocTests desde objetos que se importan desde otros módulos. (Se ignoran objetos contenidos con módulos aparte de *module*.)
- Para encontrar el nombre del archivo conteniendo el objeto.
- Para ayudar a encontrar el número de línea del objeto dentro de su archivo.

Si *module* es falso, no se hará ningún intento de encontrar el módulo. Es poco claro, de uso mayormente para probar doctest en si mismo: si *module* es `False`, o es `None` pero no se puede encontrar automáticamente, entonces todos los objetos se consideran que pertenecen al módulo (inexistente), por lo que todos los objetos contenidos se buscarán (recursivamente) por doctests.

Los globales para cada `DocTest` se forma al combinar *globs* y *extraglobs* (los enlaces en *extraglobs* anulan los enlaces en *globs*). Se crea una nueva copia superficial del diccionario de globales para cada `DocTest`. Si *globs* no se especifica, entonces su valor por defecto es el *__dict__* del módulo, si se especifica, o es `{}` de lo contrario, si *extraglobs* no se especifica, entonces su valor por defecto es `{}`.

Objetos *DocTestParser*

class doctest.**DocTestParser**
Un clase de procesamiento usada para extraer ejemplos interactivos de una cadena de caracteres, y usarlos para crear un objeto `DocTest`.

`DocTestParser` define los siguientes métodos:

get_doctest(*string*, *globs*, *name*, *filename*, *lineno*)
Extrae todos los ejemplos de *doctest* de una cadena dada, y los recolecta en un objeto `DocTest`.

globs, *name*, *filename*, y *lineno* son atributos para el nuevo objeto `DocTest`. Véase la documentación de `DocTest` para más información.

get_examples(*string*, *name='<string>'*)
Extrae todos los ejemplos de la cadena de caracteres dada, y los retorna como una lista de objetos `Example`. Los números de línea empiezan en 0. El argumento opcional *name* es una nombre identificando esta cadena, y sólo es usada para mensajes de errores.

parse(*string*, *name='<string>'*)
Divide el string dado en ejemplos y texto intermedio, y los retorna como una lista que alterna entre objetos `Example` y cadenas de caracteres. Los números de línea para los objetos `Example` empiezan en 0. El argumento opcional *name* es un nombre identificando esta cadena, y sólo se usa en mensajes de error.

Objetos *DocTestRunner*

class doctest.**DocTestRunner**(*checker=None*, *verbose=None*, *optionflags=0*)
Una clase de procesamiento usada para ejecutar y verificar los ejemplos interactivos en un `DocTest`.

La comparación entre salidas esperadas y salidas reales se hace por un `OutputChecker`. Esta comparación puede ser personalizada con un número de banderas de opción; véase la sección *Banderas de Opción* para más información. Si las banderas de opción son insuficientes, entonces la comparación también puede ser personalizada al pasar una subclase de `OutputChecker` al constructor.

La salida de la pantalla del *test runner* se puede controlar de dos maneras. Primero, se puede pasar una función de salida a `TestRunner.run()`; esta función se invocará con cadenas que deben mostrarse. Su valor por defecto es `sys.stdout.write`. Si no es suficiente capturar el resultado, entonces la salida de la pantalla también se puede personalizar al heredar de DocTestRunner, y sobreescribir los métodos `report_start()`, `report_success()`, `report_unexpected_exception()`, y `report_failure()`.

El argumento por palabra clave opcional *checker* especifica el objeto `OutputChecker` (o un reemplazo directo) que se debe usar para comparar las salidas esperadas con las salidas reales de los ejemplos de doctest.

El argumento por palabra clave opcional *verbose* controla la verbosidad de `DocTestRunner`. Si *verbose* es `True`, entonces la información de cada ejemplo se imprime , mientras se ejecuta. Si *verbose* es `False`, entonces sólo las fallas se imprimen. Si *verbose* no se especifica, o es `None`, entonces la salida verbosa se usa si y sólo se usa el modificador de la línea de comandos"-v".

El argumento por palabra clave opcional *optionflags* se puede usar para controlar cómo el *test runner* compara la salida esperada con una salida real, y cómo muestra las fallas. Para más información, véase la sección *Banderas de Opción*.

`DocTestParser` define los siguientes métodos:

report_start(*out*, *test*, *example*)
Notifica que el *test runner* está a punto de procesar el ejemplo dado. Este método es proporcionado para permitir que clases heredadas de `DocTestRunner` personalicen su salida; no debe ser invocado directamente.

example es el ejemplo a punto de ser procesado. *test* es la prueba que contiene a *example*. *out* es la función de salida que se pasó a `DocTestRunner.run()`.

report_success(*out*, *test*, *example*, *got*)
Notifica que el ejemplo dado se ejecutó correctamente. Este método es proporcionado para permitir que las clases heredadas de `DocTestRunner` personalicen su salida; no debe ser invocado directamente.

example es el ejemplo a punto de ser procesado. *got* es la salida real del ejemplo. *test* es la prueba conteniendo *example*. *out* es la función de salida que se pasa a `DocTestRunner.run()`.

report_failure(*out*, *test*, *example*, *got*)
Notifica que el ejemplo dado falló. Este método es proporcionado para permitir que clases heredadas de `DocTestRunner` personalicen su salida; no debe ser invocado directamente.

example es el ejemplo a punto de ser procesado. *got* es la salida real del ejemplo. *test* es la prueba conteniendo *example*. *out* es la función de salida que se pasa a `DocTestRunner.run()`.

report_unexpected_exception(*out*, *test*, *example*, *exc_info*)
Notifica que el ejemplo dado lanzó una excepción inesperada. Este método es proporcionado para permitir que las clases heredadas de `DocTestRunner` personalicen su salida; no debe ser invocado directamente.

example es el ejemplo a punto de ser procesado, *exc_info* es una tupla que contiene información sobre la excepción inesperada (como se retorna por `sys.exc_info()`). *test* es la prueba conteniendo *example*. *out* es la función de salida que debe ser pasada a `DocTestRunner.run()`.

run(*test*, *compileflags=None*, *out=None*, *clear_globs=True*)
Ejecuta los ejemplos en *test* (un objeto `DocTest`), y muestra los resultados usando función de escritura *out*.

Los ejemplo se ejecutan en el espacio de nombres `test.globs`. Si *clear_globs* es verdadero (el valor por defecto), entonces este espacio de nombres será limpiado después de la prueba se ejecute, para ayudar con la colección de basura. Si quisieras examinar el espacio de nombres después de que la prueba se complete, entonces use *clear_globs=False*.

compileflags da el conjunto de banderas que se deben usar por el compilador de Python cuando se ejecutan los ejemplos. Si no se especifica, entonces su valor por defecto será el conjunto de banderas de *future-import* que aplican a *globs*.

La salida de cada ejemplo es revisada usando el *output checker* del *DocTestRunner*, y los resultados se formatean por los métodos de `DocTestRunner.report_*()`.

summarize(*verbose=None*)
Imprime un resumen de todos los casos de prueba que han sido ejecutados por este *DocTestRunner*, y retorna un *named tuple* `TestResults(failed, attempted)`.

El argumento opcional *verbose* controla qué tan detallado es el resumen. Si no se especifica la verbosidad, entonces se usa la verbosidad de *DocTestRunner*.

Objetos *OutputChecker*

class doctest.**OutputChecker**
Una clase que se usa para verificar si la salida real de un ejemplo de doctest coincide con la salida esperada. *OutputChecker* define dos métodos: *check_output()*, que compara un par de salidas dadas, y retorna `True` si coinciden; y *output_difference()*, que retorna una cadena que describe las diferencias entre las dos salidas.

OutputChecker define los siguientes métodos:

check_output(*want*, *got*, *optionflags*)
Retorna `True` si y sólo si la salida real de un ejemplo (*got*) coincide con la salida esperada (*want*). Siempre se considera que estas cadenas coinciden si son idénticas; pero dependiendo de qué banderas de opción el *test runner* esté usando, varias coincidencias inexactas son posibles. Véase la sección *Banderas de Opción* para más información sobre las banderas de opción.

output_difference(*example*, *got*, *optionflags*)
Retorna una cadena que describe las diferencias entre la salida esperada para un ejemplo dado (*example*) y la salida real (*got*). *optionflags* es el conjunto de banderas de opción usado para comparar *want* y *got*.

26.7.7 Depuración

Doctest proporciona varios mecanismos para depurar los ejemplos de doctest:

- Varias funciones convierten los doctest en programas de Python ejecutables, que pueden ser ejecutadas bajo el depurador de Python, *pdb*.
- La clase *DebugRunner* es una subclase de *DocTestRunner* que lanza una excepción por el primer ejemplo fallido, conteniendo información sobre ese ejemplo. Esta información se puede usar para realizar depuración a posteriori en el ejemplo.
- Los casos de *unittest* generados por *DocTestSuite()* admiten el método *debug()* definido por *unittest.TestCase*.
- Puedes añadir una llamada a *pdb.set_trace()* en un ejemplo de doctest, y bajarás al depurador de Python cuando esa línea sea ejecutada. Entonces puedes inspeccionar los valores de las variables, y demás. Por ejemplo, supongamos que `a.py` contiene sólo este docstring de módulo:

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print(x+3)
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

Entonces una sesión interactiva puede lucir como esta:

```
>>> 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)
>>>
```

Funciones que convierten los doctest a código de Python, y posiblemente ejecuten el código sintetizado debajo del depurador:

doctest.**script_from_examples**(*s*)

Convierte texto con ejemplos a un script.

El argumento *s* es una cadena que contiene los ejemplos de doctest. La cadena se convierte a un script de Python, donde los ejemplos de doctest en *s* se convierten en código regular, y todo lo demás se convierte en comentarios de Python. El script generado se retorna como una cadena: Por ejemplo,

```
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
"""))
```

muestra:

```
# Set x and y to 1 and 2.
x, y = 1, 2
```

(continué en la próxima página)

(proviene de la página anterior)

```
#
# Print their sum:
print(x+y)
# Expected:
## 3
```

Esta función se usa internamente por otras funciones (véase más abajo), pero también pueden ser útiles cuando quieres transformar una sesión de Python interactiva en un script de Python.

doctest.**testsource**(*module*, *name*)

Convierte el doctest para un objeto en un script.

El argumento *module* es un objeto módulo, o un nombre por puntos de un módulo, que contiene el objeto cuyos doctest son de interés. El argumento *name* es el nombre (dentro del módulo) del objeto con los doctest de interés. El resultado es una cadena de caracteres, que contiene el docstring del objeto convertido en un script de Python, como se describe por *script_from_examples()* arriba. Por ejemplo, si el módulo `a.py` contiene un función de alto nivel `f()`, entonces

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

imprime una versión de script del docstring de la función `f()`, con los doctest convertidos en código, y el resto puesto en comentarios.

doctest.**debug**(*module*, *name*, *pm=False*)

Depura los doctest para un objeto.

Los argumentos *module* y *name* son los mismos que para la función *testsource()* arriba. El script de Python sintetizado para el docstring del objeto nombrado es escrito en un archivo temporal, y entonces ese archivo es ejecutado bajo el control del depurador de PYthon, *pdb*.

Se usa una copia superficial de `module.__dict__` para el contexto de ejecución local y global.

El argumento opcional *pm* controla si se usa la depuración post-mortem. Si *pm* tiene un valor verdadero, el archivo de script es ejecutado directamente, y el depurador está involucrado sólo si el script termina a través del lanzamiento de una excepción. Si lo hace, entonces la depuración post-mortem es invocada, a través de *pdb.post_mortem()*, pasando el objeto de rastreo desde la excepción sin tratar. Si no se especifica *pm*, o si es falso, el script se ejecuta bajo el depurador desde el inicio, a través de una llamada de *exec()* apropiada a *pdb.run()*.

doctest.**debug_src**(*src*, *pm=False*, *globs=None*)

Depura los doctest en una cadena de caracteres.

Es como la función function *debug()* arriba, excepto que una cadena de caracteres que contiene los ejemplos de doctest se especifica directamente, a través del argumento *src*.

El argumento opcional *pm* tiene el mismo significado como en la función *debug()* arriba.

El argumento opcional *globs* proporciona un diccionario a usar como contexto de ejecución local y global. Si no se especifica, o es `None`, se usa un diccionario vacío. Si se especifica, se usa una copia superficial del diccionario.

La clase *DebugRunner*, y las excepciones especiales que puede lanzar, son de más interés a los autores de frameworks de pruebas, y sólo serán descritos brevemente aquí. Véase el código fuente, y especialmente el docstring de *DebugRunner* (¡que es un doctest!) para más detalles:

class doctest.**DebugRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

Una subclase de *DocTestRunner* que lanza una excepción tan pronto como se encuentra una falla. Si ocurre una excepción inesperada, se lanza una excepción *UnexpectedException*, conteniendo la prueba, el ejemplo, y la excepción original. Si la salida no coincide, entonces se lanza una excepción *DocTestFailure*, conteniendo la prueba, el ejemplo, y la salida real.

Para información sobre los parámetros de construcción y los métodos, véase la documentación para *DocTestRunner* en la sección *API avanzada*.

Hay dos excepciones que se pueden lanzar por instancias de *DebugRunner*:

exception doctest.**DocTestFailure**(*test*, *example*, *got*)
Una excepción lanzada por *DocTestRunner* para señalar que la salida real del ejemplo de un doctest no coincidió con su salida esperada. Los argumentos del constructor se usan para inicializar los atributos del mismo nombre.

DocTestFailure define los siguientes atributos:

DocTestFailure.**test**
El objeto *DocTest* que estaba siendo ejecutado cuando el ejemplo falló.

DocTestFailure.**example**
El objeto *Example* que falló.

DocTestFailure.**got**
La salida real del ejemplo.

exception doctest.**UnexpectedException**(*test*, *example*, *exc_info*)
Una excepción lanzada por *DocTestRunner* para señalar que un ejemplo de doctest lanzó una excepción inesperada. Los argumentos del constructor se usan para inicializar los atributos del mismo nombre.

UnexpectedException define los siguientes atributos:

UnexpectedException.**test**
El objeto *DocTest* que estaba siendo ejecutado cuando el ejemplo falló.

UnexpectedException.**example**
El objeto *Example* que falló.

UnexpectedException.**exc_info**
Una tupla que contiene información sobre la excepción inesperada, como es retornado por *sys.exc_info()*.

26.7.8 Plataforma improvisada

Como se menciona en la introducción, *doctest* ha crecido para tener tres usos primarios:

1. Verificar los ejemplos en los docstring.
2. Pruebas de regresión.
3. Documentación ejecutable / pruebas literarias.

Estos usos tienen diferentes requerimientos, y es importante distinguirlos. En particular, llenar tus docstring con casos de prueba desconocidos conduce a mala documentación.

Cuando se escribe un docstring, escoja ejemplos de docstring con cuidado. Hay un arte para eso que se debe aprender – puede no ser natural al comienzo. Los ejemplos deben añadir valor genuino a la documentación. Un buen ejemplo a menudo puede valer muchas palabras. Si se hace con cuidado, los ejemplos serán invaluables para tus usuarios, y compensarán el tiempo que toma recolectarlos varias veces mientras los años pasan y las cosas cambian. Todavía estoy sorprendido de qué tan frecuente uno de mis ejemplos de *doctest* paran de funcionar después de un cambio «inofensivo».

Doctest también es una excelente herramienta para pruebas de regresión, especialmente si no escatimas en texto explicativo. Al intercalar prosa y ejemplos, se hace mucho más fácil mantener el seguimiento de lo que realmente se está probando, y por qué. Cuando una prueba falla, buena prosa puede hacer mucho más fácil comprender cuál es el problema, y cómo debe ser arreglado. Es verdad que puedes escribir comentarios extensos en pruebas basadas en código, pero pocos programadores lo hacen. Quizás es porque simplemente doctest hace escribir pruebas mucho más fácil que escribir código, mientras que escribir comentarios en código es mucho más difícil. Pienso que va más allá de eso: la actitud natural cuando se escribe una prueba basada en doctest es que quieres explicar los puntos finos de tu software, e ilustrarlos con ejemplos. Esto naturalmente lleva a archivos de pruebas que empiezan con las características más simples, y lógicamente progresan a complicaciones y casos extremos. Una narrativa coherente es el resultado, en vez de una colección de funciones aisladas que pruebas trozos aislados de funcionalidad aparentemente al azar. Es una actitud diferente, y produce resultados diferentes, difuminando la distinción entre probar y explicar.

Pruebas de regresión se limitan mejor a objetos o archivos dedicados. Hay varias opciones para organizar pruebas:

- Escribe archivos de texto que contienen los casos de prueba como ejemplos interactivos, y prueba los archivos usando *testfile()* o *DocFileSuite()*. Esto es lo recomendado, aunque es más fácil hacerlo para nuevos proyectos, diseñados desde el comienzo para usar doctest.
- Define funciones nombradas `_regrtest_topic` que consisten en docstrings únicas, que contienen casos de prueba por los tópicos nombrados. Estas funciones se pueden incluir en el mismo archivo que el módulo, o separadas en un archivo de prueba separado.
- Define un diccionario `__test__` que asigna desde temas de prueba de integración a los docstring que contienen casos de prueba.

Cuando has puesto tus pruebas en un módulo, el módulo puede ser el mismo *test runner*. Cuando una prueba falla, puedes hacer que tu *test runner* vuelva a ejecutar sólo los doctest fallidos mientras que tu depuras el problema. Aquí hay un ejemplo mínimo de tal *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))
```

Notas al pie de página

26.8 `unittest` — Infraestructura de tests unitarios

Código fuente: Lib/unittest/__init__.py

(Si ya estás familiarizado con los conceptos básicos de realización de tests, puedes saltar a *la lista de métodos de aserción*.)

La infraestructura de tests unitarios *unittest* se inspiró en primera instancia en JUnit y ofrece aspectos similares a las principales estructuras de tests unitarios más importantes de otros lenguajes. Da soporte a automatización de tests, inicialización compartida, código de cierre de los tests, agregación de los tests en colecciones e independencia de los tests de la infraestructura que los reporta.

Para conseguir esto, *unittest* da soporte a ciertos conceptos importantes de una forma orientada a objetos:

configuración de prueba (*fixture*) Un *test fixture* representa los preparativos para realizar una o más pruebas y las acciones de limpieza asociadas. Esto puede incluir, por ejemplo, la creación de bases de datos temporales, directorios o el arranque de procesos del servidor.

caso de prueba Un *test case* es la unidad mínima de prueba. Verifica la respuesta específica a un juego particular de entradas. *unittest* proporciona una clase base, *TestCase*, que se puede utilizar para crear nuevos casos de uso.

conjunto de pruebas Un *test suite* es una colección de casos de prueba, juegos de prueba o ambos. Se usa para agrupar pruebas que se han de ejecutar juntas.

ejecutor de pruebas Un *test runner* es un componente que dirige la ejecución de las pruebas y proporciona un resultado. El ejecutor puede disponer de una interfaz gráfica, de texto o devolver un valor especial que indique el resultado de la ejecución de las pruebas.

Ver también:

Módulo *doctest* Otro módulo de soporte a pruebas con una solución muy diferente.

Simple Smalltalk Testing: With Patterns El documento original de Kent Beck sobre infraestructuras de prueba mediante el patrón que utiliza *unittest*.

pytest Una infraestructura de pruebas unitarias de otro proveedor con una sintaxis más ligera de escritura de pruebas, por ejemplo: `assert func(10) == 42`.

The Python Testing Tools Taxonomy Una lista extensa de herramientas de prueba para Python incluyendo infraestructuras de pruebas funcionales y librerías de objetos sucedáneos.

Testing in Python Mailing List Grupo especializado en debate sobre las pruebas y las herramientas de prueba de Python.

El script `Tools/unittestgui/unittestgui.py` de la distribución de fuentes de Python es una herramienta gráfica para el descubrimiento y ejecución de pruebas. Está orientado sobre todo a principiantes en el tema de pruebas. Para entornos de producción se recomienda que las pruebas sean dirigidas por un sistema de integración continua como Buildbot, Jenkins o Hudson.

26.8.1 Ejemplo sencillo

El módulo *unittest* proporciona un conjunto de herramientas para construir y ejecutar pruebas. Esta sección demuestra que un pequeño subconjunto de las herramientas es suficiente para satisfacer las necesidades de la mayoría.

He aquí un breve script para probar estros tres métodos de cadena:

```
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()
```

Para crear un caso de prueba se genera una subclase de *unittest.TestCase*. Las tres pruebas se definen con métodos cuyos nombres comienzan por las letras `test`. Esta convención sobre nombres indica al ejecutor de pruebas qué métodos representan pruebas.

El quid de cada test es la llamada a *assertEqual()* para verificar un resultado esperado; *assertTrue()* o *assertFalse()* para verificar una condición; o *assertRaises()* para asegurar que se lanza una excepción específica. Se utilizan estos métodos en lugar de la sentencia `assert` para que el ejecutor de pruebas pueda acumular todos los resultados de la prueba de cara a realizar un informe.

Los métodos *setUp()* y *tearDown()* permiten definir instrucciones que han de ser ejecutadas antes y después, respectivamente, de cada método de prueba. Se describen con mas detalle en la sección *Organización del código de pruebas*.

El bloque final muestra un modo sencillo de ejecutar las pruebas. *unittest.main()* proporciona una interfaz de línea de órdenes para el script de prueba. Cuando se ejecuta desde la línea de órdenes, el script anterior produce una

salida como:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Si se le pasa una opción `-v` al script de prueba, se establecerá un nivel mayor de detalle del proceso de *unittest.main()* y se obtendrá la siguiente salida:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Los ejemplos anteriores muestra las características más usuales de *unittest*, que son suficientes para solventar las necesidades cotidianas de pruebas. El resto de la documentación explora el juego completo de características, que abundan en los mismos principios.

26.8.2 Interfaz de línea de comandos

Se puede usar el módulo *unittest* desde la línea de órdenes para ejecutar pruebas de módulos, clases o hasta métodos de prueba individuales:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

Se puede pasar una lista con cualquier combinación de nombres de módulo, así como clases o métodos completamente cualificados.

Se puede especificar los módulos de prueba por ruta de fichero también:

```
python -m unittest tests/test_something.py
```

Esto permite usar el completado automático de nombre de fichero de la shell para especificar el módulo de pruebas. El fichero especificado aún debe ser susceptible de importarse como módulo. La ruta se convierte en nombre de módulo eliminando ".py" y convirtiendo el separador de directorios por ".". Si se desea ejecutar un fichero de prueba que no se puede importar como módulo, se ha de ejecutar el fichero directamente.

Se pueden ejecutar las pruebas con más nivel de detalle (mayor verbosidad) pasando el parámetro *-v*:

```
python -m unittest -v test_module
```

Cuando se ejecuta sin argumentos, se inicia *Descubrimiento de pruebas*:

```
python -m unittest
```

Para obtener una lista de todas las opciones de línea de órdenes:

```
python -m unittest -h
```

Distinto en la versión 3.2: En versiones anteriores sólo era posible ejecutar métodos de prueba individuales, pero no módulos ni clases.

Opciones de la línea de órdenes

unittest da soporte a las siguientes opciones de línea de órdenes:

-b, --buffer
Los flujos de datos de salida estándar y error estándar se acumulan en un búfer durante la ejecución de pruebas. La salida de las pruebas correctas se descarta. La salida de las pruebas que fallan o devuelven un error se añade a los mensajes de fallo.

-c, --catch
La pulsación de `Control-C` durante la ejecución de pruebas espera a que termine la prueba en curso y da un informe de los resultados hasta ese momento. Una segunda pulsación de `Control-C` lanza la excepción *`KeyboardInterrupt`* usual.

Consultar en *Gestión de señales* las funciones que proporcionan esta funcionalidad.

-f, --failfast
Finaliza la ejecución tras el primer error o fallo.

-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 any of the given patterns are included.

Los patrones que contengan un comodín (`*`) se comprueban contra el nombre de la prueba usando *`fnmatch.fnmatchcase()`*; si no lo contienen, se usa una comprobación de contenido simple sensible a mayúsculas.

Los patrones se comprueban contra el nombre del método completamente cualificado como lo importa el cargador de pruebas.

Por ejemplo, `-k foo` coincide con `foo_tests.SomeTest.test_something` y con `bar_tests.SomeTest.test_foo`, pero no con `bar_tests.FooTest.test_something`.

--locals
Mostrar las variables locales en las trazas.

Nuevo en la versión 3.2: Se añadieron las opciones de línea de órdenes `-b`, `-c` y `-f`.

Nuevo en la versión 3.5: La opción de línea de órdenes `--locals`.

Nuevo en la versión 3.7: La opción de línea de órdenes `-k`.

La línea de órdenes también se puede usar para descubrimiento de pruebas, para ejecutar todas las pruebas de un proyecto o un subconjunto de éstas.

26.8.3 Descubrimiento de pruebas

Nuevo en la versión 3.2.

Unittest da soporte al descubrimiento de pruebas simples. Para ser compatible con el descubrimiento de pruebas, todos los ficheros de prueba deben ser módulos o paquetes (incluyendo *paquetes nominales*) importables desde el directorio superior del proyecto (por lo que sus nombres han de ser identificadores válidos).

El descubrimiento de pruebas está implementado en *`TestLoader.discover()`*, pero también se puede usar desde la línea de órdenes. La línea de órdenes básica es:

```
cd project_directory
python -m unittest discover
```

Nota: Como atajo, `python -m unittest` es el equivalente de `python -m unittest discover`. Si se desea pasar argumentos al descubrimiento de pruebas, hay que pasar la sub-orden `discover` explícitamente.

La sub-orden `discover` cuenta con las siguientes opciones:

-v, --verbose
Salida verbosa

-s, --start-directory directory
Directorio de inicio para el descubrimiento (. si se omite)

-p, --pattern pattern
Patrón para la búsqueda de ficheros de prueba (test*.py si se omite)

-t, --top-level-directory directory
Directorio superior del proyecto (el directorio inicial si se omite)

Las opciones *-s*, *-p*, y *-t* se pueden pasar por posición en el orden mostrado. Las siguientes líneas de órdenes son equivalentes:

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

Además de pasar una ruta, es posible pasar el nombre de un paquete, por ejemplo myproject.subpackage.test, como directorio de inicio. El nombre de paquete proporcionado será importado y su ubicación en el sistema de archivos será utilizada como directorio de inicio.

Prudencia: El descubrimiento de pruebas carga las pruebas importándolas. Una vez que el descubrimiento ha encontrado todos los ficheros de prueba a partir del directorio inicial, especificado, convierte las rutas en nombres de paquetes a importar. Por ejemplo, foo/bar/baz.py será importado como foo.bar.baz.

Si hay un paquete instalado globalmente y se intenta hacer un descubrimiento sobre una copia diferente a la instalada del paquete, *podría* ocurrir que se importe la versión incorrecta. Si ocurre esto, el descubrimiento lanza una advertencia y abandona.

Si se proporciona el directorio de inicio como nombre de paquete en lugar de ruta a un directorio, el descubrimiento asume que la ubicación importada es la deseada, así que no se da la advertencia descrita.

Los módulos y paquetes de prueba pueden personalizar la carta y descubrimiento de pruebas mediante el *protocolo load_tests*.

Distinto en la versión 3.4: El descubrimiento de pruebas admite *paquetes nominales* para el directorio de inicio. Tenga en cuenta que también se necesita el directorio de nivel superior. (por ejemplo, python -m unittest discover -s root/namespace -t root).

26.8.4 Organización del código de pruebas

Los bloques de construcción básicos de las pruebas unitarias son los *test cases*, escenarios simples que se han de configurar y cuya corrección hay que comprobar. En *unittest*, los casos de prueba están representados por instancias de *unittest.TestCase*. Para crear nuevos casos de prueba, hay que escribir subclases de *TestCase* o usar *FunctionTestCase*.

El código de pruebas de una instancia de *TestCase* debería ser completamente autónomo, de tal modo que se pueda ejecutar aisladamente o en una combinación arbitraria con otras clases de prueba.

La subclase más simple de *TestCase* se limita a implementar un método test (o un método que empiece por test) para realizar código de prueba específico:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

Adviértase que para probar algo, usamos uno de los métodos `assert*()` proporcionados por la clase base *TestCase*. Si la prueba no tiene éxito, se lanzará una excepción con un mensaje explicativo y *unittest* identificará el caso como *failure*. Cualquier otra excepción se considerará un *errors*.

Las pruebas pueden ser muchas y su preparación puede ser repetitiva. Por suerte, se puede «sacar factor común» a la preparación implementando un método *setUp()*, al que la infraestructura de prueba llamará para cada prueba que se ejecute:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        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: El orden en que se ejecutan las diversas pruebas se determina por orden alfabético de los nombres de métodos de prueba.

Si el método *setUp()* lanza una excepción mientras se ejecuta la prueba, la infraestructura considerará que la prueba ha sufrido un error y no se ejecutará el método de prueba propiamente dicho.

Análogamente, se puede proporcionar un método *tearDown()* que haga limpieza después de que se ejecute el método de prueba:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

Si *setUp()* se ejecuta sin errores, *tearDown()* se ejecutará tanto si el método de prueba tuvo éxito como si no.

Un entorno de trabajo así para el código de pruebas se llama *test fixture* . Se crea una nueva instancia de TestCase como juego de datos de prueba que se utiliza para cada método de prueba. De este modo, *setUp()*, *tearDown()* y `__init__()` se llamarán una vez por prueba.

Se recomienda usar implementaciones de *TestCase* para agrupar las pruebas de acuerdo con las características probadas. *unittest* proporciona una mecanismo para esto: el *test suite*, representado por la clase *TestSuite* de *unittest*. En la mayor parte de los casos, llamar a *unittest.main()* hará lo correcto y recolectará todos los casos de prueba del módulo para su ejecución.

Sin embargo, si se desea personalizar la construcción del paquete de pruebas, se puede hacer:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite

if __name__ == '__main__':
```

(continué en la próxima página)

(proviene de la página anterior)

```
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

Se puede poner las definiciones de los casos de prueba y de los paquetes de prueba en los mismos módulos que el código que prueban.(tal como `widget.py`), pero sacarlos a un módulo aparte como `test_widget.py` tiene diversas ventajas:

- El módulo de pruebas se puede ejecutar aisladamente desde la línea de órdenes.
- El código de pruebas se puede separar con más facilidad del código de producción.
- Disminuye la tentación de cambiar el código de prueba para ajustarse al código bajo prueba.
- El código de pruebas debería modificarse con mucha menor frecuencia que el código bajo prueba.
- Es más sencillo refactorizar el código bajo prueba.
- El código para probar módulos escritos en C ha de estar en módulos aparte, así que ¿por qué no mantener la consistencia?
- Si cambia la estrategia de prueba, no hay razón para cambiar el código fuente principal.

26.8.5 Reutilización de código de prueba anterior

Habrá personas que tengan código de prueba previo que deseen ejecutar desde *unittest*, sin conversión previa de cada antigua función de prueba a una subclase de *TestCase*.

Por esto, *unittest* proporciona una clase *FunctionTestCase*. Se puede utilizar esta subclase de *TestCase* para envolver una función de prueba existente. Se pueden proporcionar también funciones de preparación y desmontaje.

Dada la siguiente función de prueba:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

se puede crear una instancia de caso de prueba de la siguiente manera, con métodos opcionales de preparación y desmontaje:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

Nota: Aunque se puede usar *FunctionTestCase* para convertir rápidamente unas pruebas existentes a un sistema basado en *unittest*, no es una vía recomendada. Tomarse el tiempo de construir subclases bien construidas de *TestCase* hará las futuras refactorizaciones de pruebas futura considerablemente más fácil.

En algunos casos, los tests existentes pueden haber sido escritos usando el módulo *doctest*. En ese caso, *doctest* tiene una clase `DocTestSuite` que puede construir automáticamente instancias de *unittest.TestSuite* partiendo de los test basados en *doctest*.

26.8.6 Omitir tests y fallos esperados

Nuevo en la versión 3.1.

Unittest soporta omitir métodos individuales de tests e incluso clases completas de tests. Además, soporta marcar un test como un «fallo esperado», un test que está roto y va a fallar, pero no debería ser contado como fallo en *TestResult*.

Omitir un test es solo cuestión de emplear el *decorator* *skip()* o una de sus variantes condicionales, llamando a *TestCase.skipTest()* dentro de *setUp()* o en un método de test, o lanzando *SkipTest* directamente.

La omisión más básica tiene la siguiente forma:

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass

    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
```

Esta es la salida de ejecutar el ejemplo superior en modo verboso:

```
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)
```

Las clases pueden ser omitidas igual que los métodos:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

TestCase.setUp() puede omitir también el test. Esto es útil cuando un recurso que necesita ser puesto a punto no está disponible.

Los fallos esperados emplean el decorador *expectedFailure()*.

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

Es fácil lanzar tus propios decoradores que omitan haciendo un decorador que llame a *skip()* en el test cuando quiere ser omitido. Este decorador omite el test a menos que el objeto pasado tenga un cierto atributo:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

Los siguientes decoradores y la excepción implementan la omisión de tests y los fallos esperados:

`@unittest.`**`skip`**`(`*reason*`)`
Omitir incondicionalmente el test decorado. *reason* debe describir porqué el test está siendo omitido.

`@unittest.`**`skipIf`**`(`*condition*, *reason*`)`
Omitir el test decorado si *condition* es verdadero.

`@unittest.`**`skipUnless`**`(`*condition*, *reason*`)`
Omitir el test decorado a menos que *condition* sea verdadero.

`@unittest.`**`expectedFailure`**
Marca el test como un fallo esperado. Si falla el test o hay errores en la función de test misma (en lugar de en alguno de los métodos *test fixture*) será considerado un éxito. Si el test pasa, será considerado un fallo.

`exception` `unittest.`**`SkipTest`**`(`*reason*`)`
Esta excepción es lanzada para omitir un test.

Normalmente puedes usar directamente *TestCase.skipTest()* o uno de los decoradores de omisión en vez de lanzar esta excepción.

Los tests omitidos no ejecutarán *setUp()* o *tearDown()*. Las clases omitidas no ejecutarán *setUpClass()* o *tearDownClass()*. Los módulos omitidos no ejecutarán `setUpModule()` o `tearDownModule()`.

26.8.7 Distinguiendo iteraciones de tests empleando subtests

Nuevo en la versión 3.4.

Cuando hay diferencias muy pequeñas entre tus tests, por ejemplo algunos parámetros, unittest te permite distinguirlos dentro del cuerpo de un método de test empleando el administrador de contexto *subTest()* .

Por ejemplo, el siguiente 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)
```

producirá la siguiente salida:

```
======================================================================
FAIL: test_even (__main__.NumbersTest) (i=1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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
```

Sin usar un subtest, la ejecución se pararía después del primer fallo, y el error sería más difícil de diagnosticar porque el valor de `i` no se mostraría:

```
======================================================================
FAIL: test_even (__main__.NumbersTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

26.8.8 Clases y funciones

Esta sección describe en detalle la API de *unittest*.

Casos de test

class unittest.**TestCase**(*methodName='runTest'*)

Las instancias de la clase *TestCase* representan las unidades lógicas de test en el universo de *unittest*. Esta clase está pensada para ser utilizada como clase base, con los test específicos siendo implementados por subclases concretas. Esta clase implementa la interfaz que necesita el ejecutor de tests para permitirle llevar a cabo los tests, y métodos que el código de test puede utilizar para chequear y reportar distintos tipos de fallo.

Cada instancia de *TestCase* ejecutará un solo método base: el método llamado *methodName*. En la mayoría de usos de *TestCase*, no tendrás que cambiar el *methodName* ni reimplementar el método por defecto `runTest()`.

Distinto en la versión 3.2: *TestCase* puede instancias con éxito sin dar un *methodName*. Esto permite experimentar de manera sencilla con *TestCase* en el intérprete interactivo.

Las instancias de *TestCase* proveen tres grupos de métodos: un grupo empleado para ejecutar el test, otro usado para que la implementación del test chequee condiciones y reporte fallos, y algunos métodos de indagación que permiten recopilar información sobre el test en si mismo.

Los métodos en el primer grupo (ejecutando el test) son:

setUp()

Método llamado para preparar el banco de test. Es invocado inmediatamente antes de llamar al método de test; cualquier excepción lanzada por este método que no sea *AssertionError* o *SkipTest* será considerada un error en vez de un fallo del test. La implementación por defecto no hace nada.

tearDown()
Método llamado inmediatamente después de que se haya llamado el método de prueba y se haya registrado el resultado. Se llama así aunque el método de ensayo haya planteado una excepción, por lo que la aplicación en las subclases puede tener que ser especialmente cuidadosa en cuanto a la comprobación del estado interno. Cualquier excepción, que no sea *AssertionError* o *SkipTest*, planteada por este método se considerará un error adicional en lugar de un fallo de la prueba (aumentando así el número total de errores reportados). Este método sólo se llamará si *setUp()* tiene éxito, independientemente del resultado del método de prueba. La implementación por defecto no hace nada.

setUpClass()
Un método de clase llamado antes de que los tests en una clase individual sean ejecutados. `setUpClass` es llamado con la clase como el único argumento y debe ser decorada como *classmethod()*:

```
@classmethod
def setUpClass(cls):
    ...
```

Vea *Class and Module Fixtures* para más detalles.

Nuevo en la versión 3.2.

tearDownClass()
Un método de clase llamado después de que se hayan realizado tests en una clase individual. `tearDownClass` se llama con la clase como único argumento y debe ser decorado como un *classmethod()*:

```
@classmethod
def tearDownClass(cls):
    ...
```

Vea *Class and Module Fixtures* para más detalles.

Nuevo en la versión 3.2.

run(*result=None*)
Ejecutar la prueba, recogiendo el resultado en el objeto *TestResult* pasado como *result*. Si se omite *result* o `None`, se crea un objeto resultado temporal (llamando al método *defaultTestResult()*) y se emplea ese. El objeto resultante se devuelve al invocador de *run()*.

El mismo efecto puede conseguirse simplemente llamando a la instancia *TestCase*.

Distinto en la versión 3.3: Las versiones previas de `run` no retornaban el resultado. Tampoco lo hacía la llamada a una instancia.

skipTest(*reason*)
Llamar a esto durante un método de prueba o *setUp()* se salta el test actual. Ver *Omitir tests y fallos esperados* para más información.

Nuevo en la versión 3.1.

subTest(*msg=None*, ***params*)
Retorna un gestor de contexto que ejecuta el bloque de código adjunto como un subtest. *msg* y *params* son valores opcionales y arbitrarios que se muestran cuando falla un subtest, permitiéndole identificarlos claramente.

Un caso de test puede contener cualquier número de declaraciones de subtest, y pueden anidarse arbitrariamente.

Ver *Distinguiendo iteraciones de tests empleando subtests* para más información.

Nuevo en la versión 3.4.

debug()
Realice el test sin recoger el resultado. Esto permite que las excepciones planteadas por el test se propaguen al invocado, y puede utilizarse para apoyar la ejecución de tests bajo un depurador.

La clase *TestCase* proporciona varios métodos de afirmación para comprobar y reportar fallos. En la siguiente tabla se enumeran los métodos más utilizados (consulte las tablas siguientes para ver más métodos de afirmación):

Método	Comprueba que	Nuevo en
assertEqual(a, b)	`a == b`	
assertNotEqual(a, b)	`a != b`	
assertTrue(x)	`bool(x) is True`	
assertFalse(x)	`bool(x) is False`	
assertIs(a, b)	`a is b`	3.1
assertIsNot(a, b)	`a is not b`	3.1
assertIsNone(x)	`x is None`	3.1
assertIsNotNone(x)	`x is not None`	3.1
assertIn(a, b)	`a in b`	3.1
assertNotIn(a, b)	`a not in b`	3.1
assertIsInstance(a, b)	`isinstance(a, b)`	3.2
assertNotIsInstance(a, b)	`not isinstance(a, b)`	3.2

Todos los métodos de aserción aceptan un argumento *msg* que, si se especifica, se utiliza como mensaje de error en caso de fallo (véase también *longMessage*). Tenga en cuenta que el argumento de la palabra clave *msg* puede pasarse a *assertRaises()*, *assertRaisesRegex()*, *assertWarns()*, *assertWarnsRegex()* sólo cuando se utilizan como gestor de contexto.

assertEqual(*first*, *second*, *msg=None*)
Testea que *first* y *second* son iguales. Si los valores no comparan como iguales, el test fallará.

Además, si *first* y *second* son exactamente del mismo tipo y uno de lista, tuple, dict, set, frozenset o str o cualquier tipo que una subclase registre con *addTypeEqualityFunc()* se llamará a la función de igualdad específica del tipo para generar un mensaje de error por defecto más útil (véase también la *lista de métodos específicos del tipo*).

Distinto en la versión 3.1: Añadida la llamada automática de la función de igualdad de tipo específico.

Distinto en la versión 3.2: *assertMultiLineEqual()* añadido como la función por defecto para igualdad de tipos cuando se comparan cadenas.

assertNotEqual(*first*, *second*, *msg=None*)
Testea que *first* y *second* no son iguales. Si los valores son iguales, el test fallará.

assertTrue(*expr*, *msg=None*)
assertFalse(*expr*, *msg=None*)
Testea que *expr* es verdadero (o falso).

Note que esto es equivalente a `bool(expr) is True` y no a `expr is True` (use `assertIs(expr, True)` para lo último). Este método también debe evitarse cuando se disponga de métodos más específicos (por ejemplo, `assertEqual(a, b)` en lugar de `assertTrue(a == b)`), porque proporcionan un mejor mensaje de error en caso de fallo.

assertIs(*first*, *second*, *msg=None*)
assertIsNot(*first*, *second*, *msg=None*)
Testea si *first* y *second* son (o no) el mismo objeto.

Nuevo en la versión 3.1.

assertIsNone(*expr*, *msg=None*)
assertIsNotNone(*expr*, *msg=None*)
Testea que *expr* es (o no es) `None`.

Nuevo en la versión 3.1.

assertIn(*member*, *container*, *msg=None*)
assertNotIn(*member*, *container*, *msg=None*)
Testea que *member* está (o no está) en *container*.

Nuevo en la versión 3.1.

assertIsInstance(*obj*, *cls*, *msg=None*)
assertNotIsInstance(*obj*, *cls*, *msg=None*)
Testea que *obj* es (o no es) una instancia de *cls* (que puede ser una clase o una tupla de clases, de la misma forma que soporta `isinstance()`). Para chequear por el tipo exacto, use `assertIs(type(obj), cls)`.

Nuevo en la versión 3.2.

Es también posible chequear la producción de excepciones, advertencias y mensajes de log usando los siguientes métodos:

Método	Comprueba que	Nuevo en
`assertRaises(exc, fun, *args, **kwds)`	`fun(*args, **kwds)` lanza *exc*	
`assertRaisesRegex(exc, r, fun, *args, **kwds)`	`fun(*args, **kwds)` lanza *exc* y el mensaje coincide con regex *r*	3.1
`assertWarns(warn, fun, *args, **kwds)`	`fun(*args, **kwds)` lanza *warn*	3.2
`assertWarnsRegex(warn, r, fun, *args, **kwds)`	`fun(*args, **kwds)` lanza *warn* y el mensaje coincide con regex *r*	3.2
`assertLogs(logger, level)`	El bloque `with` vuelca sus logs a *logger* con el *level* mínimo	3.4
`assertLogs(logger, level)`	**El bloque `with` no ingresa** El bloque `with` vuelca sus logs a *logger* con el *level* mínimo	3.10

assertRaises(*exception*, *callable*, **args*, ***kwds*)
assertRaises(*exception*, *, *msg=None*)
Testea que se lanza una excepción cuando se llama a *callable* con cualquier argumento posicional o de palabra clave que también se pasa a `assertRaises()`. El test pasa si se lanza *exception*, es un error si se lanza otra excepción, o falla si no se lanza ninguna excepción. Para tener en cuenta cualquiera de un grupo de excepciones, una tupla que contenga las clases de excepción puede ser pasada como *exception*.

Si sólo se dan los argumentos de *exception* y posiblemente *msg*, retorna un administrador de contexto para que el código testado pueda ser escrito en línea en lugar de como una función:

```
with self.assertRaises(SomeException):
    do_something()
```

Cuando se emplea como un administrador de contexto, `assertRaises()` acepta el argumento por palabra clave adicional *msg*.

El gestor de contexto almacenará el objeto de excepción capturado en su atributo `exception` . Esto puede ser útil si la intención es realizar comprobaciones adicionales sobre la excepción planteada:

```
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: Añadió la capacidad de usar `assertRaises()` como gestor de contexto.

Distinto en la versión 3.2: Añadido el atributo `exception` .

Distinto en la versión 3.3: Añadido el argumento por palabra clave *msg* cuando se emplea un gestor de contexto.

assertRaisesRegex(*exception*, *regex*, *callable*, **args*, ***kwds*)
assertRaisesRegex(*exception*, *regex*, ***, *msg=None*)

Como *assertRaises()* pero también testea que *regex* coincide en la representación de la cadena de la excepción planteada. *regex* puede ser un objeto de expresión regular o una cadena que contiene una expresión regular adecuada para ser usada por *re.search()*. Ejemplos:

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                       int, 'XYZ')
```

o:

```
with self.assertRaisesRegex(ValueError, 'literal'):
   int('XYZ')
```

Nuevo en la versión 3.1: Añadido bajo el nombre de `assertRaisesRegexp`.

Distinto en la versión 3.2: Renombrado a *assertRaisesRegex()*.

Distinto en la versión 3.3: Añadido el argumento por palabra clave *msg* cuando se emplea un gestor de contexto.

assertWarns(*warning*, *callable*, **args*, ***kwds*)
assertWarns(*warning*, ***, *msg=None*)

Testea que una advertencia se activa cuando se llama a *callable* con cualquier argumento posicional o de palabra clave que también se pasa a *assertWarns()*. El test pasa si se activa el *warning* y falla si no lo hace. Cualquier excepción es un error. Para considerar cualquiera de un grupo de advertencias, una tupla que contenga las clases de advertencia puede ser pasada como *warnings*.

Si sólo se dan los argumentos de *advertencia* y *msg*, retorna un gestor de contexto para que el código testado pueda ser escrito en línea en lugar de como una función:

```
with self.assertWarns(SomeWarning):
    do_something()
```

Cuando se usa como gestor de contexto, *assertWarns()* acepta el argumento de palabra clave adicional *msg*.

El gestor de contexto almacenará el objeto de advertencia capturado en su atributo `warning`, y la línea del código que disparó las advertencias en los atributos `filename` y `lineno` . Esto puede ser útil si la intención es realizar comprobaciones adicionales sobre la advertencia capturada:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

Este método funciona independientemente de los filtros de aviso que estén en su lugar cuando se llame.

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Añadido el argumento por palabra clave *msg* cuando se emplea un gestor de contexto.

assertWarnsRegex(*warning*, *regex*, *callable*, **args*, ***kwds*)
assertWarnsRegex(*warning*, *regex*, ***, *msg=None*)

Como *assertWarns()* pero también testea que *regex* coincide en el mensaje del aviso disparado. *regex* puede ser un objeto de expresión regular o una cadena que contiene una expresión regular adecuada para ser usada por *re.search()*. Ejemplo:

```
self.assertWarnsRegex(DeprecationWarning,
                      r'legacy_function\(\) is deprecated',
                      legacy_function, 'XYZ')
```

o:

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Añadido el argumento por palabra clave *msg* cuando se emplea un gestor de contexto.

assertLogs(*logger=None*, *level=None*)

Un gestor de contexto para comprobar que al menos un mensaje está registrado en el *logger* o en uno de sus hijos, con al menos el *level* dado.

Si se da, *logger* debería ser un objeto *logging.Logger* o un *str* dando el nombre de un logger. El valor por defecto es el root logger, que captará todos los mensajes.

Si se da, *level* debe ser un nivel de logging numérico o su equivalente en cadena (por ejemplo, o bien `"ERROR"` o `logging.ERROR`). El valor por defecto es `logging.INFO`.

El test pasa si al menos un mensaje emitido dentro del bloque `with` coincide con las condiciones de *logger* y *level*, de lo contrario falla.

El objeto devuelto por el gestor de contexto es un ayudante de grabación que lleva un registro de los mensajes de registro que coinciden. Tiene dos atributos:

records
: Una lista de objetos *logging.LogRecord* de los mensajes de log coincidentes.

output
: Una lista de objetos *str* con la salida forrajeada en los mensajes coincidentes.

Ejemplo:

```
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.

assertNoLogs(*logger=None*, *level=None*)

Un gestor de contexto para comprobar que al menos un mensaje está registrado en el *logger* o en uno de sus hijos, con al menos el *level* dado.

Si se da, *logger* debería ser un objeto *logging.Logger* o un *str* dándole el nombre de un logger. El valor por defecto es el root logger, que captará todos los mensajes.

Si se da, *level* debe ser un nivel de logging numérico o su equivalente en cadena (por ejemplo, o bien `"ERROR"` o `logging.ERROR`). El valor por defecto es `logging.INFO`.

A diferencia de *assertLogs()*, el gestor de contexto no devolverá nada.

Nuevo en la versión 3.10.

Hay también otros métodos empleados para realizar comprobaciones más específicas, tales como:

Método	Comprueba que	Nuevo en
assertAlmostEqual(a, b)	`round(a-b, 7) == 0`	
assertNotAlmostEqual(a, b)	`round(a-b, 7) != 0`	
assertGreater(a, b)	`a > b`	3.1
assertGreaterEqual(a, b)	`a >= b`	3.1
assertLess(a, b)	`a < b`	3.1
assertLessEqual(a, b)	`a <= b`	3.1
assertRegex(s, r)	`r.search(s)`	3.1
assertNotRegex(s, r)	`not r.search(s)`	3.2
assertCountEqual(a, b)	*a* y *b* tienen los mismos elementos y en el mismo número, sin importar su orden.	3.2

assertAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)
assertNotAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)

Testea que *first* y *second* son aproximadamente (o no aproximadamente) iguales calculando su diferencia, redondeando al número dado de puntos *places* decimales (por defecto 7), y comparado a cero. Nótese que estos métodos redondean los valores al número dado de *puntos decimales* (por ejemplo como la función *round()*) y no *cifras significativas*.

Si se suministra *delta* en vez de *places* que entonces la diferencia entre *first* y *second* deba ser menor o igual a (o mayor que) *delta*.

Suministrar tanto *delta* como *places* lanza un *TypeError*.

Distinto en la versión 3.2: *assertAlmostEqual()* considera automáticamente casi iguales a los objetos que se comparan igual. *assertNotAlmostEqual()* falla automáticamente si los objetos comparan iguales. Añadido el argumento de palabra clave *delta*.

assertGreater(*first*, *second*, *msg=None*)
assertGreaterEqual(*first*, *second*, *msg=None*)
assertLess(*first*, *second*, *msg=None*)
assertLessEqual(*first*, *second*, *msg=None*)

Prueba que *first* es respectivamente >, >=, < o <= que *second* dependiendo del nombre del método. Si no, el test fallará:

```
>>> 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*)

Testea que una búsqueda *regex* coincide (o no coincide) con el *text*. En caso de fallo, el mensaje de error incluirá el patrón y el *text* (o el patrón y la parte de *text* que coincida inesperadamente). *regex* puede ser un objeto de expresión regular o una cadena que contiene una expresión regular adecuada para ser utilizada por *re.search()*.

Nuevo en la versión 3.1: Añadido bajo el nombre de `assertRegexpMatches`.

Distinto en la versión 3.2: El método `assertRegexpMatches()` ha sido renombrado a *assertRegex()*.

Nuevo en la versión 3.2: *assertNotRegex()*.

Nuevo en la versión 3.5: El nombre `assertNotRegexpMatches` es un alias obsoleto para *assertNotRegex()*.

assertCountEqual(*first*, *second*, *msg=None*)

Testea que la secuencia *first* contiene los mismos elementos que *second*, independientemente de su orden. Cuando no lo hagan, se generará un mensaje de error con las diferencias entre las secuencias.

Los elementos duplicados *no* son ignorados cuando se comparan *first* y *second*. Verifica si cada elemento tiene la misma cuenta en ambas secuencias. Equivalente a: `assertEqual(Counter(list(first)), Counter(list(second)))` pero funciona también con secuencias de objetos que no son hashable.

Nuevo en la versión 3.2.

El método `assertEqual()` envía la comprobación de igualdad de los objetos del mismo tipo a diferentes métodos específicos de tipo. Estos métodos ya están implementados para la mayoría de los tipos incorporados, pero también es posible registrar nuevos métodos usando `addTypeEqualityFunc()`:

addTypeEqualityFunc(*typeobj*, *function*)

Registra un método específico de tipo llamado por `assertEqual()` para comprobar si dos objetos del mismo *typeobj* (no subclases) comparan como iguales. *function* debe tomar dos argumentos posicionales y un tercer argumento de palabra clave msg=None tal y como lo hace `assertEqual()`. Debe lanzar `self.failureException(msg)` cuando se detecta una desigualdad entre los dos primeros parámetros, posiblemente proporcionando información útil y explicando las desigualdades en detalle en el mensaje de error.

Nuevo en la versión 3.1.

La lista de métodos específicos de tipo utilizados automáticamente por `assertEqual()` se resumen en la siguiente tabla. Tenga en cuenta que normalmente no es necesario invocar estos métodos directamente.

Método	Usado para comparar	Nuevo en
`assertMultiLineEqual(a, b)`	strings	3.1
`assertSequenceEqual(a, b)`	sequences	3.1
`assertListEqual(a, b)`	lists	3.1
`assertTupleEqual(a, b)`	tuples	3.1
`assertSetEqual(a, b)`	sets or frozensets	3.1
`assertDictEqual(a, b)`	dicts	3.1

assertMultiLineEqual(*first*, *second*, *msg=None*)

Testea que la cadena multilínea *first* es igual a la cadena *second*. Cuando no sea igual, una diferencia de las dos cadenas que resalte las diferencias se incluirá en el mensaje de error. Este método se utiliza por defecto cuando se comparan cadenas con `assertEqual()`.

Nuevo en la versión 3.1.

assertSequenceEqual(*first*, *second*, *msg=None*, *seq_type=None*)

Testea que dos secuencias son iguales. Si se suministra un *seq_type*, tanto *first* como *second* deben ser instancias de *seq_type* o se lanzará un fallo. Si las secuencias son diferentes se construye un mensaje de error que muestra la diferencia entre las dos.

Este método no es llamado directamente por `assertEqual()`, pero se usa para implementar `assertListEqual()` y `assertTupleEqual()`.

Nuevo en la versión 3.1.

assertListEqual(*first*, *second*, *msg=None*)

assertTupleEqual(*first*, *second*, *msg=None*)

Testea que dos listas o tuplas son iguales. Si no es así, se construye un mensaje de error que muestra sólo las diferencias entre las dos. También se lanza un error si alguno de los parámetros es del tipo equivocado. Estos métodos se utilizan por defecto cuando se comparan listas o tuplas con `assertEqual()`.

Nuevo en la versión 3.1.

assertSetEqual(*first*, *second*, *msg=None*)

Testea que dos conjuntos son iguales. Si no es así, se construye un mensaje de error que enumera las

diferencias entre los conjuntos. Este método se utiliza por defecto cuando se comparan los conjuntos o frozensets con *assertEqual()*.

Falla si cualquiera de *first* o *second* no tiene un método de `set.difference()`.

Nuevo en la versión 3.1.

assertDictEqual(*first*, *second*, *msg=None*)
Testea que dos diccionarios son iguales. Si no es así, se construye un mensaje de error que muestra las diferencias en los diccionarios. Este método se usará por defecto para comparar los diccionarios en las llamadas a *assertEqual()*.

Nuevo en la versión 3.1.

Finalmente, *TestCase* proporciona los siguientes métodos y atributos:

fail(*msg=None*)
Señala un fallo del test incondicionalmente, con *msg* o `None` para el mensaje de error.

failureException
Este atributo de clase da la excepción lanzada por el método de test. Si un marco de pruebas necesita utilizar una excepción especializada, posiblemente para llevar información adicional, debe subclasificar esta excepción para «jugar limpio» con el marco. El valor inicial de este atributo es *AssertionError*.

longMessage
Este atributo de clase determina lo que ocurre cuando se pasa un mensaje de fallo personalizado como el argumento msg a una llamada assertXYY que falla. `True` es el valor por defecto. En este caso, el mensaje personalizado se añade al final del mensaje de fallo estándar. Cuando se establece en `False`, el mensaje personalizado reemplaza al mensaje estándar.

La configuración de la clase puede ser anulada en los métodos de test individuales asignando un atributo de instancia, self.longMessage, a `True` o `False` antes de llamar a los métodos assert.

La configuración de la clase se reajusta antes de cada llamada de test.

Nuevo en la versión 3.1.

maxDiff
Este atributo controla la longitud máxima de las diferencias de salida de métodos assert que reportan diferencias en caso de fallo. El valor predeterminado es de 80*8 caracteres. Los métodos assert afectados por este atributo son *assertSequenceEqual()* (incluyendo todos los métodos de comparación de secuencias que le delegan), *assertDictEqual()* y *assertMultiLineEqual()*.

Poner `maxDiff` en `None` significa que no hay una longitud máxima de diferencias.

Nuevo en la versión 3.2.

Los marcos de test pueden utilizar los siguientes métodos para reunir información sobre el test:

countTestCases()
Retorna el número de tests representados por este objeto de test. Para las instancias de *TestCase*, este siempre será `1`.

defaultTestResult()
Retorna una instancia de la clase de resultado de test que debería utilizarse para esta clase de caso de test (si no se proporciona otra instancia de resultado al método *run()*).

Para las instancias de *TestCase*, ésta siempre será una instancia de *TestResult*; las subclases de *TestCase* deben anular esto según sea necesario.

id()
Devuelva una cadena que identifique el caso de test específico. Normalmente es el nombre completo del método de test, incluyendo el nombre del módulo y de la clase.

shortDescription()
Devuelve una descripción de la prueba, o `None` si no se ha proporcionado ninguna descripción. La implementación por defecto de este método devuelve la primera línea de la docstring del método de test, si está disponible, o `None` .

Distinto en la versión 3.1: En 3.1 esto se cambió para añadir el nombre del test a la descripción corta incluso en presencia de una docstring. Esto causó problemas de compatibilidad con las extensiones de unittest y la adición del nombre de test fue movida a la *TextTestResult* en Python 3.2.

addCleanup(*function*, /, **args*, ***kwargs*)
Añade una función que se llamará después de *tearDown()* a los recursos de limpieza utilizados durante el test. Las funciones se llamarán en orden inverso al orden en que se agregan (LIFO). Se llaman con cualquier argumento y argumentos de palabra clave que se pase a *addCleanup()* cuando se agregan.

Si *setUp()* falla, lo que significa que *tearDown()* no se llama, entonces cualquier función de limpieza añadida seguirá siendo llamada.

Nuevo en la versión 3.1.

doCleanups()
Este método se llama incondicionalmente después de *tearDown()*, o después de *setUp()* si *setUp()* lanza una excepción.

Es responsable de llamar a todas las funciones de limpieza añadidas por *addCleanup()*. Si necesitas que las funciones de limpieza se llamen *con anterioridad* a *tearDown()* entonces puedes llamar a *doCleanups()* tú mismo.

doCleanups() saca los métodos de la pila de funciones de limpieza uno a uno, así que se puede llamar en cualquier momento.

Nuevo en la versión 3.1.

classmethod addClassCleanup(*function*, /, **args*, ***kwargs*)
Añade una función que se llamará después de *tearDownClass()* para limpiar recursos utilizados durante la clase de test. Las funciones se llamarán en orden inverso al orden en que se agregan (LIFO). Se llaman con cualquier argumento y argumento de palabra clave que se pase a *addClassCleanup()* cuando se añadan.

Si *setUpClass()* falla, lo que significa que *tearDownClass()* no se invoca, entonces cualquier función de limpieza añadida seguirá siendo llamada.

Nuevo en la versión 3.8.

classmethod doClassCleanups()
Este método se llama incondicionalmente después de *tearDownClass()*, o después de *setUpClass()* si *setUpClass()* lanza una excepción.

Es responsable de llamar a todas las funciones de limpieza añadidas por `addCleanupClass()`. Si necesitas que las funciones de limpieza se llamen *con anterioridad* a *tearDownClass()* entonces puedes llamar a `doCleanupsClass()` tú mismo.

`doCleanupsClass()` saca los métodos de la pila de funciones de limpieza de uno en uno, así que se puede llamar en cualquier momento.

Nuevo en la versión 3.8.

class `unittest.`**IsolatedAsyncioTestCase**(*methodName='runTest'*)
Esta clase proporciona una API similar a *TestCase* y también acepta corutinas como funciones de test.

Nuevo en la versión 3.8.

coroutine asyncSetUp()
Método llamado para preparar la configuración de test. Esto se llama después de `setUp()`. Se llama inmediatamente antes de llamar al método de test; aparte de *AssertionError* o *SkipTest*, cualquier excepción lanzada por este método se considerará un error más que un fallo del test. La implementación por defecto no hace nada.

coroutine asyncTearDown()
Método llamado inmediatamente después de que se haya llamado el método de test y se haya registrado el resultado. Esto se llama antes de `tearDown()`. Se llama así aunque el método de test haya lanzado una excepción, por lo que la implementación en las subclases puede necesitar ser particularmente cuidadosa en la comprobación del estado interno. Cualquier excepción, que no sea *AssertionError* o *SkipTest*,

lanzada por este método se considerará un error adicional en lugar de un fallo del test (aumentando así el número total de errores reportados). Este método sólo se llamará si *asyncSetUp()* tiene éxito, independientemente del resultado del método de test. La implementación por defecto no hace nada.

addAsyncCleanup(*function*, */*, **args*, ***kwargs*)
Este método acepta una corutina que puede ser utilizada como función de limpieza.

run(*result=None*)
Establece un nuevo bucle de eventos para ejecutar el test, recogiendo el resultado en el objeto *TestResult* pasado como *result*. Si se omite *result* o `None`, se crea un objeto resultado temporal (llamando al método `defaultTestResult()`) y se utiliza. El objeto resultante se devuelve al invocado de *run()*. Al final del test se cancelan todas las tareas del bucle de eventos.

Un ejemplo ilustrando el orden:

```
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()
```

Después de ejecutar el test, `events` contendría `["setUp", "asyncSetUp", "test_response", "asyncTearDown", "tearDown", "cleanup"]`.

class unittest.**FunctionTestCase**(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)
Esta clase implementa la porción de la interfaz *TestCase* que permite al corredor de tests conducir los tests, pero no proporciona los métodos que el código de test puede utilizar para comprobar e informar de los errores. Se utiliza para crear casos de test utilizando código de prueba heredado, lo que permite que se integre en un marco de tests basado en *unittest*.

Alias obsoletos

Por razones históricas, algunos de los métodos de *TestCase* tenían uno o más alias que ahora están obsoletos. La siguiente tabla lista los nombres correctos junto con sus alias obsoletos:

Nombre del método	Alias deprecado	Alias deprecado
assertEqual()	failUnlessEqual	assertEquals
assertNotEqual()	failIfEqual	assertNotEquals
assertTrue()	failUnless	assert_
assertFalse()	failIf	
assertRaises()	failUnlessRaises	
assertAlmostEqual()	failUnlessAlmostEqual	assertAlmostEquals
assertNotAlmostEqual()	failIfAlmostEqual	assertNotAlmostEquals
assertRegex()		assertRegexpMatches
assertNotRegex()		assertNotRegexpMatches
assertRaisesRegex()		assertRaisesRegexp

Obsoleto desde la versión 3.1: Los alias de fail* que figuran en la segunda columna han sido declarados obsoletos.

Obsoleto desde la versión 3.2: Los alias de aserción* que figuran en la tercera columna han sido declarados obsoletos.

Obsoleto desde la versión 3.2: `assertRegexpMatches` y `assertRaisesRegexp` han sido renombrados a *assertRegex()* y *assertRaisesRegex()*.

Obsoleto desde la versión 3.5: El nombre `assertNotRegexpMatches` se ha declarado obsoleto en favor de *assertNotRegex()*.

Agrupando tests

class unittest.**TestSuite**(*tests=()*)

Esta clase representa una agregación de casos de test individuales y conjuntos de tests. La clase presenta la interfaz que necesita el corredor de tests para poder ser ejecutado como cualquier otro caso de test. Ejecutar una instancia *TestSuite* es lo mismo que iterar sobre el conjunto, ejecutando cada test individualmente.

Si se indican *tests*, debe ser un iterable de casos de test individuales u otros conjuntos de tests que se usarán para construir el conjunto inicialmente. Se proporcionan métodos adicionales para añadir casos de test y conjuntos a la colección más adelante.

Los objetos de *TestSuite* se comportan de manera muy parecida a los objetos de *TestCase*, excepto que no implementan un test. En cambio, se usan para agregar tests en grupos de tests que deben ser ejecutados juntos. Existen algunos métodos adicionales para agregar tests a las instancias de *TestSuite*:

addTest(*test*)

Añade un *TestCase* o *TestSuite* al conjunto.

addTests(*tests*)

Añade todos los tests de un iterable de *TestCase* y *TestSuite* a este conjunto de tests.

Esto equivale a iterar sobre *tests*, llamando a *addTest()* para cada elemento.

TestSuite comparte los siguientes métodos con *TestCase*:

run(*result*)

Ejecuta los tests asociados a este conjunto, recogiendo el resultado en el objeto de resultado del test pasado como *result*. Tenga en cuenta que a diferencia de *TestCase.run()*, *TestSuite.run()* requiere que se pase el objeto resultado.

debug()

Ejecuta los tests asociados con este conjunto sin recoger los resultados. Esto permite que las excepciones

lanzadas por este test sean propagadas al invocador y puedes ser usadas para apoyar tests que están ejecutándose con un debugger.

countTestCases()
Retorna el numero de tests representados por este objeto de test, incluidos todos los test individuales y los sub-conjuntos.

__iter__()
Los tests agrupados por una *TestSuite* se acceden siempre por iteración. Las subclases pueden proporcionar tests anulando *__iter__()*. Tenga en cuenta que este método puede ser invocado varias veces en un mismo conjunto (por ejemplo, cuando se cuentan los tests o se comparan por igualdad), por lo que los tests retornados por iteraciones repetidas antes de *TestSuite.run()* deben ser los mismos para cada iteración de invocación. Después de *TestSuite.run()*, los invocados no deben confiar en los tests retornados por este método a menos que el invocado utilice una subclase que anule `TestSuite._removeTestAtIndex()` para preservar las referencias de los tests.

Distinto en la versión 3.2: En versiones anteriores la *TestSuite* accedía a los test directamente en lugar de a través de la iteración, por lo que anular *__iter__()* no era suficiente para proporcionar los tests.

Distinto en la versión 3.4: En versiones anteriores, la *TestSuite* tenía referencias a cada *TestCase* después de *TestSuite.run()*. Las subclases pueden restaurar ese comportamiento anulando `TestSuite._removeTestAtIndex()`.

En el uso típico de un objeto *TestSuite*, el método *run()* es invocado por un `TestRunner` en lugar de por el marco de test de pruebas automático del usuario final.

Cargando y ejecutando tests

class unittest.**TestLoader**
La clase *TestLoader* se utiliza para crear conjuntos de tests a partir de clases y módulos. Normalmente, no es necesario crear una instancia de esta clase; el módulo *unittest* proporciona una instancia que puede ser compartida como *unittest.defaultTestLoader*. Sin embargo, el uso de una subclase o instancia permite la personalización de algunas propiedades configurables.

Los objetos *TestLoader* tienen los siguientes atributos:

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 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.

Los objetos *TestLoader* tienen los siguientes métodos:

loadTestsFromTestCase(*testCaseClass*)
Devuelve un conjunto de todos los casos de test contenidos en la *TestCase*derivada de `testCaseClass`.

Se crea una instancia de caso de test para cada método nombrado por *getTestCaseNames()*. Por defecto, estos son los nombres de los métodos que comienzan con `test`. Si *getTestCaseNames()* no retorna ningún método, pero se implementa el método `runTest()`, se crea un único caso de test para ese método.

loadTestsFromModule(*module*, *pattern=None*)
Devuelva un conjunto de todos los casos de test contenidos en el módulo dado. Este método busca en *module* clases derivadas de *TestCase* y crea una instancia de la clase para cada método de test definido para la clase.

Nota: Aunque el uso de una jerarquía de clases derivadas de *TestCase* puede ser conveniente para compartir configuraciones y funciones de ayuda, la definición de métodos de test en clases base que no

están destinadas a ser instanciadas directamente no complementa bien con este método. Hacerlo, sin embargo, puede ser útil cuando las configuraciones son diferentes y están definidas en subclases.

Si un módulo proporciona una función `load_tests` será llamado para cargar los tests. Esto permite a los módulos personalizar la carga de los tests. Este es el *load_tests protocol*. El argumento *pattern* se pasa como tercer argumento a `load_tests`.

Distinto en la versión 3.2: Se ha añadido soporte para `load_tests`.

Distinto en la versión 3.5: El argumento por defecto *use_load_tests* no documentado y no oficial es obsoleto e ignorado, aunque sigue siendo aceptado por la retrocompatibilidad. El método también acepta ahora un argumento de sólo palabra clave *pattern* que se pasa a `load_tests` como tercer argumento.

loadTestsFromName(*name*, *module=None*)

Retorna un conjunto de todos los casos de test dado un especificador de cadena.

El especificador *name* es un «nombre punteado» que puede resolverse ya sea a un módulo, una clase de caso de test, un método de test dentro de una clase de caso de test, una instancia `TestSuite`, o un objeto invocable que devuelve una instancia `TestCase` o `TestSuite`. Estas comprobaciones se aplican en el orden que se indica aquí; es decir, un método en una posible clase de caso de test se recogerá como «un método de test dentro de una clase de caso de test", en lugar de «un objeto invocable".

Por ejemplo, si tiene un módulo `SampleTests` que contiene una clase derivada de `TestCase` `SampleTestCase` con tres métodos de test (`test_one()`, `test_two()`, y `test_three()`), el especificador `SampleTests.SampleTestCase'` haría que este método devolviera una suite que ejecutara los tres métodos de prueba. El uso del especificador `SampleTests.SampleTestCase.test_two'` provocaría que este método devolviera una suite de tests que ejecutaría sólo el método de test `test_two()`. El especificador puede referirse a los módulos y paquetes que no han sido importados; serán importados como un efecto secundario.

El método opcionalmente resuelve *name* relativo al *module* dado.

Distinto en la versión 3.5: Si un `ImportError` o `AttributeError` ocurre mientras atraviesa *name* entonces se devolverá un test sintético que lanza ese error cuando se ejecuta. Estos errores están incluidos en los errores acumulados por self.errors.

loadTestsFromNames(*names*, *module=None*)

Similar a `loadTestsFromName()`, pero toma una secuencia de nombres en lugar de un solo nombre. El valor de retorno es una suite de tests que soporta todos los test definidos para cada nombre.

getTestCaseNames(*testCaseClass*)

Devuelve una secuencia ordenada de nombres de métodos encontrados dentro de *testCaseClass*; esta debería ser una subclase de `TestCase`.

discover(*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)

Encuentra todos los módulos de prueba recurriendo a subdirectorios del directorio de inicio especificado, y retorna un objeto de TestSuite que los contenga. Sólo se cargarán los archivos de test que coincidan con el *pattern*. (Utilizando la coincidencia de patrones de estilo de shell.) Sólo se cargarán los nombres de los módulos que sean importables (es decir, que sean identificadores Python válidos).

Todos los módulos de test deben ser importables desde el nivel superior del proyecto. Si el directorio de inicio no es el directorio de nivel superior, entonces el directorio de nivel superior debe ser especificado por separado.

Si la importación de un módulo falla, por ejemplo debido a un error de sintaxis, entonces esto se registrará como un error único y el descubrimiento continuará. Si el fallo en la importación se debe a que `SkipTest` se ha lanzado, se registrará como un salto en lugar de un error.

Si se encuentra un paquete (un directorio que contiene un archivo llamado `__init__.py`), se comprobará si el paquete tiene una función `load_tests`. Si existe, entonces se invocará `package.load_tests(loader, tests, pattern)`. Test discovery se encarga de asegurar que un paquete sólo se comprueba una vez durante una invocación, incluso si la propia función load_tests llama a `loader.discover`.

Si `load_tests` existe, entonces el descubrimiento *no* recurre en el paquete, `load_tests` es responsable de cargar todos los tests en el paquete.

El patrón no se almacena deliberadamente como atributo cargador para que los paquetes puedan continuar descubriéndose a sí mismos. *top_level_dir* se almacena de forma que `load_tests` no necesita pasar este argumento a `loader.discover()`.

start_dir puede ser un nombre de módulo punteado así como un directorio.

Nuevo en la versión 3.2.

Distinto en la versión 3.4: Los módulos que lanzan *SkipTest* en la importación se registran como saltos, no como errores.

Distinto en la versión 3.4: *start_dir* puede ser un *paquete de espacios de nombres*.

Distinto en la versión 3.4: Las rutas se ordenan antes de ser importadas para que el orden de ejecución sea el mismo, incluso si el orden del sistema de archivos subyacente no depende del nombre del archivo.

Distinto en la versión 3.5: Los paquetes encontrados son ahora comprobados para `load_tests` sin importar si su ruta coincide con el *pattern*, porque es imposible que el nombre de un paquete coincida con el patrón por defecto.

Los siguientes atributos de un *TestLoader* pueden ser configurados ya sea por subclasificación o asignación en una instancia:

testMethodPrefix
Cadena que da el prefijo de los nombres de métodos que serán interpretados como métodos de test. El valor por defecto es `'test'`.

Esto afecta a *getTestCaseNames()* y a todos los métodos `loadTestsFrom*()` .

sortTestMethodsUsing
Función que se utiliza para comparar los nombres de los métodos al clasificarlos en *getTestCaseNames()* y todos los métodos `loadTestsFrom*()`.

suiteClass
Objeto invocable que construye un conjunto de pruebas a partir de una lista de pruebas. No se necesitan métodos en el objeto resultante. El valor por defecto es la clase *TestSuite*.

Esto afecta a todos los métodos `loadTestsFrom*()`.

testNamePatterns
List of Unix shell-style wildcard test name patterns that test methods have to match to be included in test suites (see `-k` 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 `-k` option, simple substring patterns will have to be converted using `*` wildcards.

Esto afecta a todos los métodos `loadTestsFrom*()`.

Nuevo en la versión 3.7.

class unittest.**TestResult**
Esta clase se utiliza para recopilar información sobre qué tests han tenido éxito y cuáles han fracasado.

Un objeto *TestResult* almacena los resultados de una serie de pruebas. Las clases *TestCase* y *TestSuite* aseguran que los resultados se registren correctamente; los autores de los tests no tienen que preocuparse de registrar el resultado de las mismas.

Los marcos de pruebas construidos sobre *unittest* pueden querer acceder al objeto *TestResult* generado por la ejecución de un conjunto de tests con fines de reporte; una instancia *TestResult* es devuelta por el método `TestRunner.run()` para este propósito.

Las instancias de *TestResult* tienen los siguientes atributos que serán de interés cuando se inspeccionen los resultados de la ejecución de un conjunto de tests:

errors
Una lista que contiene 2 tuplas de instancias *TestCase* y cadenas con formato de tracebacks. Cada tupla representa una prueba que lanzó una excepción inesperada.

failures
Una lista que contiene 2 tuplas de instancias *TestCase* y cadenas con formato de traceback. Cada tupla representa un test en el que un fallo fue explícitamente señalado usando los métodos `TestCase.assert*()`.

skipped
Una lista que contiene 2 tuplas de instancias de *TestCase* y cadenas que contienen la razón para saltarse el test.

Nuevo en la versión 3.1.

expectedFailures
Una lista que contiene 2 tuplas de instancias *TestCase* y cadenas con formato de traceback. Cada tupla representa un fallo esperado del caso de test.

unexpectedSuccesses
Una lista que contiene instancias de *TestCase* que fueron marcadas como fracasos esperados, pero tuvieron éxito.

shouldStop
Puesto en `True` cuando la ejecución de los tests se detenga por *stop()*.

testsRun
El número total de tests realizados hasta ahora.

buffer
Si se ajusta a true, `sys.stdout` y `sys.stderr` serán almacenados entre *startTest()* y *stopTest()* siendo llamados. La salida recolectada sólo tendrá eco en el verdadero `sys.stdout` y `sys.stderr` si la prueba falla o se equivoca. Cualquier salida también se adjunta al mensaje de fallo / error.

Nuevo en la versión 3.2.

failfast
Si se ajusta a true *stop()* se llamará al primer fallo o error, deteniendo la ejecución de la prueba.

Nuevo en la versión 3.2.

tb_locals
Si se ajusta a true entonces las variables locales se mostrarán en los tracebacks.

Nuevo en la versión 3.5.

wasSuccessful()
Devuelve `True` si todas las pruebas realizadas hasta ahora han pasado, de lo contrario devuelve `False`.

Distinto en la versión 3.4: Devuelve `False` si hubo algún *unexpectedSuccesses* de las pruebas marcadas con el decorador *expectedFailure()*.

stop()
Este método puede ser llamado para señalar que el conjunto de pruebas que se están ejecutando debe ser abortado poniendo el atributo *shouldStop* en `True`. `TestRunner` los objetos deben respetar esta bandera y regresar sin ejecutar ninguna prueba adicional.

Por ejemplo, esta característica es utilizada por la clase *TextTestRunner* para detener el marco de pruebas cuando el usuario señala una interrupción desde el teclado. Las herramientas interactivas que proporcionan implementaciones de `TestRunner` pueden usar esto de manera similar.

Los siguientes métodos de la clase *TestResult* se utilizan para mantener las estructuras de datos internos, y pueden ampliarse en subclases para apoyar los requisitos de información adicionales. Esto es particularmente útil para construir herramientas que apoyen la presentación de informes interactivos mientras se ejecutan las pruebas.

startTest(*test*)
Llamado cuando el caso de prueba *test* está a punto de ser ejecutado.

stopTest(*test*)
Llamado después de que el caso de prueba *prueba* haya sido ejecutado, independientemente del resultado.

startTestRun()
Llamado una vez antes de que se ejecute cualquier prueba.

Nuevo en la versión 3.1.

stopTestRun()
Llamado una vez después de que se ejecuten todas las pruebas.

Nuevo en la versión 3.1.

addError(*test*, *err*)
Llamado cuando el caso de prueba *test* plantea una excepción inesperada. *err* es una tupla de la forma devuelta por *sys.exc_info()*: `(type, value, traceback)`.

La implementación por defecto añade una tupla `(test, formatted_err)` al atributo *errors* de la instancia, donde *formatted_err* es una traza formateada derivada de *err*.

addFailure(*test*, *err*)
Llamado cuando el caso de prueba *test* señala un fallo. *err* es una tupla de la forma devuelta por *sys.exc_info()*: `(type, value, traceback)`.

La implementación por defecto añade una tupla `(test, formatted_err)` al atributo *failures* de la instancia, donde *formatted_err* es una traza formateada derivada de *err*.

addSuccess(*test*)
Llamado cuando el caso de prueba *test* tenga éxito.

La implementación por defecto no hace nada.

addSkip(*test*, *reason*)
Llamado cuando se salta el caso de prueba *test*. *reason* es la razón que la prueba dio para saltarse.

La implementación por defecto añade una tupla `(test, reason)` al atributo *skipped* de la instancia.

addExpectedFailure(*test*, *err*)
Llamó cuando el caso de prueba *test* falla, pero fue marcado con el decorador *expectedFailure()*.

La implementación por defecto añade una tupla `(test, formatted_err)` al atributo *expectedFailures* de la instancia, donde *formatted_err* es una traza formateada derivada de *err*.

addUnexpectedSuccess(*test*)
Llamó cuando el caso de prueba *test* se marcó con el decorador *expectedFailure()*, pero tuvo éxito.

La implementación por defecto añade la prueba al atributo *unexpectedSuccesses* de la instancia.

addSubTest(*test*, *subtest*, *outcome*)
Llamado cuando termina una subtest. *test* es el caso de prueba correspondiente al método de test. *subtest* es una instancia personalizada *TestCase* que describe el test.

Si *outcome* es *None*, el subtest tuvo éxito. De lo contrario, falló con una excepción en la que *outcome* es una tupla de la forma devuelta por *sys.exc_info()*: `(type, value, traceback)`.

La implementación por defecto no hace nada cuando el resultado es un éxito, y registra los fallos del subtest como fallos normales.

Nuevo en la versión 3.4.

class unittest.**TextTestResult**(*stream*, *descriptions*, *verbosity*)
Una implementación concreta de *TestResult* utilizado por el *TextTestRunner*.

Nuevo en la versión 3.2: Esta clase se llamaba anteriormente `_TextTestResult`. El antiguo nombre todavía existe como un alias pero está obsoleto.

unittest.**defaultTestLoader**
Instancia de la clase *TestLoader* destinada a ser compartida. Si no es necesario personalizar la clase *TestLoader*, esta instancia puede utilizarse en lugar de crear repetidamente nuevas instancias.

class unittest.**TextTestRunner**(*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *warnings=None*, ***, *tb_locals=False*)
Una implementación básica del test runner que produce resultados en una corriente. Si *stream* es None, el valor por defecto, *sys.stderr* se utiliza como flujo de salida. Esta clase tiene unos pocos parámetros configurables, pero es esencialmente muy simple. Las aplicaciones gráficas que ejecutan las suites de prueba deben proporcionar implementaciones alternativas. Tales implementaciones deberían aceptar **kwargs como interfaz para construir los cambios de los corredores cuando se añaden características a unittest.

Por defecto este runner muestra *DeprecationWarning*, *PendingDeprecationWarning*, *ResourceWarning* y *ImportWarning* aunque estén *ignorados por defecto*. Las advertencias de deprecación causadas por *métodos deprecated unittest* también tienen un caso especial y, cuando los filtros de advertencia están 'default' o 'always', aparecerán sólo una vez por módulo, para evitar demasiados mensajes de advertencia. Este comportamiento puede ser anulado usando las opciones -Wd o -Wa de Python (ver Control de advertencias) y dejando *warnings* a None.

Distinto en la versión 3.2: Añadió el argumento warnings.

Distinto en la versión 3.2: El flujo por defecto está configurado como *sys.stderr* en tiempo de instanciación en lugar de tiempo de importación.

Distinto en la versión 3.5: Añadido el parámetro tb_locals.

_makeResult()
Este método devuelve la instancia de TestResult usada por *run()*. No está destinado a ser llamado directamente, pero puede ser anulado en subclases para proporcionar un TestResult personalizado.

_makeResult() instanciando la clase o el pasaje llamado en el constructor TextTestRunner como el argumento de resultclass. Por defecto es *TextTestResult* si no se proporciona ninguna resultclass. La clase de resultado se instanciará con los siguientes argumentos:

```
stream, descriptions, verbosity
```

run(*test*)
Este método es la principal interfaz pública del TextTestRunner. Este método toma una instancia *TestSuite* o *TestCase*. Se crea una *TestResult* llamando a *_makeResult()* y se ejecuta(n) la(s) prueba(s) y se imprimen los resultados a 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*)
Un programa de línea de comandos que carga un conjunto de pruebas de *módulo* y las ejecuta; esto es principalmente para hacer los módulos de prueba convenientemente ejecutables. El uso más simple de esta función es incluir la siguiente línea al final de un guión de prueba:

```
if __name__ == '__main__':
    unittest.main()
```

Puedes hacer pruebas con información más detallada pasando el argumento de la verbosity:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

El argumento *defaultTest* es el nombre de una prueba única o un iterable de nombres de pruebas a ejecutar si no se especifican nombres de pruebas a través de *argv*. Si no se especifica o Ninguno y no se proporcionan nombres de pruebas vía *argv*, se ejecutan todas las pruebas encontradas en *modulo*.

El argumento *argv* puede ser una lista de opciones pasadas al programa, siendo el primer elemento el nombre del programa. Si no se especifica o Ninguno, se utilizan los valores de *sys.argv*.

El argumento *testRunner* puede ser una clase de corredor de prueba o una instancia ya creada de él. Por defecto `main` llama *sys.exit()* con un código de salida que indica el éxito o el fracaso de la ejecución de las pruebas.

El argumento *testLoader* tiene que ser una instancia *TestLoader*, y por defecto *defaultTestLoader*.

`main` apoya el uso del intérprete interactivo pasando el argumento `exit=False`. Esto muestra el resultado en la salida estándar sin llamar a *sys.exit()*:

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

Los parámetros *failfast*, *catchbreak* y *buffer* tienen el mismo efecto que las *command-line options* del mismo nombre.

El argumento *warnings* especifica el *filtro de aviso* que debe ser usado mientras se realizan los tests. Si no se especifica, permanecerá como `None` si se pasa una opción `-W` a **python** (ver Warning control), de lo contrario se establecerá como `'default'`.

Invocar `main` en realidad devuelve una instancia de la clase `TestProgram`. Esto almacena el resultado de las pruebas ejecutadas como el atributo `result`.

Distinto en la versión 3.1: El parámetro *exit* fue añadido.

Distinto en la versión 3.2: Los parámetros *verbosity*, *failfast*, *catchbreak*, *buffer* y *warnings* fueron añadidos.

Distinto en la versión 3.4: El parámetro *defaultTest* fue cambiado para aceptar también un iterable de nombres de pruebas.

load_tests protocolo

Nuevo en la versión 3.2.

Los módulos o paquetes pueden personalizar la forma en que se cargan las pruebas a partir de ellos durante las ejecuciones de prueba normales o el descubrimiento de pruebas mediante la implementación de una función llamada `load_tests`.

Si un módulo de tests define `load_tests` será llamado por *TestLoader.loadTestsFromModule()* con los siguientes argumentos:

```
load_tests(loader, standard_tests, pattern)
```

donde *pattern* se pasa directamente desde `loadTestsFromModule`. Por defecto es `None`.

Debe retornar una *TestSuite*.

loader es la instancia de *TestLoader* haciendo la carga. *standard_tests* son los tests que se cargarían por defecto desde el módulo. Es común que los módulos de test sólo quieran añadir o quitar tests del conjunto de tests estándar. El tercer argumento se usa cuando se cargan paquetes como parte del descubrimiento de tests.

Una típica función de `load_tests` que carga pruebas de un conjunto específico de `TestCase`class:`TestCase` puede ser como:

```
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
```

Si discovery se inicia en un directorio que contiene un paquete, ya sea desde la línea de comandos o llamando a *TestLoader.discover()*, entonces el paquete `__init__.py` se comprobará por `load_tests`. Si esa

función no existe, discover se reincorporará al paquete como si fuera un directorio más. De lo contrario, el descubrimiento de los tests del paquete se dejará en `load_tests` que se llama con los siguientes argumentos:

```
load_tests(loader, standard_tests, pattern)
```

Esto debería devolver un *TestSuite* que represente todas las pruebas del paquete. (`test_estándar` sólo contendrá las pruebas recogidas de `__init__.py`.)

Debido a que el patrón se pasa a `load_tests` el paquete es libre de continuar (y potencialmente modificar) el descubrimiento de pruebas. Una función de "no hace nada" `load_test` para un paquete de pruebas se vería como:

```
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 ya no comprueba si los nombres de los paquetes coinciden con el *patrón* debido a la imposibilidad de que los nombres de los paquetes coincidan con el patrón por defecto.

26.8.9 Instalaciones para clases y módulos

Los accesorios de nivel de clase y módulo se implementan en *TestSuite*. Cuando el conjunto de pruebas se encuentra con una prueba de una nueva clase entonces se llama `tearDownClass()` de la clase anterior (si existe), seguido de `setUpClass()` de la nueva clase.

Del mismo modo, si una prueba es de un módulo diferente de la prueba anterior, entonces se ejecuta `DesmontarMódulo` del módulo anterior, seguido de `DesmontarMódulo` del nuevo módulo.

Después de todas las pruebas, se ejecutan los últimos `tearDownClass` y `tearDownModule`.

Tenga en cuenta que los accesorios compartidos no juegan bien con las características [potenciales] como la paralelización de la prueba y rompen el aislamiento de la prueba. Deben ser usados con cuidado.

El orden por defecto de las pruebas creadas por los cargadores de pruebas unitarias es agrupar todas las pruebas de los mismos módulos y clases. Esto llevará a que `setUpClass` / `setUpModule` (etc) sea llamado exactamente una vez por clase y módulo. Si se aleatoriza el orden, de manera que las pruebas de diferentes módulos y clases sean adyacentes entre sí, entonces estas funciones compartidas de fixture pueden ser llamadas varias veces en una sola ejecución de prueba.

Los accesorios compartidos no están pensados para trabajar con suites con pedidos no estándar. Todavía existe una `BaseTestSuite` para los marcos de trabajo que no quieren soportar accesorios compartidos.

Si hay alguna excepción planteada durante una de las funciones compartidas del aparato, la prueba se notifica como un error. Debido a que no hay una instancia de prueba correspondiente, se crea un objeto `_ErrorHolder` (que tiene la misma interfaz que un *TestCase*) para representar el error. Si sólo estás usando el standard unittest test runner entonces este detalle no importa, pero si eres un autor de marcos de trabajo puede ser relevante.

setUpClass y tearDownClass

Estos deben ser implementados como métodos de clase:

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
```

(continué en la próxima página)

(proviene de la página anterior)

```
    def tearDownClass(cls):
        cls._connection.destroy()
```

Si quieres que se invoque a `SetUpClass` y `BreakdownClass` en clases base, debes llamarlos tú mismo. Las implementaciones en *`TestCase`* están vacías.

Si se lanza una excepción durante una `setUpClass`, entonces los tests de la clase no se ejecutan y la `tearDownClass` no se ejecuta. Las clases que se salten no tendrán `setUpClass` o `tearDownClass`. Si la excepción es una *`SkipTest`* entonces la clase será reportada como salteada en lugar de como un error.

setUpModule y tearDownModule

Estos deben ser implementados como funciones:

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

Si se lanza una excepción en un `setUpModule`, entonces no se ejecutará ninguna de las pruebas del módulo y no se ejecutará el `tearDownModule`. Si la excepción es una *`SkipTest`* entonces el módulo será reportado como saltado en lugar de como un error.

Para agregar código de limpieza que se debe ejecutar incluso en el caso de una excepción, utilice `addModuleCleanup`:

`unittest.`**`addModuleCleanup`**(*function*, */*, **args*, ***kwargs*)

Añade una función que se llamará después de `tearDownModule()` para limpiar los recursos utilizados durante la clase de test. Las funciones se llamarán en orden inverso al orden en que se agregan (LIFO). Se llaman con cualquier argumento y palabra clave que se pase a *`addModuleCleanup()`* cuando se añadan.

Si `setUpModule()` falla, lo que significa que `tearDownModule()` no se invoca, entonces cualquier función de limpieza añadida seguirá siendo invocada.

Nuevo en la versión 3.8.

`unittest.`**`doModuleCleanups`**()

Esta función se llama incondicionalmente después de `tearDownModule()`, o después de `setUpModule()` si `setUpModule()` lanza una excepción.

It is responsible for calling all the cleanup functions added by *`addModuleCleanup()`*. If you need cleanup functions to be called *prior* to `tearDownModule()` then you can call *`doModuleCleanups()`* yourself.

`doModuleCleanups()` saca los métodos de la pila de funciones de limpieza uno a uno, así que se puede llamar en cualquier momento.

Nuevo en la versión 3.8.

26.8.10 Manejo de señales

Nuevo en la versión 3.2.

La opción *`-c/--catch`* línea de comando para unittest, junto con el parámetro `catchbreak` de *`unittest.main()`*, proporcionan un manejo más amigable del control-C durante una prueba. Con el comportamiento catch break habilitado, control-C permitirá que se complete la prueba que se está ejecutando actualmente, y la ejecución de la prueba terminará y reportará todos los resultados hasta ahora. Un segundo control-C lanzará una *`KeyboardInterrupt`* de la manera habitual.

El manejador de señales de manejo de control-c intenta permanecer compatible con el código o las pruebas que instalan su propio manejador *`signal.SIGINT`* . Si se llama al manejador `unittest` pero *no es* el manejador

signal.SIGINT instalado, es decir, ha sido reemplazado por el sistema bajo test y delegado, entonces llama al manejador por defecto. Este será normalmente el comportamiento esperado por el código que reemplaza un manejador instalado y delega en él. Para las pruebas individuales que necesiten el manejo de control-c de `unittest` deshabilitado se puede usar el decorador *removeHandler()*.

Hay algunas funciones de utilidad para que los autores de marcos de trabajo habiliten la funcionalidad de control de control-c dentro de los marcos de prueba.

`unittest.`**`installHandler`**`()`
Instala el controlador de control-c. Cuando se recibe una *signal.SIGINT* (normalmente en respuesta a que el usuario presione control-c) todos los resultados registrados tienen *stop()* llamado.

`unittest.`**`registerResult`**`(`*result*`)`
Registrar un objeto *TestResult* para el manejo de control-c. El registro de un resultado almacena una referencia débil a él, por lo que no evita que el resultado sea recogido por el recolector de basura.

El registro de un objeto *TestResult* no tiene efectos secundarios si el manejo de control-c no está habilitado, por lo que los marcos de pruebas pueden registrar incondicionalmente todos los resultados que crean independientemente de si el manejo está habilitado o no.

`unittest.`**`removeResult`**`(`*result*`)`
Elimine un resultado registrado. Una vez que un resultado ha sido eliminado, *stop()* ya no se llamará en ese objeto de resultado en respuesta a un control-c.

`unittest.`**`removeHandler`**`(`*function=None*`)`
Cuando se llama sin argumentos, esta función quita el gestor control-c si se ha instalado. Esta función también se puede utilizar como decorador de tests para quitar temporalmente el controlador mientras se ejecuta el test:

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

26.9 `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 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.

26.9.1 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 nor-

malmente 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='wheeeeee')
>>> str(mock)
'wheeeeee'
```

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__`.

26.9.2 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 interaccionado 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*)

Crea 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 lanzará 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 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, acceder a cualquier atributo con un nombre que empiece con *assert*, *assret*, *asert*, *aseert* o *assrt* lanzará un `AttributeError`. Pasar `unsafe=True` permitirá acceder 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

[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*.

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()
Assert cuando 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()
Assert 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*)
Assert 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*)
assert 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 solo 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*)

assert 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()

Assert 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: Added two keyword-only arguments to the reset_mock function.

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*, and `side_effect` are keyword-only arguments.

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. Solo 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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
  ...
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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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: Propiedades `args` y `kwargs` agregadas.

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()
<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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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 desempaquetarlo 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
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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 sincrónicas 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()

Assert 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()

Assert 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):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_awaited_with(**args*, ***kwargs*)

Assert 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')
```

(continué en la próxima página)

(proviene de la página anterior)

```
Traceback (most recent call last):
...
AssertionError: expected call not found.
Expected: mock('other')
Actual: mock('foo', bar='bar')
```

assert_awaited_once_with(**args*, ***kwargs*)

Assert 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*)

Assert 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*)

Assert 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):
...     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()

Assert 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)
...
>>> 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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:
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     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')]
```

26.9.3 Parcheadores

Los decoradores patch se utilizan para parchear los objetos solo dentro del ámbito de la función que decoran. Se encargan automáticamente de desparchear 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 clave es realizar el parche en el mismo espacio de nombre. Para más detalles consultar la sección *where to patch*.

---

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 *AsyncMock* si el objeto parcheado es asincrónico, si es de otra forma *MagicMock* o a un *new_callable* si es especificado.

`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
...     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 si mismos del objeto simulado padre creado. Eso si, 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 si que puede ser expandido en una llama a *patch()* usando `**`:

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> 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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
  ...
AttributeError: <module 'sys' (built-in)> does not have the attribute 'non_
↪existing_attribute'
```

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'}):
...     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. La 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. Solo 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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

Both *patch* and *patch.object* correctly patch and restore descriptors: class methods, static methods and properties. You should patch these on the *class* rather than an instance. They also work with *some* objects that proxy attribute access, like the django settings object.

26.9.4 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 normales[2], este soporte ha sido especialmente implementado. Esto significa que solo 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 háznoslo 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 argumento[3].

```
>>> 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 si se registran en *mock_calls*.

[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.

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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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__`

26.9.5 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
>>> result
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 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 solo 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()* (solo 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'),
```

(continué en la próxima página)

(proviene de la página anterior)

```
 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 solo 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)  # Intentional typo!
```

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  # Intentional typo!
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

La especificación solo 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()  # Intentional typo!
```

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 anidadas (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
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 lanzará el error correcto:

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with  # Intentional typo!
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_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 segura[4].

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
...
>>> 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
...
```

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 solo estás estableciendo atributos predeterminados en `__init__()`, proporcionarlos a través de atributos de clase (que , por supuesto, son compartidos entre instancias) también es más rápido.

[4] Esto solo 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()*).

```
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):
...     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.

26.10 unittest.mock — primeros pasos

Nuevo en la versión 3.3.

26.10.1 Usando Mock

Métodos de parcheo Mock

Usos comunes para objetos *Mock* incluye:

- Métodos de parcheo
- Métodos de grabación de llamadas sobre objetos

Es posible que desee reemplazar un método en un objeto para comprobar que se llama con los argumentos correctos por otra parte del sistema:

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

Una vez que se ha utilizado nuestro mock (`real.method` en este ejemplo) tiene métodos y atributos que le permiten hacer afirmaciones sobre cómo se ha utilizado.

Nota: En la mayoría de estos ejemplos, las clases *Mock* y *MagicMock* son intercambiables. Como el `MagicMock` es la clase más capaz, hace que sea sensato usarlo por defecto.

Una vez que el mock ha sido llamado su atributo *called* se establece en `True`. Lo que es más importante, podemos usar el método *assert_called_with()* o *assert_called_once_with()* para comprobar que se llamó con los argumentos correctos.

En este ejemplo se prueba que llamar a `ProductionClass().method` da como resultado una llamada al método `something`:

```
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

Mock de llamadas a métodos sobre un objeto

En el último ejemplo, parcheamos un método directamente en un objeto para comprobar que se llamó correctamente. Otro caso de uso común es pasar un objeto a un método (o a alguna parte del sistema sometido a prueba) y, a continuación, comprobar que se utiliza de la manera correcta.

La sencilla `ProductionClass` a continuación tiene un método `closer`. Si se llama con un objeto, entonces llama a `close` en él.

```
>>> class ProductionClass:
...     def closer(self, something):
```

(continué en la próxima página)

(proviene de la página anterior)

```
...         something.close()
...
```

Así que para probarlo necesitamos pasar un objeto con un método `close` y comprobar que se llamó correctamente.

```
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

No tenemos que hacer ningún trabajo para proporcionar el método de "close" en nuestro mock. El acceso al cierre lo crea. Por lo tanto, si "close" aún no se ha llamado, entonces acceder a él en la prueba lo creará, pero *assert_called_with()* lanzará una excepción de error.

Clases Mocking

Un caso de uso común es simular las clases que crea la instancia del código que se está probando. Cuando se aplica un parche a una clase, esa clase se reemplaza por un mock. Las instancias se crean *llamando a la clase*. Esto significa que tiene acceso a la «instancia mock» examinando el valor devuelto de la clase simulada.

En el ejemplo siguiente tenemos una función `some_function` que crea una instancia de `Foo` y llama a un método en él. La llamada a *patch()* reemplaza la clase `Foo` con un mock. La instancia `Foo` es el resultado de llamar al mock, por lo que se configura modificando el mock `-Mock.return_value`.

```
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

Nombrando tus mocks

Puede ser útil poner un nombre a tus mocks. El nombre se muestra en la reproducción del mock y puede ser útil cuando el mock aparece en los mensajes de error de la prueba. El nombre también se propaga a los atributos o métodos del mock:

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

Siguiendo todas las llamadas

A menudo, desea realizar un seguimiento de más de una llamada a un método. El atributo `mock_calls` registra todas las llamadas a los atributos secundarios del mock, y también a sus hijos.

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

Si realiza una afirmación sobre `mock_calls` y se ha llamado a cualquier método inesperado, la aserción fallará. Esto es útil porque además de afirmar que se han realizado las llamadas que esperaba, también está comprobando que se hicieron en el orden correcto y sin llamadas adicionales:

Utiliza el objeto *call* para construir listas y compararlas con `mock_calls`:

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

Sin embargo, los parámetros de las llamadas que devuelven mocks no se registran, lo que significa que no es posible realizar un seguimiento de las llamadas anidadas donde los parámetros utilizados para crear ancestros son importantes:

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

Establecer valores de retorno y atributos

Establecer los valores de retorno en un objeto mock es sumamente fácil:

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

Por supuesto, puede hacer lo mismo con los métodos en el mock:

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

El valor devuelto también se puede establecer en el constructor:

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

Si necesitas una configuración de atributo en su mock, simplemente haga:

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

A veces desea simular una situación más compleja, como por ejemplo `mock.connection.cursor().execute("SELECT 1")`. Si esperamos que esta llamada devuelva una lista, entonces tenemos que configurar el resultado de la llamada anidada.

Podemos usar *call* para construir el conjunto de llamadas en una «llamada encadenada» como esta para una fácil afirmación después:

```
>>> 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
```

Es la llamada a `.call_list()` la que convierte nuestro objeto de llamada en una lista de llamadas que representan las llamadas encadenadas.

Generar excepciones con mocks

Un atributo útil es *`side_effect`*. Si establece esto en una clase o instancia de excepción, se producirá la excepción cuando se llame al mock.

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Funciones de efectos secundarios e iterables

`side_effect` también se puede asignar en una función o en una iterable. El caso de uso para `side_effect` como un iterable es donde se va a llamar a su mock varias veces, y desea que cada llamada devuelva un valor diferente. Cuando se asigna `side_effect` en un iterable cada llamada al mock devuelve el siguiente valor de lo iterable:

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

Para casos de uso más avanzados, como variar dinámicamente los valores devueltos en función de cómo se llame al mock, `side_effect` puede ser una función. Se llamará a la función con los mismos argumentos que el mock. Lo que sea que la función devuelve es lo que devuelve la llamada:

```
>>> 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
```

Iteradores asincrónicos de Mocking

Desde Python 3.8, `AsyncMock` y `MagicMock` tienen soporte para mock async-iterators through `__aiter__`. El *return_value* atributo de `__aiter__` puede ser usado para asignar los valores de retorno que podrían ser usados por iteración.

```
>>> 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]
```

El gestor de contexto asincrónico de Mocking

Desde Python 3.8, `AsyncMock` y `MagicMock` tienen soporte para mock async-context-managers a través de `__aenter__` y `__aexit__`. De forma predeterminada, las instancias `__aenter__` y `__aexit__` son instancias de `AsyncMock` que devuelven una función asincrónica.

```
>>> 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
...
>>> asyncio.run(main())
>>> mock_instance.__aenter__.assert_awaited_once()
>>> mock_instance.__aexit__.assert_awaited_once()
```

Creando un mock desde un objeto existente

Un problema con el uso excesivo de mocking es que combina sus pruebas a la implementación de sus mocks en lugar de su código real. Supongamos que tiene una clase que implementa `some_method`. En una prueba para otra clase, se proporciona un mock de este objeto que *also* proporciona `some_method`. Si más tarde refactoriza la primera clase, para que ya no tenga `some_method` - entonces sus pruebas seguirán pasando a pesar de que su código está ahora roto!

Mock le permite proporcionar un objeto como especificación para el mock, utilizando el argumento de palabra clave *spec*. El acceso a métodos / atributos en el mock que no existen en el objeto de especificación lanzará inmediatamente un error de atributo. Si cambia la implementación de la especificación, las pruebas que usan esa clase comenzarán a fallar inmediatamente sin tener que crear instancias de la clase en esas pruebas.

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
   ...
AttributeError: object has no attribute 'old_method'
```

El uso de una especificación también permite una coincidencia más inteligente de las llamadas realizadas al mock, independientemente de si algunos parámetros se pasaron como argumentos posicionales o con nombre:

```
>>> 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)
```

Si desea que esta coincidencia más inteligente también funcione con llamadas de método en el mock, puede usar *auto-speccing*.

Si desea una forma más fuerte de especificación que impida la configuración de atributos arbitrarios, así como la obtención de ellos, entonces puede usar *spec_set* en lugar de *spec*.

26.10.2 Decoradores de Parches

Nota: Con `patch()` importa que parchee objetos en el espacio de nombres donde se buscan. Esto es normalmente sencillo, pero para una guía rápida lea *where to patch*.

Una necesidad común en las pruebas es aplicar revisiones a un atributo de clase o a un atributo de módulo, por ejemplo, aplicar revisiones a una clase integrada o parchear una clase en un módulo para probar que se crea una instancia. Los módulos y las clases son efectivamente globales, por lo que el parcheo en ellos tiene que deshacerse después de la prueba o el parche persistirá en otras pruebas y causará problemas difíciles de diagnosticar.

mock proporciona tres decoradores convenientes para esto: `patch()`, `patch.object()` y `patch.dict()`. `patch` toma una sola cadena del formulario `package.module.Class.attribute` para especificar el atributo que está parcheando. También, opcionalmente, toma un valor con el que desea que se reemplace el atributo (o clase o lo que sea). "patch.object" toma un objeto y el nombre del atributo con el que desea parchear, además, opcionalmente, del valor con el que parcharlo.

`patch.object`:

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... 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()
```

Si está parcheando un módulo (incluyendo `builtins`) entonces use `patch()` en lugar de `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"
```

EL nombre del módulo puede ser "dotted", en el formulario `package.module` si es necesario:

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

Un buen patrón en realidad es decorar los métodos de pruebas propios:

```
>>> 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
```

Si desea parchear con un Mock, puede usar *patch()* con un solo argumento (o *patch.object()* con dos argumentos). El mock se creará para usted y se pasará a la función de prueba / método:

```
>>> 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()
```

Puede apilar varios decoradores de parches utilizando este patrón:

```
>>> 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)
...
>>> MyTest('test_something').test_something()
```

Al anidar decoradores de parches, los mocks se pasan a la función decorada en el mismo orden en que se aplicaron (el orden normal *Python* que se aplican los decoradores). Esto significa de abajo hacia arriba, así que en el ejemplo anterior el simulacro de `test_module.ClassName2` se pasa primero.

También está *patch.dict()* para establecer valores en un diccionario solo durante 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
```

`patch`, `patch.object` and `patch.dict` se pueden utilizar como gestores de contexto.

Donde utilice *patch()* para crear un simulacro para usted, puede obtener una referencia al simulacro utilizando la forma «as» de la instrucción with:

```
>>> class ProductionClass:
...     def method(self):
...         pass
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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)
```

Como alternativa `patch`, `patch.object` and `patch.dict` se pueden utilizar como decoradores de clase. Cuando se utiliza de esta manera es lo mismo que aplicar el decorador individualmente a cada método cuyo nombre comienza con «test».

26.10.3 Otros Ejemplos

Estos son algunos ejemplos más para algunos escenarios ligeramente más avanzados.

Mocking de llamadas encadenadas

Mocking de las llamadas encadenadas es en realidad sencillo con mock una vez que entiende el atributo *`return_value`* . Cuando se llama a un mock por primera vez, o se obtiene su `return_value` antes de que se llame, se crea un nuevo *`Mock`*.

Esto significa que puede ver cómo se ha utilizado el objeto devuelto de una llamada a un objeto simulado interrogando el simulado `return_value`:

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

Desde aquí es un paso simple para configurar y luego hacer aserciones sobre llamadas encadenadas. Por supuesto, otra alternativa es escribir su código de una manera más comprobable en primer lugar…

Por lo tanto, supongamos que tenemos algún código que se ve un poco como este:

```
>>> 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
```

Suponiendo que `BackendProvider` ya está bien probado, ¿cómo probamos `method()` ? En concreto, queremos probar que la sección de código `# more code` usa el objeto de respuesta de la manera correcta.

A medida que esta cadena de llamadas se realiza a partir de un atributo de instancia, podemos parchear el atributo `backend` en una instancia de `Something`. En este caso en particular sólo estamos interesados en el valor devuelto de la llamada final a `start_call` por lo que no tenemos mucha configuración que hacer. Supongamos que el objeto que devuelve es "similar a un archivo", por lo que nos aseguraremos de que nuestro objeto de respuesta utilice el compilado *`open()`* as its `spec`.

Para ello creamos una instancia mock como nuestro back-end simulado y creamos un objeto de respuesta mock para ella. Para establecer la respuesta como el valor devuelto de ese `start_call` final podríamos hacer lo siguiente:

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_
↪value = mock_response
```

Podemos hacerlo de una manera un poco mejor usando el método *`configure_mock()`* para establecer directamente el valor devuelto para nosotros:

```
>>> 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)
```

Con estos monkey patch el «backend simulado» en su lugar y puede hacer la llamada real:

```
>>> something.backend = mock_backend
>>> something.method()
```

Usando *mock_calls* podemos comprobar la llamada encadenada con una sola afirmación. Una llamada encadenada es varias llamadas en una línea de código, por lo que habrá varias entradas en `mock_calls`. Podemos usar *call.call_list()* para crear esta lista de llamadas para nosotros:

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

Mocking parcial

En algunas pruebas quería un mock de una llamada a *datetime.date.today()* para devolver una fecha conocida, pero no quería evitar que el código sometido a prueba creara nuevos objetos de fecha. Desafortunadamente *datetime.date* está escrito en C, por lo que no podía simplemente parchear mono el método estático `date.today()` .

Encontré una forma sencilla de hacer esto que implicaba ajustar eficazmente la clase date con un mock, pero pasar llamadas al constructor a la clase real (y devolver instancias reales).

El *patch decorator* se utiliza aquí como un mock de la clase `date` en el módulo en pruebas. A continuación, el atributo `side_effect` de la clase de fecha simulada se establece en una función lambda que devuelve una fecha real. Cuando la clase de fecha simulada se denomina fecha real, se construirá y devolverá `side_effect`.

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     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)
```

Tenga en cuenta que no parcheamos *datetime.date* globalmente, parcheamos `date` en el módulo que *uses*. Consulte *where to patch*.

Cuando `date.today()` es llamada se retorna una fecha conocida, pero llama al constructor `date(...)` este constructor todavía devuelve fechas normales. Sin esto, puede encontrarse teniendo que calcular un resultado esperado utilizando exactamente el mismo algoritmo que el código en prueba, que es un anti-patrón de prueba clásico.

Las llamadas al constructor de fecha se registran en los atributos `mock_date` (`call_count` y amigos) que también pueden ser útiles para las pruebas.

Una forma alternativa de tratar con fechas de mock, u otras clases integradas, se discute en esta entrada de blog.

Mocking de un Método Generador

Un generador de Python es una función o método que utiliza la instrucción `yield` para devolver una serie de valores cuando se itera sobre[1].

Se llama a un método / función del generador para devolver el objeto generador. Es el objeto generador que luego se itera. El método de protocolo para la iteración es *`__iter__()`*, por lo que podemos hacer mock de esto usando una *`MagicMock`*.

Aquí hay una clase de ejemplo con un método «iter» implementado como generador:

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

¿Cómo haríamos un mock de esta clase y, en particular, de su método «iter»?

Para configurar los valores devueltos desde la iteración (implícita en la llamada a *`list`*), necesitamos configurar el objeto devuelto por la llamada a `foo.iter()`.

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

Aplicar el mismo parche a cada método de prueba

Si desea que se coloquen varias revisiones para varios métodos de prueba, la forma obvia es aplicar los decoradores de parches a cada método. Esto puede parecer una repetición innecesaria. Para Python 2.6 o más reciente puede utilizar *`patch()`* (en todas sus diversas formas) como decorador de clases. Esto aplica las revisiones a todos los métodos de prueba de la clase. Un método de prueba se identifica mediante métodos cuyos nombres comienzan con `test`:

```
>>> @patch('mymodule.SomeClass')
... class MyTest(unittest.TestCase):
...
...     def test_one(self, MockSomeClass):
...         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'
```

Una forma alternativa de administrar parches es usar *métodos start y stop de patch*. Estos le permiten mover el parche en sus métodos `setUp` y `tearDown`.

[1] También hay expresiones generadoras y más usos avanzados <http://www.dabeaz.com/coroutines/index.html>"_ de generadores, pero no nos preocupan los que están aquí. Una muy buena introducción a los generadores y lo potentes que son es: Generator Tricks for Systems Programmers.

```
>>> 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()
```

Si utiliza esta técnica, debe asegurarse de que la aplicación de parches se «undone» llamando a `stop`. Esto puede ser más complicado de lo que podría pensar, porque si se produce una excepción en el setUp, no se llama a tearDown. *unittest.TestCase.addCleanup()* hace que esto sea más fácil:

```
>>> 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 de métodos sin enlazar

Mientras escribía pruebas hoy en día, necesitaba parchear un método *unbound method* (parchear el método en la clase en lugar de en la instancia). Necesitaba que se pasara a sí mismo como primer argumento porque quiero hacer aserciones sobre qué objetos estaban llamando a este método en particular. El problema es que no se puede parchear con un mock para esto, porque si reemplaza un método independiente con un mock, no se convierte en un método enlazado cuando se obtiene de la instancia, por lo que no se pasa por sí mismo. La solución es revisar el método independiente con una función real en su lugar. El decorador *patch()* hace que sea tan simple parchear métodos con un mock que tener que crear una función real se convierte en una molestia.

Si pasa `autospec=True` al parche, entonces hace el parche con un objeto de función *real*. Este objeto de función tiene la misma firma que el que está reemplazando, pero delega en un mock bajo el capó. Usted todavía consigue su mock de auto-creado exactamente de la misma manera que antes. Lo que significa, sin embargo, es que si lo usa para parchear un método independiente en una clase, la función simulada se convertirá en un método enlazado si se obtiene de una instancia. Tendrá `self` pasado como el primer argumento, que es exactamente lo que quería:

```
>>> 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)
```

Si no usamos `autospec=True` entonces el método independiente se parchea con una instancia de Mock en su lugar, y no se llama con `self`.

Comprobación de varias llamadas con mock

mock tiene una buena API para hacer aserciones sobre cómo se usan sus objetos ficticios.

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

Si su mock solo se llama una vez, puede usar el método `assert_called_once_with()` que también afirma que `call_count` es uno.

```
>>> 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.
```

Tanto `assert_called_with` como `assert_called_once_with` hacen afirmaciones sobre la llamada *más reciente*. Si su mock va a ser llamado varias veces, y desea hacer aserciones sobre *todas* esas llamadas que puede utilizar *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()]
```

El helper *call* facilita la toma de aserciones sobre estas llamadas. Puede crear una lista de llamadas esperadas y compararla con `call_args_list`. Esto se ve notablemente similar al repr de la `call_args_list`:

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

Copiando con argumentos mutables

Otra situación es rara, pero puede morderte, es cuando se llama a tu mock con argumentos mutables. `call_args` y `call_args_list` almacenan *referencias* a los argumentos. Si el código sometido a prueba muta los argumentos ya no puede realizar aserciones sobre cuáles eran los valores cuando se llamó al mock.

Este es un código de ejemplo que muestra el problema. Imagine las siguientes funciones definidas en “mymodule”:

```
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

Cuando tratamos de probar que `grob` llama `frob` con el argumento correcto mira lo que sucede:

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
```

(continué en la próxima página)

(proviene de la página anterior)

```
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
    ...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

Una posibilidad sería que el mock copiara los argumentos que pasa. Esto podría causar problemas si realiza aserciones que se basan en la identidad del objeto para la igualdad.

Aquí hay una solución que utiliza la funcionalidad `side_effect`. Si proporciona una función `side_effect` para un mock, se llamará a `side_effect` con los mismos argumentos que el mock. Esto nos da la oportunidad de copiar los argumentos y almacenarlos para aserciones posteriores. En este ejemplo estoy usando *otro* mock para almacenar los argumentos de modo que pueda usar los métodos ficticios para hacer la aserción. Una vez más, una función auxiliar configura esto para mí.

```
>>> 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` se llama con el mock que se llamará. Devuelve un nuevo mock en el que hacemos la aserción. La función `side_effect` hace una copia de los args y llama a nuestro `new_mock` con la copia.

Nota: Si su simulacro solo se va a usar una vez, hay una forma más fácil de verificar los argumentos en el punto en que se llaman. Simplemente puede hacer la comprobación dentro de una función `side_effect`.

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
    ...
AssertionError
```

Un enfoque alternativo es crear una subclase de *Mock* o *MagicMock* que copie (usando *copy.deepcopy()*) los argumentos. A continuación se muestra un ejemplo de implementación:

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, /, *args, **kwargs):
```

(continué en la próxima página)

(proviene de la página anterior)

```
...         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='...'>
```

Cuando subclases `Mock` o `MagicMock` todos los atributos creados dinámicamente, y el `return_value` usará tu subclase automáticamente. Eso significa que todos los elementos secundarios de un `CopyingMock` también tendrán el tipo `CopyingMock`.

Anidando Parches

Usar parches como administradores de contexto es bueno, pero si haces varios parches, puedes terminar con instrucciones anidadas que se sangran cada vez más a la derecha:

```
>>> 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
```

Con las funciones unittest `cleanup` y *métodos start y stop de patch* podemos lograr el mismo efecto sin la sangría anidada. Un método auxiliar simple, `create_patch`, pone el parche en su lugar y devuelve el mock creado para nosotros:

```
>>> 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

Mocking a un diccionario usando MagickMock

Es posible que desee simular un diccionario u otro objeto contenedor, registrando todo el acceso a él mientras todavía se comporta como un diccionario.

Podemos hacer esto con *MagicMock*, que se comportará como un diccionario, y usando *side_effect* para delegar el acceso del diccionario a un diccionario subyacente real que está bajo nuestro control.

Cuando se llama a los métodos `__getitem__()` y `__setitem__()` de nuestro `MagicMock` (acceso normal al diccionario), entonces se llama a `side_effect` con la clave (y en el caso de `__setitem__` el valor también). También podemos controlar lo que se devuelve.

Después de que se haya utilizado el `MagicMock` podemos usar atributos como *call_args_list* para afirmar cómo se usó el diccionario:

```
>>> 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: Una alternativa al uso de `MagicMock` es usar `Mock` y *solo* proporcionar los métodos mágicos que desea específicamente:

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

Una *tercera* opción es usar `MagicMock` pero pasando `dict` como el argumento *spec* (o *spec_set*) para que el `MagicMock` creado solo tenga métodos mágicos de diccionario disponibles:

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

Con estas funciones de efectos secundarios en su lugar, el `mock` se comportará como un diccionario normal pero registrando el acceso. Incluso genera un *KeyError* si intenta acceder a una clave que no existe.

```
>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
Traceback (most recent call last):
    ...
KeyError: 'd'
>>> mock['b'] = 'fish'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

Una vez utilizado, puede realizar aserciones sobre el acceso utilizando los métodos y atributos mock normales:

```
>>> 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 de subclases y sus atributos

Hay varias razones por las que es posible que desee crear subclases *Mock* Una razón podría ser agregar métodos auxiliares. Aquí hay un ejemplo simple:

```
>>> 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
```

El comportamiento estándar para las instancias `Mock` es que los atributos y los simulacros de valor devuelto son del mismo tipo que el simulacro en el que se accede a ellos. Esto garantiza que los atributos `Mock` son `Mocks` y los atributos `MagicMock` son `MagicMocks`[2]. Por lo tanto, si está creando subclases para agregar métodos auxiliares, también estarán disponibles en los atributos y el valor devuelto mock de las instancias de su subclase.

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has_been_called()
True
```

A veces esto es inconveniente. Por ejemplo, un usuario está creando subclases de mock para crear un Twisted adaptor. Tener esto aplicado a los atributos también puede causar errores en realidad.

`Mock` (en todos sus sabores) utiliza un método llamado `_get_child_mock` para crear estos «sub-mocks» para atributos y valores devueltos. Puede evitar que la subclase se utilice para los atributos invalidando este método. La firma es que toma argumentos de palabra clave arbitrarios (`**kwargs`) que luego se pasan al constructor ficticio:

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, /, **kwargs):
...         return MagicMock(**kwargs)
```

(continué en la próxima página)

[2] Una excepción a esta regla son los mock no invocables. Los atributos usan la variante a la que se puede llamar porque, de lo contrario, los simulacros a los que no se puede llamar no podrían tener métodos a los que se puede llamar.

(proviene de la página anterior)

```
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

Importaciones de Mocking con patch.dict

Una situación en la que un mocking puede ser difícil es cuando tiene una importación local dentro de una función. Estos son más difíciles de simular porque no están usando un objeto del espacio de nombres del módulo que podemos revisar.

Por lo general, deben evitarse las importaciones locales. A veces se hacen para evitar dependencias circulares, para las que hay *generalmente* una manera mucho mejor de resolver el problema (refactorizar el código) o para evitar «costos iniciales» retrasando la importación. Esto también se puede resolver de mejores maneras que una importación local incondicional (almacenar el módulo como una clase o atributo de módulo y solo hacer la importación en el primer uso).

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.

Esto significa que puede usar *patch.dict()* para *temporalmente* poner un mock en su lugar sobre *sys.modules*. Cualquier importación mientras este parche está activo recuperará el mock. Cuando el parche está completo (la función decorada sale, el cuerpo de la instrucción with está completo o se llama a `patcher.stop()`) entonces lo que había anteriormente se restaurará de forma segura.

Aquí hay un ejemplo de mock del módulo “fooble”.

```
>>> 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()
```

Como puede ver, el `import fooble` tiene éxito, pero al salir no queda ningún “fooble” en *sys.modules*.

Esto también funciona para el formulario `from module import name`:

```
>>> 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()
```

Con un poco más de trabajo también puede simular las importaciones de paquetes:

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     fooble()
...
<Mock name='mock.module.fooble()' id='...'>
>>> mock.module.fooble.assert_called_once_with()
```

Seguimiento del orden de las llamadas y de las aserciones de llamadas menos detalladas

La clase *Mock* le permite realizar un seguimiento del *orden* de las llamadas a métodos en sus objetos mock a través del atributo *method_calls*. Esto no le permite rastrear el orden de las llamadas entre objetos simulados separados, sin embargo, podemos usar *mock_calls* para lograr el mismo efecto.

Debido a que los mocks rastrean las llamadas a mocks secundarios en `mock_calls`, y el acceso a un atributo arbitrario de un mock crea un mock secundario, podemos crear nuestros mocks separados de uno primario. Las llamadas a esos mocks hijos se grabarán, en orden, en el `mock_calls` del padre:

```
>>> 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()]
```

A continuación, podemos realizar afirmaciones sobre las llamadas, incluido el orden, comparando con el atributo `mock_calls` en el mock administrador:

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

Si `patch` está creando y poniendo en su lugar, sus mocks, puede adjuntarlos a un mock administrador usando el método *attach_mock()* . Después de adjuntar las llamadas se registrarán en `mock_calls` del administrador.

```
>>> 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='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
call.MockClass1().foo(),
call.MockClass2(),
call.MockClass2().bar()]
```

Si se han realizado muchas llamadas, pero solo está interesado en una secuencia particular de ellas, entonces una alternativa es usar el método *assert_has_calls()* . Esto toma una lista de llamadas (construidas con el objeto *call*). Si esa secuencia de llamadas sobre *mock_calls*, la aserción se realiza correctamente.

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
```

(continué en la próxima página)

(proviene de la página anterior)

```
<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)
```

Aunque la llamada encadenada `m.one().two().three()` no son las únicas llamadas que se han realizado al mock, la aserción aún tiene éxito.

A veces, un mock puede tener varias llamadas, y solo le interesa afirmar sobre *algunas* de esas llamadas. Puede que ni siquiera importe el pedido. En este caso, puede pasar `any_order=True` a `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)
```

Coincidencia de argumentos más compleja

Usando el mismo concepto básico que *ANY* podemos implementar comparadores para hacer afirmaciones más complejas en objetos usados como argumentos para mocks.

Supongamos que esperamos que algún objeto se pase a un mock que, por defecto, se compara igual en función de la identidad del objeto (que es el valor predeterminado de Python para las clases definidas por el usuario). Para usar *assert_called_with()* tendríamos que pasar exactamente el mismo objeto. Si solo estamos interesados en algunos de los atributos de este objeto, podemos crear un comparador que verifique estos atributos por nosotros.

Puede ver en este ejemplo cómo una llamada "estándar" a `assert_called_with` no es suficiente:

```
>>> 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...>)
```

Una función de comparación para nuestra clase `Foo` podría verse así:

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

Y un objeto comparador que puede usar funciones de comparación como esta para su operación de igualdad se vería así:

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
```

(continué en la próxima página)

(proviene de la página anterior)

```
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

Poniendo todo esto junto:

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

El `Matcher` es instanciado con nuestra función de comparación y el objeto `Foo` con el que queremos comparar. En `assert_called_with` se llamará al método de igualdad `Matcher`, que compara el objeto con el que se llamó al mock con el que creamos nuestro matcher. Si coinciden, entonces pasa `assert_called_with`, y si no lo hacen, se lanzará un *AssertionError*:

```
>>> 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...>,), {})
```

Con un poco de ajuste, podría hacer que la función de comparación genere el *AssertionError* directamente y proporcione un mensaje de falla más útil.

A partir de la versión 1.5, la biblioteca de pruebas de Python PyHamcrest proporciona una funcionalidad similar, que puede ser útil aquí, en la forma de su comparador de igualdad (hamcrest.library.integration.match_equality).

26.11 2to3 — Automated Python 2 to 3 code translation

2to3 es un programa de Python que lee el código fuente de Python 2.x y aplica una serie de *fixers* para transformarlo en un código válido de Python 3.x. La biblioteca estándar contiene un amplio conjunto de arreglos que manejarán casi todo el código. Biblioteca de soporte 2to3 *lib2to3* es, sin embargo, una biblioteca flexible y genérica, por lo que es posible escribir sus propios arreglos para 2to3.

26.11.1 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, `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 identación 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 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. Acá, 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.

Dado que algunas declaraciones de impresión se pueden analizar como llamadas a funciones o declaraciones, 2to3 no siempre puede leer archivos que contienen la función de impresión. Cuando 2to3 detecta la presencia de la directiva del compilador `from __future__ import print_function`, modifica su gramática interna para interpretar *`print()`* como una función. Este cambio también se puede habilitar manualmente con la opción `-p`. Utilice `-p` para ejecutar correctores en el código que ya tiene sus declaraciones de impresión convertidas. También `-e` puede usarse para hacer *`exec()`* una función.

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 sobreescriben 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 una 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
```

26.11.2 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
`failUnlessEqual(a, b)`	*assertEqual(a, b)*
`assertEquals(a, b)`	*assertEqual(a, b)*
`failIfEqual(a, b)`	*assertNotEqual(a, b)*
`assertNotEquals(a, b)`	*assertNotEqual(a, b)*
`failUnless(a)`	*assertTrue(a)*
`assert_(a)`	*assertTrue(a)*
`failIf(a)`	*assertFalse(a)*
`failUnlessRaises(exc, cal)`	*assertRaises(exc, cal)*
`failUnlessAlmostEqual(a, b)`	*assertAlmostEqual(a, b)*
`assertAlmostEquals(a, b)`	*assertAlmostEqual(a, b)*
`failIfAlmostEqual(a, b)`	*assertNotAlmostEqual(a, b)*
`assertNotAlmostEquals(a, b)`	*assertNotAlmostEqual(a, b)*

basestring
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 metaclase (`__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` is 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
Renames definitions of methods called `__nonzero__()` to `__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
`operator.isCallable(obj)`	`callable(obj)`
`operator.sequenceIncludes(obj)`	`operator.contains(obj)`
`operator.isSequenceType(obj)`	`isinstance(obj, collections.abc.Sequence)`
`operator.isMappingType(obj)`	`isinstance(obj, collections.abc.Mapping)`
`operator.isNumberType(obj)`	`isinstance(obj, numbers.Number)`
`operator.repeat(obj, n)`	`operator.mul(obj, n)`
`operator.irepeat(obj, n)`	`operator.imul(obj, n)`

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.

26.11.3 `lib2to3` — 2to3's library

Código fuente: Lib/lib2to3/

Obsoleto desde la versión 3.11, se eliminará en la versión 3.13: Python 3.9 cambió a un analizador PEG (consulte **PEP 617**) mientras que lib2to3 usa un analizador LL (1) menos flexible. Python 3.10 incluye una nueva sintaxis de lenguaje que el analizador LL (1) de lib2to3 no puede analizar (consulte **PEP 634**). El módulo `lib2to3` se marcó como pendiente de ser obsoleto en Python 3.9 (aumentando *PendingDeprecationWarning* en la importación) y completamente obsoleto en Python 3.11 (aumentando *DeprecationWarning*). Se eliminará de la biblioteca estándar en Python 3.13. Considere alternativas de terceros como LibCST o parso.

Nota: La API del módulo *lib2to3* debe considerarse inestable y puede cambiar drásticamente en el futuro.

26.12 `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.

26.12.1 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 ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...
```

(continué en la próxima página)

(proviene de la página anterior)

```
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'

  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.

26.12.2 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.

26.13 `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. Está en desuso a favor de pruebas basadas en *unittest*y los métodos de aserción de *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.**LOOPBACK_TIMEOUT**
Tiempo de espera en segundos para las pruebas que utilizan un servidor de red que escucha en la interfaz local de loopback de la red como `127.0.0.1`.

El tiempo de espera es lo suficientemente largo para evitar el fracaso de la prueba: tiene en cuenta que el cliente y el servidor pueden ejecutarse en diferentes hilos o incluso en diferentes procesos.

El tiempo de espera debe ser lo suficientemente largo para los métodos *connect()*, *recv()* y *send()* de *socket.socket*.

Su valor por defecto es de 5 segundos.

Ver también *INTERNET_TIMEOUT*.

test.support.**INTERNET_TIMEOUT**
Tiempo de espera en segundos para las solicitudes de red que van a Internet.

El tiempo de espera es lo suficientemente corto como para evitar que una prueba espere demasiado tiempo si la solicitud de Internet se bloquea por cualquier motivo.

Normalmente, un tiempo de espera utilizando *INTERNET_TIMEOUT* no debería marcar una prueba como fallida, sino que debería omitir la prueba: ver *transient_internet()*.

Su valor por defecto es de 1 minuto.

Ver también *LOOPBACK_TIMEOUT*.

test.support.**SHORT_TIMEOUT**
Tiempo de espera en segundos para marcar una prueba como fallida si la prueba tarda «demasiado».

El valor del tiempo de espera depende de la opción de línea de comandos regrtest `--timeout`.

Si una prueba que utiliza *SHORT_TIMEOUT* empieza a fallar aleatoriamente en buildbots lentos, utilice en su lugar *LONG_TIMEOUT*.

Su valor por defecto es de 30 segundos.

test.support.**LONG_TIMEOUT**
Tiempo de espera en segundos para detectar cuando se cuelga una prueba.

Es lo suficientemente largo como para reducir el riesgo de fracaso de la prueba en los buildbots de Python más lentos. No debe utilizarse para marcar una prueba como fallida si la prueba tarda «demasiado». El valor del tiempo de espera depende de la opción de línea de comandos regrtest `--timeout`.

Su valor por defecto es de 5 minutos.

Ver también *LOOPBACK_TIMEOUT*, *INTERNET_TIMEOUT* y *SHORT_TIMEOUT*.

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**
Set to `True` if Python is built without docstrings (the `WITH_DOC_STRINGS` macro is not defined). See the `configure --without-doc-strings` option.

See also the *HAVE_DOCSTRINGS* variable.

test.support.**HAVE_DOCSTRINGS**
Set to `True` if function docstrings are available. See the `python -OO` option, which strips docstrings of functions implemented in Python.

See also the *MISSING_C_DOCSTRINGS* variable.

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.**NEVER_EQ**
Objeto que no es igual a nada (incluso a *ALWAYS_EQ*). 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.**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.**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.**match_test**(*test*)
Determine whether *test* matches the patterns set in *set_match_tests()*.

test.support.**set_match_tests**(*accept_patterns=None*, *ignore_patterns=None*)
Define match patterns on test filenames and test method names for filtering tests.

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*)
Use this check to guard CPython's implementation-specific tests or to run them only on the implementations guarded by the arguments. This function returns `True` or `False` depending on the host platform. Example usage:

```
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.**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 regrtest.

test.support.**get_original_stdout**()
Retorna el *stdout* original establecido por *record_original_stdout()* o `sys.stdout` si no está configurado.

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.**disable_faulthandler**()
A context manager that temporary disables *faulthandler*.

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**()
A context manager that disables the garbage collector on entry. On exit, the garbage collector is restored to its prior state.

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.**print_warning**(*msg*)
Imprime una advertencia en *sys.__stderr__*. Formatea el mensaje como: `f "Warning -- {msg}"`. Si *msg* se compone de varias líneas, añade el prefijo `"Warning --"` a cada línea.

Nuevo en la versión 3.9.

test.support.**wait_process**(*pid*, *, *exitcode*, *timeout=None*)
Espere hasta que el proceso *pid* termine y comprobará que el código de salida del proceso es *exitcode*.

Lanza un *AssertionError* si el código de salida del proceso no es igual a *exitcode*.

Si el proceso se ejecuta durante más tiempo que *timeout* en segundos (*SHORT_TIMEOUT* por defecto), mata el proceso y lanza un *AssertionError*. La función de tiempo de espera no está disponible en Windows.

Nuevo en la versión 3.9.

test.support.**calcobjsize**(*fmt*)
Return the size of the PyObject whose structure members are defined by *fmt*. The returned value includes the size of the Python object header and alignment.

test.support.**calcvobjsize**(*fmt*)
Return the size of the PyVarObject whose structure members are defined by *fmt*. The returned value includes the size of the Python object header and alignment.

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.**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.**system_must_validate_cert**(*f*)
A decorator that skips the decorated test on TLS certification validation failures.

@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*)
Decorator for the minimum version when running test on FreeBSD. If the FreeBSD version is less than the minimum, the test is skipped.

@test.support.**requires_linux_version**(**min_version*)
Decorator for the minimum version when running test on Linux. If the Linux version is less than the minimum, the test is skipped.

@test.support.**requires_mac_version**(**min_version*)
Decorator for the minimum version when running test on macOS. If the macOS version is less than the minimum, the test is skipped.

@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**
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**
Decorador para desactivar temporalmente el seguimiento durante la duración de la prueba.

@test.support.**refcount_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.**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**
Decorator for tests that fill the address space.

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 lanza 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.**open_urlresource**(*url*, **args*, ***kw*)
Abre *url*. Si la apertura falla, se lanza *TestFailed*.

test.support.**reap_children**()
Se utiliza esto al final de `test_main` siempre que se inicien subprocesos. 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.**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.**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

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

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=()*)
Assert instances of *cls* are deallocated after iterating.

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=()*, *not_exported=()*)
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 *not_exported* 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        not_exported = {'baz'}  # Undocumented name.
        # bar imports part of its API from _bar.
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra, not_exported=not_exported)
```

Nuevo en la versión 3.6.

test.support.**skip_if_broken_multiprocessing_synchronize**()
Omite las pruebas si el módulo no se encuentra al `multiprocessing.synchronize`, si no hay una implementación de semáforo disponible, o si crear un *lock* lanza un *OSError*.

Nuevo en la versión 3.10.

test.support.**check_disallow_instantiation**(*test_case*, *tp*, **args*, ***kwds*)
Se aserciona que el tipo *tp* no pueda ser instanciado usando *args* y *kwds*.

Nuevo en la versión 3.10.

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 3.10.7.

El módulo *test.support* define las siguientes clases:

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.**SaveSignals**
Clase para guardar y restaurar manejadores (*handlers*) de señal registrados por el manejador de señal Python.

save(*self*)
Save the signal handlers to a dictionary mapping signal numbers to the current signal handler.

restore(*self*)
Set the signal numbers from the *save()* dictionary to the saved handler.

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.**BasicTestRunner**

run(*test*)
Retorna *test* y retorna el resultado..

26.14 test.support.socket_helper — Utilidades para pruebas de socket

El módulo *test.support.socket_helper* proporciona soporte para las pruebas de socket.

Nuevo en la versión 3.9.

test.support.socket_helper.**IPV6_ENABLED**
Se establece como `True` si IPv6 está habilitado en este host, `False` en caso contrario.

test.support.socket_helper.**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.socket_helper.**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.socket_helper.**bind_unix_socket**(*sock*, *addr*)
Bind a Unix socket, raising *unittest.SkipTest* if *PermissionError* is raised.

@test.support.socket_helper.**skip_unless_bind_unix_socket**
Un decorador para ejecutar pruebas que requieren un enlace(`bind()`) funcional para sockets en sistemas Unix.

test.support.socket_helper.**transient_internet**(*resource_name*, ***, *timeout=30.0*, *errnos=()*)
Un gestor de contexto que lanza *ResourceDenied* cuando varios problemas con la conexión a Internet se manifiestan como excepciones.

26.15 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 `PYTHONUSERSITE` 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 subproceso. Los valores pueden incluir `__isolated`, `__cleanenv`, `__cwd` y `TERM`.

Distinto en la versión 3.9: La función ya no elimina los espacios en blanco de *stderr*.

`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)`.

If the *__cleanenv* keyword-only parameter is set, *env_vars* is used as a fresh environment.

Python is started in isolated mode (command line option `-I`), except if the *__isolated* keyword-only parameter is set to `False`.

Distinto en la versión 3.9: La función ya no elimina los espacios en blanco de *stderr*.

`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.

Distinto en la versión 3.9: La función ya no elimina los espacios en blanco de *stderr*.

`test.support.script_helper.`**`spawn_python`**(**args*, *stdout=subprocess.PIPE*, *stderr=subprocess.STDOUT*, ***kw*)
Ejecuta un subproceso 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.

26.16 test.support.bytecode_helper — Herramientas de apoyo para comprobar la correcta generación de bytecode

El módulo *test.support.bytecode_helper* proporciona soporte para probar e inspeccionar la generación de código de bytes.

Nuevo en la versión 3.9.

El módulo define la siguiente clase:

class test.support.bytecode_helper.**BytecodeTestCase**(*unittest.TestCase*)
Esta clase tiene métodos de aserción personalizados para inspeccionar el código de bytes.

BytecodeTestCase.**get_disassembly_as_string**(*co*)
Retorna el desensamblaje de *co* como cadena.

BytecodeTestCase.**assertInBytecode**(*x*, *opname*, *argval=_UNSPECIFIED*)
Retorna instr si se encuentra *opname*, de lo contrario lanza *AssertionError*.

BytecodeTestCase.**assertNotInBytecode**(*x*, *opname*, *argval=_UNSPECIFIED*)
Lanza *AssertionError* si se encuentra *opname*.

26.17 test.support.threading_helper — Utilidades para pruebas con hilos

El módulo *test.support.threading_helper* proporciona soporte para las pruebas de socket.

Nuevo en la versión 3.10.

test.support.threading_helper.**join_thread**(*thread*, *timeout=None*)
Se une un *thread* dentro de *timeout*. Lanza un *AssertionError* si el hilo sigue vivo después de unos segundos de *timeout*.

@test.support.threading_helper.**reap_threads**
Decorador para garantizar que los hilos se limpien incluso si la prueba falla.

test.support.threading_helper.**start_threads**(*threads*, *unlock=None*)
Context manager to start *threads*, which is a sequence of threads. *unlock* is a function called after the threads are started, even if an exception was raised; an example would be *threading.Event.set()*. start_threads will attempt to join the started threads upon exit.

test.support.threading_helper.**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.threading_helper.**threading_setup**()
Retorna el recuento del hilo actual y copia de subprocesos colgantes.

test.support.threading_helper.**wait_threads_exit**(*timeout=None*)
El administrador de contexto debe esperar hasta que salgan todos los hilos creados en la declaración with.

test.support.threading_helper.**catch_threading_exception**()
El administrador de contexto captura *threading.Thread* excepción usando *threading.excepthook()*.

Atributos establecidos cuando se captura 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 threading_helper.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.

26.18 test.support.os_helper — Utilidades para pruebas de sistemas operativos

El módulo *test.support.os_helper* proporciona soporte para las pruebas de sistemas operativos.

Nuevo en la versión 3.10.

test.support.os_helper.**FS_NONASCII**
Un carácter no codificable ASCII codificable por *os.fsencode()*.

test.support.os_helper.**SAVEDCWD**
Establecido *os.getcwd()*.

test.support.os_helper.**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.os_helper.**TESTFN_NONASCII**
Set to a filename containing the *FS_NONASCII* character, if it exists. This guarantees that if the filename exists, it can be encoded and decoded with the default filesystem encoding. This allows tests that require a non-ASCII filename to be easily skipped on platforms where they can't work.

test.support.os_helper.**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.os_helper.**TESTFN_UNDECODABLE**
Establecido un nombre de archivo (tipo de bytes) que no pueda descodificarse mediante la codificación del sistema de archivos en modo estricto. Puede ser `None` si no es posible generar dicho nombre de archivo.

test.support.os_helper.**TESTFN_UNICODE**
Establecido un nombre que no sea ASCII para un archivo temporal.

class test.support.os_helper.**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 el `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.

class test.support.os_helper.**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__()`.

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`.

test.support.os_helper.**can_symlink**()
Retorna `True` si el sistema operativo admite links simbólicos, de lo contrario `False`.

test.support.os_helper.**can_xattr**()
Retorna `True` si el sistema operativo admite *xattr*, de lo contrario `False`.

test.support.os_helper.**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.os_helper.**create_empty_file**(*filename*)
Crea un archivo vacío con *filename*. Si ya existe, truncarlo.

test.support.os_helper.**fd_count**()
Cuenta el número de descriptores de archivo abiertos.

test.support.os_helper.**fs_is_case_insensitive**(*directory*)
Retorna `True` si el sistema de archivos para *directory* no distingue entre mayúsculas y minúsculas.

test.support.os_helper.**make_bad_fd**()
Se crea un descriptor de archivo no válido abriendo y cerrando un archivo temporal y retornando su descriptor.

test.support.os_helper.**rmdir**(*filename*)
Call *os.rmdir()* on *filename*. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the file, which is needed due to antivirus programs that can hold files open and prevent deletion.

test.support.os_helper.**rmtree**(*path*)
Call *shutil.rmtree()* on *path* or call *os.lstat()* and *os.rmdir()* to remove a path and its contents. As with *rmdir()*, on Windows platforms this is wrapped with a wait loop that checks for the existence of the files.

@test.support.os_helper.**skip_unless_symlink**
Un decorador para ejecutar pruebas que requieren soporte para enlaces simbólicos.

@test.support.os_helper.**skip_unless_xattr**
Un decorador para ejecutar pruebas que requieren soporte para *xattr*.

test.support.os_helper.**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.os_helper.**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.os_helper.**temp_umask**(*umask*)
Un administrador de contexto que establece temporalmente el proceso *umask*.

test.support.os_helper.**unlink**(*filename*)
Call *os.unlink()* on *filename*. As with *rmdir()*, on Windows platforms, this is wrapped with a wait loop that checks for the existence of the file.

26.19 test.support.import_helper — Utilidades para pruebas de importación

El módulo *test.support.import_helper* proporciona soporte para las pruebas de importación.

Nuevo en la versión 3.10.

test.support.import_helper.**forget**(*module_name*)
Elimina el módulo llamado *module_name* de sys.modules y elimina los archivos compilados por bytes del módulo.

test.support.import_helper.**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.import_helper.**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_helper.**modules_setup**()
Retorna una copia de *sys.modules*.

test.support.import_helper.**modules_cleanup**(*oldmodules*)
Elimina los módulos a excepción de *oldmodules* y encodings para preservar la memoria caché interna.

test.support.import_helper.**unload**(*name*)
Elimina *name* de sys.modules.

test.support.import_helper.**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*.

class test.support.import_helper.**CleanImport**(**module_names*)
A context manager to force import to return a new module reference. This is useful for testing module-level behaviors, such as the emission of a *DeprecationWarning* on import. Example usage:

```
with CleanImport('foo'):
    importlib.import_module('foo')  # New reference.
```

class test.support.import_helper.**DirsOnSysPath**(**paths*)
A context manager to temporarily add directories to *sys.path*.

Esto hace una copia de *sys.path*, agrega cualquier directorio dado como argumento posicional, luego revierte *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.

26.20 test.support.warnings_helper — Utilidades para pruebas de advertencias

El módulo *test.support.warnings_helper* proporciona soporte para las pruebas de advertencias.

Nuevo en la versión 3.10.

test.support.warnings_helper.**ignore_warnings**(**, category*)
Suppress warnings that are instances of *category*, which must be *Warning* or a subclass. Roughly equivalent to *warnings.catch_warnings()* with *warnings.simplefilter('ignore', category=category)*. For example:

```
@warning_helper.ignore_warnings(category=DeprecationWarning)
def test_suppress_warning():
    # do something
```

Nuevo en la versión 3.8.

test.support.warnings_helper.**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.warnings_helper.**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.warnings_helper.**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 lanzaran 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*.

class test.support.warnings_helper.**WarningsRecorder**

La clase utilizada para registrar advertencias para pruebas unitarias. Consulte la documentación de *check_warnings()* arriba para obtener más detalles.

CAPÍTULO 27

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

Esta tabla contiene todos los eventos generados por las llamadas *sys.audit()* o `PySys_Audit()` en todo el tiempo de ejecución de CPython y la biblioteca estándar. Estas llamadas se agregaron en 3.8.0 o posterior (consulte **PEP 578**).

Ver *sys.addaudithook()* y `PySys_AddAuditHook()` para informarse en el manejo de estos eventos.

Detalles de implementación de CPython: 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
array.__new__	`typecode`, `initializer`
builtins.breakpoint	`breakpointhook`
builtins.id	`id`
builtins.input	`prompt`
builtins.input/result	`result`
code.__new__	`code`, `filename`, `name`, `argcount`, `posonlyargcount`, `kwonlyargcount`, `nloc`
compile	`source`, `filename`
cpython.PyInterpreterState_Clear	
cpython.PyInterpreterState_New	
cpython._PySys_ClearAuditHooks	
cpython.run_command	`command`
cpython.run_file	`filename`
cpython.run_interactivehook	`hook`
cpython.run_module	`module-name`
cpython.run_startup	`filename`

Tabla 1 – proviene de la página anterior

Audit event	Arguments
cpython.run_stdin	
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
glob.glob/2	pathname, recursive, root_dir, dir_fd
http.client.connect	self, host, port
http.client.send	self, data
imaplib.open	self, host, port
imaplib.send	self, data
import	module, filename, sys.path, sys.meta_path, sys.path_hooks
marshal.dumps	value, version
marshal.load	
marshal.loads	bytes
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	path, mode, flags
os.add_dll_directory	path
os.chdir	path
os.chflags	path, flags
os.chmod	path, mode, dir_fd

Tabla 1 – proviene de la página anterior

Audit event	Arguments
os.chown	path, uid, gid, dir_fd
os.exec	path, args, env
os.fork	
os.forkpty	
os.fwalk	top, topdown, onerror, follow_symlinks, dir_fd
os.getxattr	path, attribute
os.kill	pid, sig
os.killpg	pgid, sig
os.link	src, dst, src_dir_fd, dst_dir_fd
os.listdir	path
os.listxattr	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.startfile/2	path, operation, arguments, cwd, show_cmd
os.symlink	src, dst, dir_fd
os.system	command
os.truncate	fd, length
os.unsetenv	key
os.utime	path, times, ns, dir_fd
os.walk	top, topdown, onerror, followlinks
pathlib.Path.glob	self, pattern
pathlib.Path.rglob	self, pattern
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

Tabla 1 – proviene de la página anterior

Audit event	Arguments
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
sqlite3.connect	database
sqlite3.connect/handle	connection_handle
sqlite3.enable_load_extension	connection, enabled
sqlite3.load_extension	connection, path
subprocess.Popen	executable, args, cwd, env
sys._current_exceptions	
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

Tabla 1 – proviene de la página anterior

Audit event	Arguments
winreg.SaveKey	`key`, `file_name`
winreg.SetValue	`key`, `sub_key`, `type`, `value`

Los siguientes eventos se generan internamente y no corresponden a ninguna API pública de CPython:

Evento de auditoría	Argumentos
_winapi.CreateFile	`file_name`, `desired_access`, `share_mode`, `creation_disposition`, `flags_and_attributes`
_winapi.CreateJunction	`src_path`, `dst_path`
_winapi.CreateNamedPipe	`name`, `open_mode`, `pipe_mode`
_winapi.CreatePipe	
_winapi.CreateProcess	`application_name`, `command_line`, `current_directory`
_winapi.OpenProcess	`process_id`, `desired_access`
_winapi.TerminateProcess	`handle`, `exit_code`
ctypes.PyObj_FromPtr	`obj`

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=False*, *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 *bpbynumber* y por pares `(archivo, linea)` 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.

When creating a breakpoint, its associated *file name* should be in canonical form. If a *funcname* is defined, a breakpoint *hit* will be counted when the first line of that function is executed. A *conditional* breakpoint always counts a *hit*.

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:

- Breakpoint number.
- Temporary status (del or keep).
- File/line position.
- Break condition.
- Number of times to ignore.
- Number of times hit.

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.

Breakpoint instances have the following attributes:

file
File name of the *Breakpoint*.

line
Line number of the *Breakpoint* within *file*.

temporary
True if a *Breakpoint* at (file, line) is temporary.

cond
Condition for evaluating a *Breakpoint* at (file, line).

funcname
Function name that defines whether a *Breakpoint* is hit upon entering the function.

enabled
True if *Breakpoint* is enabled.

bpbynumber
Numeric index for a single instance of a *Breakpoint*.

bplist
Dictionary of *Breakpoint* instances indexed by (*file*, *line*) tuples.

ignore
Number of times to ignore a *Breakpoint*.

hits
Count of the number of times a *Breakpoint* has been hit.

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*)
Return canonical form of *filename*.

For real file names, the canonical form is an operating-system-dependent, *case-normalized absolute path*. A *filename* with angle brackets, such as "<stdin>" generated in interactive mode, is returned unchanged.

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.

is_skipped_line(*module_name*)
Return True if *module_name* matches any skip pattern.

stop_here(*frame*)
Return True if *frame* is below the starting frame in the stack.

break_here(*frame*)
Return True if there is an effective breakpoint for this line.

Check whether a line or function breakpoint exists and is in effect. Delete temporary breakpoints based on information from *effective()*.

break_anywhere(*frame*)
Return True if any breakpoint exists for *frame*'s filename.

Las clases derivadas deben redefinir estos métodos para adquirir el control sobre las operaciones de depuración.

user_call(*frame*, *argument_list*)
Called from *dispatch_call()* if a break might stop inside the called function.

user_line(*frame*)
Called from *dispatch_line()* when either *stop_here()* or *break_here()* returns `True`.

user_return(*frame*, *return_value*)
Called from *dispatch_return()* when *stop_here()* returns `True`.

user_exception(*frame*, *exc_info*)
Called from *dispatch_exception()* when *stop_here()* returns `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*, *lineno=None*)
Stop when the line with the *lineno* greater than the current one is reached or when returning from current frame.

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=False*, *cond=None*, *funcname=None*)
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*)
Delete the breakpoints in *filename* and *lineno*. If none were set, return an error message.

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 indice fuera de rango, se retorna un mensaje de error.

clear_all_file_breaks(*filename*)
Delete all breakpoints in *filename*. If none were set, return an error message.

clear_all_breaks()
Delete all existing breakpoints. If none were set, return an error message.

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*)
Return True if there is a breakpoint for *lineno* in *filename*.

get_breaks(*filename*, *lineno*)
Retorna todos los puntos de interrupción en la linea 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*)
Return a list of (frame, lineno) tuples in a stack trace, and a size.

The most recently called frame is last in the list. The size is the number of frames below the frame where the debugger was invoked.

format_stack_entry(*frame_lineno*, *lprefix=': '*)
Return a string with information about a stack entry, which is a `(frame, lineno)` tuple. The return string contains:

- The canonical filename which contains the frame.
- The function name or `"<lambda>"`.
- Los argumentos de entrada.
- El valor de retorno.
- La linea 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*, ***kwds*)
Depura una única llamada a función y retorna su resultado.

Por último, el módulo también define las siguientes funciones:

bdb.**checkfuncname**(*b*, *frame*)
Return True if we should break here, depending on the way the `Breakpoint` *b* was set.

If it was set via line number, it checks if `b.line` is the same as the one in *frame*. If the breakpoint was set via `function name`, we have to check we are in the right *frame* (the right function) and if we are on its first executable line.

bdb.**effective**(*file*, *line*, *frame*)
Return `(active breakpoint, delete temporary flag)` or `(None, None)` as the breakpoint to act upon.

The *active breakpoint* is the first entry in `bplist` for the (`file`, `line`) (which must exist) that is `enabled`, for which `checkfuncname()` is True, and that has neither a False `condition` nor positive `ignore` count. The *flag*, meaning that a temporary breakpoint should be deleted, is False only when the `cond` cannot be evaluated (in which case, `ignore` count is ignored).

If no such entry exists, then (None, None) is returned.

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.

El *Modo de Desarrollo Python* llama a `failurehandler.enable()` al inicio de Python.

Ver también:

Module ***pdb*** Interactive source code debugger for Python programs.

Module ***traceback*** Standard interface to extract, format and print stack traces of Python programs.

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.

Ver también:

`traceback.print_tb()`, which can be used to print a traceback object.

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 descriptores 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.

Distinto en la versión 3.10: Ahora el volcado menciona si se está ejecutando una colección de recolectores de basura si *all_threads* es verdadero.

`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 estado=1 después de volcar los rastreos. (Nota: `_exit()` termina el proceso inmediatamente, lo que significa que no hace ninguna limpieza como vaciar los búfers 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):
  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*.

Ver también:

Module *faulthandler* Used to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal.

Module *traceback* Standard interface to extract, format and print stack traces of Python programs.

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:

```
python -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` now accepts a `-m` option that execute modules similar to the way `python -m` does. As with a script, the debugger will pause execution just before the first line of the module.

The typical usage to break into the debugger is to insert:

```
import pdb; pdb.set_trace()
```

at the location you want to break into the debugger, and then run the program. You can then step through the code following this statement, and continue running without the debugger using the *continue* command.

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*)
Evaluate the *expression* (given as a string or a code object) under debugger control. When `runeval()` returns, it returns the value of the *expression*. Otherwise this function is similar to `run()`.

pdb.**runcall**(*function*, **args*, ***kwds*)
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 predeterminado).

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]

By default, Pdb sets a handler for the SIGINT signal (which is sent when the user presses `Ctrl-C` on the console) when you give a `continue` command. This allows you to break into the debugger again by pressing `Ctrl-C`. If you want Pdb not to touch the SIGINT handler, set *nosigint* to 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*)

[1] Si se considera que un marco se origina en un determinado módulo se determina por el `__name__` en el marcos globales.

runeval(*expression*, *globals=None*, *locals=None*)
runcall(*function*, **args*, ***kwds*)
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.

Multiple commands may be entered on a single line, separated by `;;`. (A single `;` is not used as it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands; the input is split at the first `;;` pair, even if it is in the middle of a quoted string. A workaround for strings with double semicolons is to use implicit string concatenation `';'';'` or `";"";"`.

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 ...]`
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 ...]`
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 ...]`
Habilitar los puntos de interrupción especificados.

ignore `bpnumber [count]`
Set the ignore count for the given breakpoint number. If *count* is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the *count* is decremented each time the breakpoint is reached and the breakpoint is not disabled and any associated condition evaluates to true.

condition `bpnumber [condition]`
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 *bpnumber*, `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.

If you use the `silent` command in the command list, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the other commands print anything, you see no sign that the breakpoint was reached.

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.

With *lineno*, continue execution until a line with a number greater or equal to *lineno* is reached. In both cases, also stop when the current frame returns.

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
Evaluate *expression* in the current context and print its value.

Nota: `print()` también se puede usar, pero no es un comando de depuración — esto ejecuta la función Python *print()*.

---

pp expression
Like the *p* command, except the value of *expression* is pretty-printed using the *pprint* module.

whatis expression
Print the type of *expression*.

source expression
Try to get source code of *expression* and display it.

Nuevo en la versión 3.2.

display [expression]
Display the value of *expression* if it changed, each time execution stops in the current frame.

Without *expression*, list all display expressions for the current frame.

Nuevo en la versión 3.2.

undisplay [expression]
Do not display *expression* anymore in the current frame. Without *expression*, clear all display expressions for the current frame.

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]]
Create an alias called *name* that executes *command*. The *command* must *not* be enclosed in quotes. Replaceable parameters can be indicated by `%1`, `%2`, and so on, while `%*` is replaced by all the parameters. If *command* is omitted, the current alias for *name* is shown. If no arguments are given, all aliases are listed.

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(f"%1.{k} = {%1.__dict__[k]}")
# Print instance variables in self
alias ps pi self
```

unalias name
Delete the specified alias *name*.

! 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 ...]
Restart the debugged Python program. If *args* is supplied, it is split with *shlex* and the result is used as the new *sys.argv*. History, breakpoints, actions and debugger options are preserved. *restart* is an alias for *run*.

q(uit)
Salga del depurador. El programa que se está ejecutando fue abortado.

debug code
Enter a recursive debugger that steps through *code* (which is an arbitrary expression or statement to be executed in the current environment).

retval
Imprime el valor de retorno para el último retorno de una función.

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  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 re.py:212(compile)
     1    0.000    0.000    0.001    0.001 re.py:268(_compile)
     1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
     1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

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.

tottime para el tiempo total empleado en la función dada (y excluyendo el tiempo realizado en llamadas a subfunciones)

percall es el cociente de `tottime` 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(function) 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()`* or `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()*.

get_stats_profile()
Este método retorna una instancia de StatsProfile, la cual contiene un mapeo de nombres de función a instancias de FunctionProfile. Cada instancia de FunctionProfile mantiene información relacionada al perfil de la función, como cuánto demoró la función en ejecutarse, cuantas veces fue llamada, etc…

Nuevo en la versión 3.9: Añadidas las siguientes clases de datos: StatsProfile, FunctionProfile. Añadida la siguiente función: get_stats_profile.

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 la cantidad de llamadas de Python dadas por el argumento, directamente y nuevamente bajo el generador de perfiles, midiendo el tiempo para ambos. Luego, calcula la sobrecarga oculta por evento del generador de perfiles y la retorna como un valor flotante. Por ejemplo, en un Intel Core i5 de 1.8Ghz que ejecuta macOS y usa el 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

# 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 forma sencilla de cronometrar pequeños fragmentos de código Python. Tiene tanto una *Interfaz de línea de comandos* como una *invocable*. Evita una serie de trampas comunes para medir los tiempos de ejecución. Véase también la introducción de Tim Peters al capítulo «Algoritmos» en la segunda edición de *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*)
Crea una instancia de `Timer` con la instrucción dada, el código *setup* y la función *timer* y ejecuta su método `repeat()` con las ejecuciones *repeat* y *number* dadas. 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.

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 >= 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
```

In the output, there are three fields. The loop count, which tells you how many times the statement body was run per timing loop repetition. The repetition count ("best of 5") which tells you how many times the timing loop was repeated, and finally the time the statement body took on average within the best repetition of the timing loop. That is, the time the fastest repetition took divided by the loop count.

```
>>> 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__
```

(continué en la próxima página)

(proviene de la página anterior)

```
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

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()`.

  runctx(*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*, ***kwds*)
  : 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`, `runctx` 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
# 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/regrtest.py", line 1276
    test_runner()
  File "/usr/lib/python3.4/test/regrtest.py", line 976
    display_failure=not verbose)
  File "/usr/lib/python3.4/test/regrtest.py", line 761
    match_tests=ns.match_tests)
  File "/usr/lib/python3.4/test/regrtest.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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
  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 linea `import pdb` el módulo *doctest*. El rastreo puede cambiar si se carga un nuevo módulo.

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 `<unkownn>`:

```
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()
```

(continué en la próxima página)

(proviene de la página anterior)

```
#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.

Graba los tamaños actual y máximo de todos los bloques de memoria rastreados

El siguiente código calcula dos sumas como `0 + 1 + 2 + ...` de forma ineficiente, creando una lista con estos números. Esta lista consume mucha memoria de forma temporal. Podemos usar *get_traced_memory()* y *reset_peak()* para observar el pequeño uso de memoria después de que la suma es calculada, así como también el uso máximo de memoria durante el cálculo:

```
import tracemalloc

tracemalloc.start()

# Example code: compute a sum with a large temporary list
large_sum = sum(list(range(100000)))

first_size, first_peak = tracemalloc.get_traced_memory()

tracemalloc.reset_peak()

# Example code: compute a sum with a small temporary list
small_sum = sum(list(range(1000)))

second_size, second_peak = tracemalloc.get_traced_memory()

print(f"{first_size=}, {first_peak=}")
print(f"{second_size=}, {second_peak=}")
```

Salida:

```
first_size=664, first_peak=3592984
second_size=804, second_peak=29704
```

El uso de *reset_peak()* asegura que podemos registrar precisamente el uso máximo de memoria durante el cálculo de `small_sum`, incluso si es mucho menor al máximo total desde la llamada a *start()*. Sin la llamada a *reset_peak()*, `second_peak` seguiría siendo el uso máximo de memoria del cálculo de `large_sum` (es decir, igual a `first_peak`). En este caso ambos máximos son mucho mayores al uso final de memoria, lo que sugiere que podemos optimizar (removiendo la llamada innecesaria a *list*, y escribiendo `sum(range(...))`).

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*)
Obtiene 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**()
Obtiene 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**()
Obtiene el tamaño actual y tamaño pico de los bloques de memorias rastreados por el módulo *tracemalloc* como una tupla: `(current: int, peak: int)`.

tracemalloc.**reset_peak**()
Establece el tamaño máximo de los bloques de memorias rastreados por el módulo *tracemalloc* al tamaño actual.

No realiza ninguna acción si el módulo *tracemalloc* no está rastreando asignaciones de memoria.

Esta función sólo modifica el valor guardado para el uso máximo de memoria, y no modifica o borra ningún rastreo, no como *clear_traces()*. Capturas tomadas con *take_snapshot()* antes de una llamada a *reset_peak()* pueden ser comparadas significativamente con capturas hechas después de la llamada.

Mira también la función *get_traced_memory()*.

Nuevo en la versión 3.9.

tracemalloc.**get_tracemalloc_memory**()
Obtiene 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*.

El número original de cuadros totales que componen el seguimiento observando el atributo *Traceback.total_nframe*.

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**()
Detiene el rastreo de las asignaciones de memoria de Python: desinstala los *hooks*. También limpia todos los rastros de memoria recolectados acerca de los bloques de memoria asignados por 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 linea (`int`) del filtro. Si *lineno* es `None`, el filtro se relaciona con cualquier número de linea.

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 linea 0.

Cuando se toma una captura, los seguimientos de los rastreos se limitan a *get_traceback_limit()* cuadros. Ver la función *take_snapshot()*. El número original de cuadros del seguimiento se guarda en el atributo *Traceback.total_nframe*. Esto permite saber si un seguimiento fue truncado por el límite de seguimientos.

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.

total_nframe
Número total de cuadros que componen el seguimiento antes de ser truncado. Este atributo puede ser `None` si no hay información disponible.

Distinto en la versión 3.9: El atributo *Traceback.total_nframe* fue añadido.

format(*limit=None*, *most_recent_first=False*)
Formatea el seguimiento como una lista de líneas. Usa el módulo *linecache* para recuperar líneas del código fuente. Si se configura el límite, formatea los cuadros más recientes de los límites, si límite es positivo. De otra manera, formatea los cuadros más antiguos de `abs(limit)`. Si *most_recent_first* es `True`, se invierte el orden de los cuadros formateados, devolviendo el cuadro más reciente en lugar 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())
```

CAPÍTULO 28

Empaquetado y distribución de software

Estas bibliotecas te ayudan a publicar e instalar software de Python. Aunque estos módulos están diseñados para funcionar junto con Python Package Index, también pueden ser utilizados con un servidor de indexado local, o sin servidor de indexado.

28.1 `distutils` — Creación e instalación de módulos Python

distutils está en desuso con la eliminación planificada para Python 3.12. Consulte Novedades para más información.

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 de versión cruzada 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.

Source code: Lib/ensurepip

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 sea al menos tan reciente como la disponible en 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 que especifica la versión disponible de pip que se instalará al arrancar 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 disponible.

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 subproceso 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/

The `venv` module supports creating lightweight «virtual environments», each with their own independent set of Python packages installed in their *site* directories. A virtual environment is created on top of an existing Python installation, known as the virtual environment's «base» Python, and may optionally be isolated from the packages in the base environment, so only those explicitly installed in the virtual environment are available.

When used from within a virtual environment, common installation tools such as pip will install Python packages into a virtual environment without needing to be told to do so explicitly.

See **PEP 405** for more background on Python virtual environments.

Ver también:

Guía del usuario de la Paquetería Python: Creación y uso de entornos virtuales

28.3.1 Creación de entornos virtuales

La creación de *entornos virtuales* se realiza al ejecutar el comando `venv`:

```
python -m venv /path/to/new/virtual/environment
```

Al ejecutar este comando crea el directorio de destino (la creación de los directorios principales que aún no existen) y coloca un archivo `pyvenv.cfg` en él con una clave `home` que apunta a la instalación de Python desde la cual se ejecutó el comando (un nombre común para el directorio de destino es `.venv`). También crea un subdirectorio `bin` (o `Scripts` en Windows) que contiene una copia / enlace simbólico de los binarios Python (según sea apropiado para la plataforma o los argumentos que se usaron en el momento de la creación del entorno). También crea un subdirectorio (inicialmente vacío) `lib/pythonX.Y/site-packages` (`Lib\site-packages` en Windows). Si se especifica un directorio existente, se reutilizará.

Obsoleto desde la versión 3.6: `pyvenv` was the recommended tool for creating virtual environments for Python 3.3 and 3.4, and is deprecated in Python 3.6.

Distinto en la versión 3.5: Ahora se recomienda el uso de `venv` para la creación de entornos virtuales.

En Windows, invoque el comando `venv` de la siguiente manera:

```
c:\>c:\Python35\python -m venv c:\path\to\myenv
```

Alternativamente, si configuró las variables `PATH` y `PATHEXT` para su instalación 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] [--upgrade-deps]
            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.
  --upgrade-deps        Upgrade core dependencies: pip setuptools to the
                        latest version in PyPI
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 3.9: Agrega la opción `--upgrade-deps` para actualizar pip + setuptools a la última versión en PyPI

Distinto en la versión 3.4: Instala pip por defecto, se agregaron las opciones `--without-pip` y `--copies`

Distinto en la versión 3.4: En versiones anteriores, si el directorio de destino ya existía, se lanzaba un error, a menos que se proporcionara la opción `--clear` o `--upgrade`.

Nota: Aunque los enlaces simbólicos son compatibles en Windows, no se recomiendan. Es de interés particular que al hacer doble clic en `python.exe` en el explorador de archivos, el enlace simbólico se resolverá e ignorará el entorno virtual.

Nota: En Microsoft Windows puede ser necesario habilitar el script `Activate.ps1` estableciendo la directiva de ejecución para el usuario. Puede hacer esto escribiendo el siguiente comando de PowerShell:

PS C:> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser

Ver Acerca de las directivas de ejecución para más información.

El archivo `pyvenv.cfg` creado también incluye la clave `include-system-site-packages`, establecida en `true` si se ejecuta `venv` con la opción `--system-site-packages`, `false` en caso contrario.

A menos que se proporcione la opción `--without-pip`, se invocará *ensurepip* para arrancar `pip` en el entorno virtual.

Se pueden asignar múltiples rutas a `venv`, en cuyo caso se creará un entorno virtual idéntico, de acuerdo a las opciones dadas, en cada ruta proporcionada.

28.3.2 How venvs work

When a Python interpreter is running from a virtual environment, *sys.prefix* and *sys.exec_prefix* point to the directories of the virtual environment, whereas *sys.base_prefix* and *sys.base_exec_prefix* point to those of the base Python used to create the environment. It is sufficient to check `sys.prefix == sys.base_prefix` to determine if the current interpreter is running from a virtual environment.

A virtual environment may be «activated» using a script in its binary directory (`bin` on POSIX; `Scripts` on Windows). This will prepend that directory to your `PATH`, so that running **`python`** will invoke the environment's Python interpreter and you can run installed scripts without having to use their full path. The invocation of the activation script is platform-specific (*<venv>* must be replaced by the path to the directory containing the virtual environment):

Plataforma	Shell	Comando para activar el entorno virtual
POSIX	bash/zsh	`$ source <venv>/bin/activate`
	fish	`$ source <venv>/bin/activate.fish`
	csh/tcsh	`$ source <venv>/bin/activate.csh`
	PowerShell	`$ <venv>/bin/Activate.ps1`
Windows	cmd.exe	`C:\> <venv>\Scripts\activate.bat`
	PowerShell	`PS C:\> <venv>\Scripts\Activate.ps1`

Nuevo en la versión 3.4: **`fish`** and **`csh`** activation scripts.

Nuevo en la versión 3.8: Scripts de activación de PowerShell instalados en POSIX para compatibilidad con PowerShell Core.

You don't specifically *need* to activate a virtual environment, as you can just specify the full path to that environment's Python interpreter when invoking Python. Furthermore, all scripts installed in the environment should be runnable without activating it.

In order to achieve this, scripts installed into virtual environments have a «shebang» line which points to the environment's Python interpreter, i.e. `#!/<path-to-venv>/bin/python`. This means that the script will run with that interpreter regardless of the value of `PATH`. On Windows, «shebang» line processing is supported if you have the launcher installed. Thus, double-clicking an installed script in a Windows Explorer window should run it with the correct interpreter without the environment needing to be activated or on the `PATH`.

When a virtual environment has been activated, the `VIRTUAL_ENV` environment variable is set to the path of the environment. Since explicitly activating a virtual environment is not required to use it, `VIRTUAL_ENV` cannot be relied upon to determine whether a virtual environment is being used.

Advertencia: Because scripts installed in environments should not expect the environment to be activated, their shebang lines contain the absolute paths to their environment's interpreters. Because of this, environments are inherently non-portable, in the general case. You should always have a simple means of recreating an environment (for example, if you have a requirements file `requirements.txt`, you can invoke `pip install -r requirements.txt` using the environment's `pip` to install all of the packages needed by the environment). If for any reason you need to move the environment to a new location, you should recreate it at the desired location and delete the one at the old location. If you move an environment because you moved a parent directory of it, you should recreate the environment in its new location. Otherwise, software installed into the environment may not work as expected.

You can deactivate a virtual environment by typing `deactivate` in your shell. The exact mechanism is platform-specific and is an internal implementation detail (typically, a script or shell function will be used).

28.3.3 API

El método de alto nivel descrito anteriormente utiliza una sencilla API que proporciona mecanismos para que les 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*, *upgrade_deps=False*)

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 de caracteres que se utilizará después de que se active el entorno virtual (el valor predeterminado es `None`, lo que significa que se utilizaría el nombre del directorio del entorno). Si se proporciona la cadena especial `"."`, el nombre base del directorio actual se utiliza como indicador.
- `upgrade_deps` – actualiza los módulos venv base a lo último en PyPI

Distinto en la versión 3.4: Añadido el parámetro `with_pip`

Nuevo en la versión 3.6: Añadido el parámetro `prompt`

Nuevo en la versión 3.9: Se agregó el parámetro `upgrade_deps`

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*)
Creates the environment directory and all necessary subdirectories that don't already exist, and returns a context object. This context object is just a holder for attributes (such as paths) for use by the other methods. If the *EnvBuilder* is created with the arg `clear=True`, contents of the environment directory will be cleared and then all necessary subdirectories will be recreated.

The returned context object is a *types.SimpleNamespace* with the following attributes:

- `env_dir` - The location of the virtual environment. Used for `__VENV_DIR__` in activation scripts (see *install_scripts()*).
- `env_name` - The name of the virtual environment. Used for `__VENV_NAME__` in activation scripts (see *install_scripts()*).
- `prompt` - The prompt to be used by the activation scripts. Used for `__VENV_PROMPT__` in activation scripts (see *install_scripts()*).
- `executable` - The underlying Python executable used by the virtual environment. This takes into account the case where a virtual environment is created from another virtual environment.
- `inc_path` - The include path for the virtual environment.
- `lib_path` - The purelib path for the virtual environment.
- `bin_path` - The script path for the virtual environment.
- `bin_name` - The name of the script path relative to the virtual environment location. Used for `__VENV_BIN_NAME__` in activation scripts (see *install_scripts()*).
- `env_exe` - The name of the Python interpreter in the virtual environment. Used for `__VENV_PYTHON__` in activation scripts (see *install_scripts()*).
- `env_exec_cmd` - The name of the Python interpreter, taking into account filesystem redirections. This can be used to run Python in the virtual environment.

Distinto en la versión 3.12: The attribute `lib_path` was added to the context, and the context object was documented.

Distinto en la versión 3.11: The *venv sysconfig installation scheme* is used to construct the paths of the created directories.

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.

upgrade_dependencies(*context*)
Actualiza los paquetes de dependencia principales de venv (actualmente `pip` y `setuptools`) en el entorno. Esto se hace desembolsando el ejecutable `pip` en el entorno.

Nuevo en la versión 3.9.

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*, *upgrade_deps=False*)
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`

Distinto en la versión 3.9: Se agregó el parámetro `upgrade_deps`

28.3.4 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):
        """
        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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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()
        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.
```

(continué en la próxima página)

(proviene de la página anterior)

```
        """
        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.")
        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.')
```

(continué en la próxima página)

(proviene de la página anterior)

```
        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» is 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')
```

To update the file in place, do the replacement in memory using a *BytesIO* object, and then overwrite the source afterwards. Note that there is a risk when overwriting a file in place that an error will result in the loss of the original file. This code does not protect against such errors, but production code should do so. Also, this method will only work if the archive fits in memory:

```
>>> 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 */
)
#else
int wmain()
#endif
{
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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. In 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/interprete`).

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 subdirectorio). 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.

CAPÍTULO 29

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*)
: Agrega el *hook* invocable a la lista de enlaces de inspección activos para el (sub)intérprete actual.

  Cuando se genera un evento de auditoría a través de la función *`sys.audit()`*, cada hook será llamado en el orden en que se agregó con el nombre del evento y la tupla de argumentos. Los hooks nativos agregados por `PySys_AddAuditHook()` se llaman primero, seguidos de los hooks agregados en el (sub)intérprete actual. Los hooks pueden registrar un evento, lanzar una excepción para abortar la operación, o terminar el proceso completamente.

  Note that audit hooks are primarily for collecting information about internal or otherwise unobservable actions, whether by Python or libraries written in Python. They are not suitable for implementing a «sandbox». In particular, malicious code can trivially disable or bypass hooks added using this function. At a minimum, any security-sensitive hooks must be added using the C API `PySys_AddAuditHook()` before initialising the runtime, and any modules allowing arbitrary memory modification (such as *`ctypes`*) should be completely removed or closely monitored.

  Llamar *`sys.addaudithook()`* lanzará por si mismo un evento de auditoria llamado `sys.addaudithook` sin argumentos. Si uno de los hooks existentes lanza un excepción derivada de

RuntimeError, el nuevo hook no será agregado y la excepción será suprimida. Como resultado, los que llaman no podrán asumir que el hook a sido agregado a menos que ellos controlen todos los hooks existentes.

Consulte la *tabla de eventos de auditoría* para todos los eventos lanzados por CPython, y **PEP 578** para discusión del diseño original.

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.

Detalles de implementación de CPython: 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*.

También ver *sys.orig_argv*.

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*)

Activar un evento de auditoría y activar cualquier hook de auditoría activo. *event* es una cadena que identifica el evento, y *args* puede contener argumentos opcionales con más información sobre el evento. El número y los tipos de argumentos para un evento determinado se consideran una API pública y estable y no deberían modificarse entre versiones.

Por ejemplo, un evento de auditoría se llama `os.chdir`. Este evento tiene un argumento llamado *path* que contendrá el nuevo directorio de trabajo solicitado.

sys.audit() llamará a los ganchos de auditoría existentes, pasando el nombre del evento y los argumentos, y volverá a lanzar la primera excepción de cualquier hook. En general, si se lanza una excepción, no debería ser manejada y el proceso debería terminar lo más rápidamente posible. Esto permite que las implementaciones de los hook decidan cómo responder a determinados eventos: pueden limitarse a registrar el evento o abortar la operación lanzando una excepción.

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 contiene 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.)

También ver la lista *sys.stdlib_module_names*.

sys.**call_tracing**(*func*, *args*)
Llame 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**()
Borre 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.**_current_exceptions**()
Retorna un diccionario que mapea el identificador de cada hilo a la excepción mas alta actualmente activa en ese hilo en el tiempo que la función fue llamada. Si un hilo no esta manejando una excepción en el momento, no se incluye en el resultado del diccionario.

Esto es más útil para la elaboración de perfiles estadísticos.

Esta función debe usarse solo para fines internos y especializados.

Lanza un *auditing event* `sys._current_exceptions` 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 adi-

cionales (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.

If Python is built in debug mode (`configure --with-pydebug option`), it also performs some expensive internal consistency checks.

Nuevo en la versión 3.3.

Detalles de implementación de CPython: 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**
If this is set (not None), Python will write bytecode-cache .pyc files to (and read them from) a parallel directory tree rooted at this directory, rather than from __pycache__ directories in the source code tree. Any __pycache__ directories in the source code tree will be ignored and new .pyc files written within the pycache prefix. Thus if you use *compileall* as a pre-build step, you must ensure you run it with the same pycache prefix (if any) that you will use at runtime.

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 retorna 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/python`*XY*`/config`, y los módulos de la biblioteca compartida se instalan en *exec_prefix*`/lib/python`*XY*`/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*])
Raise a *SystemExit* exception, signaling an intention to exit the interpreter.

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.

Since *exit()* ultimately «only» raises an exception, it will only exit the process when called from the main thread, and the exception is not intercepted. Cleanup actions specified by finally clauses of `try` statements are honored, and it is possible to intercept the exit attempt at an outer level.

Distinto en la versión 3.6: Si se produce un error en la limpieza después de que el intérprete de Python haya detectado `SystemSalir` (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
`debug`	`-d`
inspect	`-i`
`interactive`	`-i`
`isolated`	`-I`
`optimize`	`-O` o `-OO`
dont_write_bytecode	`-B`
`no_user_site`	`-s`
`no_site`	`-S`
`ignore_environment`	`-E`
`verbose`	`-v`
`bytes_warning`	`-b`
`quiet`	`-q`
`hash_randomization`	`-R`
`dev_mode`	`-X dev` (*Modo de desarrollo de Python*)
`utf8_mode`	`-X utf8`
`int_max_str_digits`	`-X int_max_str_digits` (*integer string conversion length limitation*)

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 *Modo de Desarrollo de Python* y el atributo `utf8_mode` para la bandera `-X utf8`.

Distinto en la versión 3.10.7: 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
`epsilon`	`DBL_EPSILON`	diferencia entre 1.0 y el menor valor mayor que 1.0 que es representable como flotante Vea también *`math.ulp()`*.
`dig`	`DBL_DIG`	número máximo de dígitos decimales que se pueden representar fielmente en un flotante; véase a continuación
`mant_dig`	`DBL_MANT_DIG`	precisión de flotantes: el número de dígitos base-`radix` en el significando de un flotante
`max`	`DBL_MAX`	máximo punto flotante positivo representable
`max_exp`	`DBL_MAX_EXP`	entero máximo *e* tal que `radix**(e-1)` es un flotante finito representable
`max_10_exp`	`DBL_MAX_10_EXP`	entero máximo *e* tal que `10**e` está en el rango de flotantes finitos representables
`min`	`DBL_MIN`	flotante *normalizado* mínimo representable positivo Usa *`math.ulp(0.0)`* para obtener el menor flotante positivo *denormalizado* representable.
`min_exp`	`DBL_MIN_EXP`	entero mínimo *e* tal que `radix**(e-1)` es un flotante normalizado
`min_10_exp`	`DBL_MIN_10_EXP`	entero mínimo *e* tal que `10**e` es un valor flotante normalizado
`radix`	`FLT_RADIX`	base de representación de exponente
`rounds`	`FLT_ROUNDS`	integer representing the rounding mode for floating-point arithmetic. This reflects the value of the system `FLT_ROUNDS` macro at interpreter startup time: `-1` indeterminable, `0` toward zero, `1` to nearest, `2` toward positive infinity, `3` toward negative infinity All other values for `FLT_ROUNDS` characterize implementation-defined rounding behavior.

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.**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**()
Get the *filesystem encoding*: the encoding used with the *filesystem error handler* to convert between Unicode filenames and bytes filenames. The filesystem error handler is returned from *getfilesystemencodeerrors()*.

Para la mejor compatibilidad, str debe ser usado por nombres de archivos en todos los casos, aunque representar nombres de archivos en bytes también es soportado. Funciones que acepta o retornan nombres de archivos deben soportar tanto str como bytes e internamente convertir el valor a representación preferida por el sistema.

os.fsencode() y *os.fsdecode()* deben usarse para garantizar que se utilizan la codificación correcta y el modo de errores.

El *codificación del sistema de archivos y manejo de errores* está configurado en el inicio de Python por la función `PyConfig_Read()`: ver `filesystem_encoding` y `filesystem_errors` que son miembros de `PyConfig`.

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'` si *Modo UTF-8 de Python* esta habilitado.

sys.**getfilesystemencodeerrors**()
Obtener la *codificación de sistema de archivos*: La codificación usada con el *gestor de errores de sistema de*

archivos para convertir nombres de archivos en Unicode a nombres de archivos en bytes. El gestor de errores del sistema de archivos es retornado desde `getfilesystemencoding()`.

`os.fsencode()` y `os.fsdecode()` deben usarse para garantizar que se utilizan la codificación correcta y el modo de errores.

El *codificación del sistema de archivos y manejo de errores* está configurado en el inicio de Python por la función `PyConfig_Read()`: ver `filesystem_encoding` y `filesystem_errors` que son miembros de `PyConfig`.

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 3.10.7.

`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()`.

Note that the returned value may not actually reflect how many references to the object are actually held. Consequently, do not rely on the returned value to be accurate, other than a value of 0 or 1.

`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 lanzará 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.

Detalles de implementación de CPython: 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()`.

Detalles de implementación de CPython: 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 retorna la versión principal, la versión secundaria y el número de compilación del sistema operativo actual, en lugar de la versión que se está emulando para el proceso. Está diseñado para su uso en el registro en lugar de para la detección de características.

Nota: *platform_version* deriva la versión desde kernel32.dll que puede ser de una versión diferente a la versión de SO. Por favor usar el módulo *platform* para obtener una versión de SO precisa.

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**()
Returns an *asyncgen_hooks* object, which is similar to a *namedtuple* of the form `(firstiter, finalizer)`, where *firstiter* and *finalizer* are expected to be either `None` or functions which take an *asynchronous generator iterator* as an argument, and are used to schedule finalization of an asynchronous generator by an event loop.

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 corrutina 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
`width`	ancho en bits usado para valores hash
`modulus`	primer módulo P utilizado para el esquema de hash numérico
`inf`	valor hash retornado para un infinito positivo
`nan`	(este atributo ya no se encuentra en uso)
`imag`	multiplicador utilizado para la parte imaginaria de un número complejo
`algorithm`	nombre del algoritmo para el hash de str, bytes y memoryview
`hash_bits`	tamaño de salida interno del algoritmo hash
`seed_bits`	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
`bits_per_digit`	número de bits retenidos en cada dígito. Los enteros de Python se almacenan internamente en la base `2**int_info.bits_per_digit`
`sizeof_digit`	tamaño en bytes del tipo C utilizado para representar un dígito
`default_max_str_digits`	default value for *`sys.get_int_max_str_digits()`* when it is not otherwise explicitly configured.
`str_digits_check_threshold`	minimum non-zero value for *`sys.set_int_max_str_digits()`*, `PYTHONINTMAXSTRDIGITS`, or `-X int_max_str_digits`.

Nuevo en la versión 3.1.

Distinto en la versión 3.10.7: 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**
A list of *meta path finder* objects that have their *find_spec()* methods called to see if one of the objects can find the module to be imported. By default, it holds entries that implement Python's default import semantics. The *find_spec()* method is called with at least the absolute name of the module being imported. If the module to be imported is contained in a package, then the parent package's __path__ attribute is passed in as a second argument. The method returns a *module spec*, or `None` if the module cannot be found.

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**
This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. However, replacing the dictionary will not necessarily work as expected and deleting essential items from the dictionary may cause Python to fail. If you want to iterate over this global dictionary always use `sys.modules.copy()` or `tuple(sys.modules)` to avoid exceptions as its size may change during iteration as a side effect of code or activity in other threads.

sys.**orig_argv**
La lista de los argumentos originales de la línea de comando que son pasados al ejecutable de Python.

También ver *sys.argv*.

Nuevo en la versión 3.10.

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.**platlibdir**
Nombre del directorio de la biblioteca específica de la plataforma. Se utiliza para construir la ruta de la biblioteca estándar y las rutas de los módulos de extensión instalados.

Es igual a "lib" en la mayoría de las plataformas. En Fedora y SuSE, es igual a "lib64" en plataformas de 64 bits, lo que da las siguientes rutas sys.path (donde X.Y es la versión mayor.menor de Python):

- /usr/lib64/pythonX.Y/: Biblioteca estándar (como os.py del módulo *os*)
- /usr/lib64/pythonX.Y/lib-dynload/: Módulos de extensión C de la biblioteca estándar (como el módulo *errno*, el nombre exacto del archivo es específico de la plataforma)

- `/usr/lib/pythonX.Y/site-packages/` (utilice siempre `lib`, no *sys.platlibdir*): Módulos de terceros
- `/usr/lib64/pythonX.Y/site-packages/`: Módulos de extensión C de paquetes de terceros

Nuevo en la versión 3.9.

sys.prefix
A string giving the site-specific directory prefix where the platform independent Python files are installed; on Unix, the default is `/usr/local`. This can be set at build time with the `--prefix` argument to the **configure** script. See *Rutas de instalación* for derived paths.

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.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(*maxdigits*)
Set the *integer string conversion length limitation* used by this interpreter. See also *get_int_max_str_digits()*.

Nuevo en la versión 3.10.7.

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.

Detalles de implementación de CPython: 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 set *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 del sistema de archivos y manejo 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.

También ver *sys.getfilesystemencoding()* y *sys.getfilesystemencodeerrors()*.

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 y manejo de error son inicializado desde PyConfig.stdio_encoding y PyConfig.stdio_errors.

  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 (por ejemplo, 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. Esto pasa por defecto al sistema de *codificación local* si el proceso no esta inicialmente acoplado 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 es interactivo, el flujo de stdout se almacena en búfer de línea. En caso contrario, se almacena en búfer de bloque como los archivos de texto normales. El flujo stderr tiene un búfer de línea en ambos casos. Puede hacer que ambos flujos no tengan búfer pasando la opción de línea de comandos -u o estableciendo la variable de entorno PYTHONUNBUFFERED.

Distinto en la versión 3.9: Ahora, stderr no interactivo tiene búfer de línea en lugar de búfer completo.

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.**stdlib_module_names**
Un fronzenset de strings que contiene los nombres de los módulos de la librería estándar.

Es lo mismo para todas las plataformas. Módulos que no están disponibles en algunas plataformas y deshabilitados durante la construcción de Python también son listados. Todos los tipos de modulo son listados: Python puro, incorporados, congelados y módulos de extensión. Módulos de prueba son excluidos.

Para paquete, solo el principal es listado; sub-paquetes y sub-módulos no son listados. Por ejemplo, el paquete `email` es listado, pero el sub-paquete `email.mime` y el sub-módulo `email.message` no es listado.

También ver la lista *sys.builtin_module_names*.

Nuevo en la versión 3.10.

sys.**thread_info**
Un *named tuple* que contiene información sobre la implementación del hilo.

Atributo	Explicación
`name`	Nombre de la implementación de hilos (*thread implementation*): • `'nt'`: hilos de Windows • `'pthread'`: hilos de POSIX • `'solaris'`: hilos de Solaris
`lock`	Nombre de la implementación de bloqueo (*lock implementation*): • `'semaphore'`: un bloqueo que utiliza un semáforo • `'mutex+cond'`: un bloqueo que utiliza un mutex y una variable de condición • `None` si esta información es desconocida
version	Nombre y versión de la biblioteca de subprocesos. Es una cadena de caracteres, o `None` 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* 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`**
The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the major and minor versions of the running Python interpreter. It is provided in the *sys* module for informational purposes; modifying this value has no effect on the registry keys used by 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}
```

Detalles de implementación de CPython: 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 un 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 currently supports six schemes:

- *posix_prefix*: esquema para plataformas POSIX como Linux o macOS. Este es el esquema predeterminado que se usa cuando se instala Python o un componente.
- *posix_home*: esquema para plataformas POSIX usado cuando se usa una opción *home* en la instalación. Este esquema se usa cuando se instala un componente a través de *Disutils* con un prefijo de inicio específico.
- *posix_user*: esquema para plataformas POSIX usado cuando se instala un componente a través de Distutils y se usa la opción *user*. Este esquema define rutas ubicadas bajo el directorio de inicio del usuario.
- *nt*: esquema para plataformas NT como Windows.
- *nt_user*: esquema para plataformas NT, cuando se usa la opción *user*.
- *osx_framework_user*: scheme for macOS, when the *user* option is used.

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_default_scheme**()
Retorna el nombre del esquema predeterminado para la plataforma actual.

Nuevo en la versión 3.10: Esta función se llamaba anteriormente `_get_default_scheme()` y se consideraba un detalle de implementación.

sysconfig.**get_preferred_scheme**(*key*)
Retorna un nombre de esquema preferido para un diseño de instalación especificado por *key*.

key debe ser `"prefix"`, `"home"` o `"user"`.

El valor de retorno es un nombre de esquema que aparece en *get_scheme_names()*. Se puede pasar a funciones *sysconfig* que toman un argumento *scheme*, como *get_paths()*.

Nuevo en la versión 3.10.

sysconfig.**_get_preferred_schemes**()
Retorna un diccionario que contiene los nombres de los esquemas preferidos en la plataforma actual. Los implementadores y redistribuidores de Python pueden agregar sus esquemas preferidos al valor global de nivel de módulo `_INSTALL_SCHEMES` y modificar esta función para devolver esos nombres de esquema, p. Ej.

proporcionan diferentes esquemas para que los utilicen los administradores de paquetes de idioma y sistema, de modo que los paquetes instalados por uno no se mezclen con los del otro.

Los usuarios finales no deben utilizar esta función, sino *get_default_scheme()* y *get_preferred_scheme()* en su lugar.

Nuevo en la versión 3.10.

sysconfig.**get_path_names**()
Retorna una tupla que contiene todo 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*, lanza un *KeyError*.

sysconfig.**get_paths**([*scheme*[, *vars*[, *expand*]]])
Retorna un diccionario de 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)

macOS 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"
        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 se 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 código de nivel máximo

En Python, el nombre especial __main__ es utilizado para dos constructos importantes:

1. el nombre del entorno de máximo nivel del programa, que puede ser verificado usando la expresión __name__=='__main__'; y
2. el archivo __main__.py en paquetes de Python.

Ambos de estos mecanismos están relacionados con módulos de Python; como los usuarios interactúan con ellos y como ellos interactúan entre sí. Están explicados en detalle más abajo. Si estás empezando con los módulos de Python, mira la sección tutorial tut-modules para una introducción.

29.4.1 __name__ == '__main__'

Cuando un módulo o paquete de Python es importado, __name__ es asignado al nombre del módulo. Normalmente, este es el nombre del archivo de Python sin la extensión .py:

```
>>> import configparser
>>> configparser.__name__
'configparser'
```

Si el archivo es parte de un paquete, __name__ también incluirá la ruta del paquete padre:

```
>>> from concurrent.futures import process
>>> process.__name__
'concurrent.futures.process'
```

Sin embargo, si el módulo es ejecutado en el entorno de código de máximo nivel, su __name__ se le asigna el valor del string '__main__'.

¿Qué es el «entorno de código de máximo nivel»?

__main__ es el nombre del entorno en el cual se ejecuta el código de máximo nivel. «Código de máximo nivel» es el primer módulo de Python especificado por el usuario que empieza a ejecutarse. Es «de máximo nivel» porque importa todos los demás módulos que necesita el programa. A veces «código de máximo nivel» es llamado un *punto de entrada* a la aplicación.

El entorno de código de máximo nivel puede ser:

- el ámbito de un intérprete interactivo:

  ```
  >>> __name__
  '__main__'
  ```

- el módulo de Python pasado al intérprete de Python como un argumento de archivo:

  ```
  $ python3 helloworld.py
  Hello, world!
  ```

- el módulo o paquete de Python pasado al interprete de Python con el argumento -m:

  ```
  $ python3 -m tarfile
  usage: tarfile.py [-h] [-v] (...)
  ```

- Código de Python leído por el interprete desde input estándar:

  ```
  $ echo "import this" | python3
  The Zen of Python, by Tim Peters

  Beautiful is better than ugly.
  Explicit is better than implicit.
  ...
  ```

- Código de Python pasado al intérprete Python con el argumento -c:

  ```
  $ python3 -c "import this"
  The Zen of Python, by Tim Peters

  Beautiful is better than ugly.
  Explicit is better than implicit.
  ...
  ```

En cada una de estas situaciones, al __name__ del módulo de máximo nivel se le asigna el valor '__main__'.

En consecuencia, un módulo puede descubrir si se está ejecutando o no en el ámbito principal al verificar su propio `__name__`, lo cual permite un vocablo común para ejecutar código condicionalmente cuando el módulo no es inicializado desde una declaración de importado:

```
if __name__ == '__main__':
    # Execute when the module is not initialized from an import statement.
    ...
```

Ver también:

Para una vista más detallada de como `__name__` es asignado en todas las situaciones, ver la sección tutorial tut-modules.

Uso Idiomático

Some modules contain code that is intended for script use only, like parsing command-line arguments or fetching data from standard input. If a module like this was imported from a different module, for example to unit test it, the script code would unintentionally execute as well.

Aquí es donde es útil usar el código de bloque `if __name__=='__main__'`. El código dentro desde bloque no se ejecutará a menos que el módulo se ejecute en el entorno de máximo nivel.

Poner cuantas menos declaraciones posible en el bloque debajo de `if __name__=='__main__'` puede mejorar la claridad y validez del código. Mas a menudo, la función llamada `main` encapsula el comportamiento principal del programa:

```
# echo.py

import shlex
import sys

def echo(phrase: str) -> None:
   """A dummy wrapper around print."""
   # for demonstration purposes, you can imagine that there is some
   # valuable and reusable logic inside this function
   print(phrase)

def main() -> int:
    """Echo the input arguments to standard output"""
    phrase = shlex.join(sys.argv)
    echo(phrase)
    return 0

if __name__ == '__main__':
    sys.exit(main())  # next section explains the use of sys.exit
```

Note que si el módulo no encapsulase el código dentro de la función `main` pero en vez lo pusiese directo dentro del bloque `if __name__=='__main__'`, la variable `phrase` sería global al módulo entero. Esto está propenso a generar errores ya que otras funciones dentro del módulo pudiesen estar usando involuntariamente la variable global en lugar de un nombre local. Una función `main` resuelve este problema.

Usar una función `main` tiene el beneficio añadido de que la función `echo` está aislada y es importable en otros sitios. Cuando `echo.py` es importado, las funciones `echo` y `main` serán definidas, pero ninguna de ellas será llamada porque `__name!='__main__'`.

Consideraciones de empaquetado

Las funciones `main` se utilizan a menudo para crear líneas de comando al especificarlas como puntos de entrada para scripts de terminal. Cuando esto se hace, pip inserta la llamada de la función a un script plantilla, donde el valor retornado de `main` se pasa a *`sys.exit()`*. Por ejemplo:

```
sys.exit(main())
```

Dado que la llamada a `main` está dentro de *`sys.exit()`*, la expectativa es que tu función devolverá un valor aceptable como una entrada a *`sys.exit()`*; típicamente, un int o `None` (que se retorna implícitamente si tu función no tiene una declaración de retorno).

Al seguir pro-activamente esta convención nosotros mismo, nuestro módulo tendrá el mismo comportamiento cuando se ejecuta directamente (es decir, `python3 echo.py`) que si luego lo empaquetamos como un punto de entrada de script de terminal en un paquete instalable mediante pip.

En particular, ten cuidado al devolver cadenas de texto con tu función `main`. *`sys.exit()`* interpretará un argumento de cadena de texto como un mensaje de fallo, entonces tu programa tendrá un código de salida de `1`, indicando fallo, y la cadena de texto será escrita a *`sys.stderr`*. El ejemplo `echo.py` de antes muestra como usar la convención `sys.exit(main())`.

Ver también:

Python Packaging User Guide contiene una colección de tutoriales y referencias sobre como distribuir e instalar paquetes de Python con herramientas modernas.

29.4.2 `__main__.py` en paquetes de Python

Si no estás familiarizado con paquetes de Python, ver la sección tut-packages del tutorial. Comúnmente, el archivo `__main__.py` es utilizado para proveer una interfaz de línea de comando para un paquete. Considera el siguiente paquete hipotético, «*bandclass*»:

```
bandclass
  ├── __init__.py
  ├── __main__.py
  └── student.py
```

`__main__.py` será ejecutado cuando el paquete sea invocado directamente desde la línea de comandos usando el indicador `-m`. Por ejemplo:

```
$ python3 -m bandclass
```

Este comando causará que `__main__.py` se ejecute. El como se use este mecanismo dependerá de la naturaleza del paquete que estás escribiendo, pero en este caso hipotético, puede tener sentido permitir que el profesor busque estudiantes:

```
# bandclass/__main__.py

import sys
from .student import search_students

student_name = sys.argv[2] if len(sys.argv) >= 2 else ''
print(f'Found student: {search_students(student_name)}')
```

Note that `from .student import search_students` is an example of a relative import. This import style can be used when referencing modules within a package. For more details, see intra-package-references in the tut-modules section of the tutorial.

Uso Idiomático

Los contenidos de `__main__.py` no están típicamente acotados dentro de bloques `if __name__=='__main__'`. En cambio, esos archivos se mantienen cortos, funciones para ejecutar desde otros módulos. A esos otros módulos se les puede fácilmente realizar pruebas unitarias y son apropiadamente re-utilizables.

Si se usa, un bloque `if __name__=='__main__'` seguirá funcionando como se espera para un archivo `__main__.py` dentro de un paquete, porque su atributo `__name__` incluirá la ruta del paquete si es importado:

```
>>> import asyncio.__main__
>>> asyncio.__main__.__name__
'asyncio.__main__'
```

This won't work for `__main__.py` files in the root directory of a .zip file though. Hence, for consistency, minimal `__main__.py` like the *venv* one mentioned below are preferred.

Ver también:

See *venv* for an example of a package with a minimal `__main__.py` in the standard library. It doesn't contain a `if __name__ == '__main__'` block. You can invoke it with `python -m venv [directory]`.

Ver *runpy* para más detalles sobre el indicador `-m` para el interprete ejecutable.

Ver *zipapp* para más información sobre como ejecutar aplicaciones empaquetadas como archivos *.zip*. En este caso Python busca un archivo `__main__.py` en el directorio raíz del archivo comprimido.

29.4.3 `import __main__`

Independientemente de con cual módulo se ha iniciado un programa de Python, otros módulos que están siendo ejecutados dentro del mismo programa pueden importar el ámbito del entorno de máximo nivel (*namespace*) al importar el módulo `__main__`. Esto no importa un archivo `__main__.py` pero en su lugar cualquier módulo que recibió el nombre especial `'__main__'`.

Acá hay un módulo ejemplo que consume el nombre de espacio `__main__`:

```
# namely.py

import __main__

def did_user_define_their_name():
    return 'my_name' in dir(__main__)

def print_user_name():
    if not did_user_define_their_name():
        raise ValueError('Define the variable `my_name`!')

    if '__file__' in dir(__main__):
        print(__main__.my_name, "found in file", __main__.__file__)
    else:
        print(__main__.my_name)
```

Ejemplo del uso de este módulo puede ser:

```
# start.py

import sys

from namely import print_user_name

# my_name = "Dinsdale"
```

(continué en la próxima página)

(proviene de la página anterior)

```
def main():
    try:
        print_user_name()
    except ValueError as ve:
        return str(ve)

if __name__ == "__main__":
    sys.exit(main())
```

Si ahora iniciamos nuestro programa el resultado sería así:

```
$ python3 start.py
Define the variable `my_name`!
```

El código de salida del programa sería 1, indicando un error. Des-comentando la línea con `my_name = "Dinsdale"` arregla el programa y ahora sale con un código de estado 0, indicando éxito:

```
$ python3 start.py
Dinsdale found in file /path/to/start.py
```

Note que importar `__main__` no causa ningún problema de involuntariamente ejecutar código de máximo nivel que ha sido pensado para uso por scripts que es puesto en el bloque `if __name__=="__main__"` del módulo `start`. ¿Por qué funciona esto?

Python inserta un módulo `__main__` vacío en *`sys.modules`* al inicio del intérprete, y lo puebla ejecutando código de máximo nivel. En nuestro ejemplo este es el módulo `start` que corre línea a línea e importa `namely`. A su vez, `namely` importa `__main__` (que es en verdad `start`). ¡Es un ciclo de importado! Afortunadamente, como el módulo parcialmente poblado `__main__` está presente en *`sys.modules`*, Python pasa eso a `namely`. Ver Special considerations for __main__ en la referencia del sistema para información detallada de como funciona.

El REPL Python es otro ejemplo de un «entorno de máximo nivel», por lo tanto, cualquier cosa definida en el REPL se hace parte del ámbito `__main__`:

```
>>> import namely
>>> namely.did_user_define_their_name()
False
>>> namely.print_user_name()
Traceback (most recent call last):
...
ValueError: Define the variable `my_name`!
>>> my_name = 'Jabberwocky'
>>> namely.did_user_define_their_name()
True
>>> namely.print_user_name()
Jabberwocky
```

Note que en este caso el ámbito `__main__``no contiene un atributo ``__file__` ya que es interactivo.

El ámbito `__main__` es utilizado en la implementación de *`pdb`* y *`rlcompleter`*.

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
Warning	Esta es la clase principal para todas las clases que pertenecen a la categoría de advertencia. Es una subclase de *Exception*.
UserWarning	La categoría por defecto para *warn()*.
DeprecationWarning	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 __main__).
SyntaxWarning	Categoría principal para las advertencias sobre características sintácticas dudosas.
RuntimeWarning	Categoría principal para las advertencias sobre características dudosas de tiempo de ejecución.
FutureWarning	Categoría principal para las advertencias sobre características obsoletas cuando esas advertencias están destinadas a los usuarios finales de aplicaciones escritas en Python.
PendingDeprecationWarning	Categoría principal para las advertencias sobre las características que serán desaprobadas en el futuro (ignoradas por defecto).
ImportWarning	Categoría principal para las advertencias que se activan durante el proceso de importación de un módulo (se ignoran por defecto).
UnicodeWarning	Categoría principal para las advertencias relacionadas con Unicode.
BytesWarning	Categoría principal para las advertencias relacionadas con *bytes* y *bytearray*.
ResourceWarning	Base category for warnings related to resource usage (ignored by default).

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 coinciden
"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* is a string containing a regular expression that the start of the warning message must match, case-insensitively. In -W and PYTHONWARNINGS, *message* is a literal string that the start of the warning message must contain (case-insensitively), ignoring any whitespace at the start or end of *message*.

- *category* es una clase (una subclase de *Warning*) de la cual la categoría de advertencia debe ser una subclase para poder coincidir.
- *module* is a string containing a regular expression that the start of the fully qualified module name must match, case-sensitively. In -W and PYTHONWARNINGS, *module* is a literal string that the fully qualified module name must be equal to (case-sensitively), ignoring any whitespace at the start or end of *module*.
- *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 lanzamos 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"
```

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 una compilación de depuración, la lista de filtros de advertencia predeterminados 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 issubclass(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 ignorar 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 Contenidos del módulo

@dataclasses.**dataclass**(*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*, *match_args=True*, *kw_only=False*, *slots=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,
           match_args=True, kw_only=False, slots=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__()`.

  Si `__hash__()` no está definido explícitamente, o si está configurado como `None`, entonces *`dataclass()`* *puede* agregar un método implícito `__hash__()`. Aunque no se recomienda, puede forzar un *`dataclass()`* a crear un método `__hash__()` con `unsafe_hash=True`. Este podría ser el caso si su clase es lógicamente inmutable pero, no obstante, puede mutar. Este es un caso de uso especializado y debe considerarse detenidamente.

  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.

- `match_args`: si es verdadero (el valor predeterminado es `True`), la tupla `__match_args__` se creará a partir de la lista de parámetros para el método `__init__()` generado (incluso si no se genera `__init__()`, consulte más arriba). Si es falso, o si `__match_args__` ya está definido en la clase, no se generará `__match_args__`.

  Nuevo en la versión 3.10.

- `kw_only`: si es verdadero (el valor predeterminado es `False`), todos los campos se marcarán solo como palabra clave. Si un campo está marcado como solo de palabra clave, el único efecto es que el parámetro `__init__()` generado a partir de un campo de solo palabra clave debe especificarse con una palabra clave cuando se llama a `__init__()`. No hay ningún efecto sobre ningún otro aspecto de las clases de datos. Consulte la entrada del glosario *parameter* para obtener más detalles. Consulte también la sección *`KW_ONLY`*.

  Nuevo en la versión 3.10.

- `slots`: si es verdadero (el valor predeterminado es `False`), se generará el atributo `__slots__` y se devolverá una nueva clase en lugar de la original. Si `__slots__` ya está definido en la clase, se genera *`TypeError`*.

  Nuevo en la versión 3.10.

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, init=True, repr=True, hash=None, compare=True, metadata=None, kw_only=MISSING)`

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 que se usa para detectar si el usuario proporciona algunos parámetros. Este centinela se utiliza porque `None` es un valor válido para algunos parámetros con un significado distinto. Ningún código 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.
- `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.
- `compare`: Si es verdadero (por defecto), este campo es incluido en los métodos de comparación generados (`__eq__()`, `__gt__()` y otros).
- `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*.

- `kw_only`: si es verdadero, este campo se marcará como solo palabra clave. Se utiliza cuando se calculan los parámetros del método `__init__()` generado.

  Nuevo en la versión 3.10.

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**(*obj*, *, *dict_factory=dict*)

Converts the dataclass `obj` to a dict (by using the factory function `dict_factory`). Each dataclass is converted to a dict of its fields, as `name: value` pairs. dataclasses, dicts, lists, and tuples are recursed into. Other objects are copied with *`copy.deepcopy()`*.

Example of using *`asdict()`* on nested dataclasses:

```
@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}]}
```

To create a shallow copy, the following workaround may be used:

```
dict((field.name, getattr(obj, field.name)) for field in fields(obj))
```

asdict() raises *TypeError* if `obj` is not a dataclass instance.

dataclasses.**astuple**(*obj*, *, *tuple_factory=tuple*)

Converts the dataclass `obj` to a tuple (by using the factory function `tuple_factory`). Each dataclass is converted to a tuple of its field values. dataclasses, dicts, lists, and tuples are recursed into. Other objects are copied with *copy.deepcopy()*.

Continuando con el ejemplo anterior:

```
assert astuple(p) == (10, 20)
assert astuple(c) == ([(0, 0), (10, 4)],)
```

To create a shallow copy, the following workaround may be used:

```
tuple(getattr(obj, field.name) for field in dataclasses.fields(obj))
```

astuple() raises *TypeError* if `obj` is not a dataclass instance.

dataclasses.**make_dataclass**(*cls_name*, *fields*, *, *bases=()*, *namespace=None*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*, *match_args=True*, *kw_only=False*, *slots=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**(*obj*, */*, ***changes*)

Creates a new object of the same type as `obj`, replacing fields with values from `changes`. If `obj` is not a Data Class, raises *TypeError*. If values in `changes` do not specify fields, raises *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`**(*obj*)

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)
```

`dataclasses.`**`MISSING`**

Un valor centinela que significa que falta un default o default_factory.

`dataclasses.`**`KW_ONLY`**

Un valor centinela utilizado como anotación de tipo. Cualquier campo después de un pseudocampo con el tipo de *`KW_ONLY`* se marca como campos de solo palabras clave. Tenga en cuenta que, de lo contrario, un pseudocampo de tipo *`KW_ONLY`* se ignora por completo. Esto incluye el nombre de dicho campo. Por convención, se utiliza un nombre de _ para un campo *`KW_ONLY`*. Los campos de solo palabras clave significan parámetros `__init__()` que deben especificarse como palabras clave cuando se crea una instancia de la clase.

En este ejemplo, los campos `y` y `z` se marcarán como campos de solo palabras clave:

```
@dataclass
class Point:
  x: float
  _: KW_ONLY
  y: float
  z: float

p = Point(0, y=1.5, z=2.0)
```

En una sola clase de datos, es un error especificar más de un campo cuyo tipo es *`KW_ONLY`*.

Nuevo en la versión 3.10.

exception `dataclasses.`**`FrozenInstanceError`**

Se genera cuando se llama a un `__setattr__()` o `__delattr__()` definido implícitamente en una clase de datos que se definió con `frozen=True`. Es una subclase de *`AttributeError`*.

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
```

El método `__init__()` generado por *dataclass()* no llama a los métodos `__init__()` de la clase base. Si la clase base tiene un método `__init__()` que debe llamarse, es común llamar a este método en un método `__post_init__()`:

```
@dataclass
class Rectangle:
    height: float
    width: float

@dataclass
class Square(Rectangle):
    side: float

    def __post_init__(self):
        super().__init__(self.side, self.side)
```

Sin embargo, tenga en cuenta que, en general, no es necesario llamar a los métodos `__init__()` generados por la clase de datos, ya que la clase de datos derivada se encargará de inicializar todos los campos de cualquier clase base que sea una clase de datos en sí.

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

One of the few places where *dataclass()* actually inspects the type of a field is to determine if a field is a class variable as defined in **PEP 526**. It does this by checking if the type of the field is `typing.ClassVar`. If a field is a `ClassVar`, it is excluded from consideration as a field and is ignored by the dataclass mechanisms. Such `ClassVar` pseudo-fields are not returned by the module-level *fields()* function.

29.6.4 Variable de solo inicialización

Another place where *dataclass()* inspects a type annotation is to determine if a field is an init-only variable. It does this by seeing if the type of a field is of type `dataclasses.InitVar`. If a field is an `InitVar`, it is considered a pseudo-field called an init-only field. As it is not a true field, it is not returned by the module-level *fields()* function. Init-only fields are added as parameters to the generated `__init__()` method, and are passed to the optional `__post_init__()` method. They are not otherwise used by dataclasses.

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 = None
    database: InitVar[DatabaseType | None] = 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
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 Reordenar los parámetros de solo palabras clave en `__init__()`

Una vez que se calculan los parámetros necesarios para `__init__()`, todos los parámetros de solo palabras clave se mueven después de todos los parámetros normales (no solo de palabras clave). Este es un requisito de cómo se implementan los parámetros de solo palabras clave en Python: deben ir después de los parámetros que no son solo de palabras clave.

En este ejemplo, `Base.y`, `Base.w` y `D.t` son campos de solo palabras clave, y `Base.x` y `D.z` son campos regulares:

```
@dataclass
class Base:
    x: Any = 15.0
    _: KW_ONLY
    y: int = 0
    w: int = 1

@dataclass
class D(Base):
    z: int = 10
    t: int = field(kw_only=True, default=0)
```

El método `__init__()` generado para `C` se verá como:

```
def __init__(self, x: Any = 15.0, z: int = 10, *, y: int = 0, w: int = 1, t: int =␣
↪0):
```

Tenga en cuenta que los parámetros se han reordenado a partir de cómo aparecen en la lista de campos: los parámetros derivados de los campos regulares son seguidos por los parámetros derivados de los campos de solo palabras clave.

El orden relativo de los parámetros de solo palabras clave se mantiene en la lista de parámetros `__init__()` reordenada.

29.6.8 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.

29.6.9 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):
        self.x += element

assert D().x is D().x
```

Esto 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 clase compartirán la misma copia de

`x`. Debido a que las clases de datos solo usan la creación normal de clases de Python, también comparten este comportamiento. No existe una forma general de que las clases de datos detecten esta condición. En cambio, el decorador *`dataclass()`* lanzará un *`TypeError`* si detecta un parámetro predeterminado de tipo `list`, `dict` o `set`. Esta es una solución parcial, pero protege contra muchos errores 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
```

29.6.10 Descriptor-typed fields

Fields that are assigned descriptor objects as their default value have the following special behaviors:

- The value for the field passed to the dataclass's `__init__` method is passed to the descriptor's `__set__` method rather than overwriting the descriptor object.
- Similarly, when getting or setting the field, the descriptor's `__get__` or `__set__` method is called rather than returning or overwriting the descriptor object.
- To determine whether a field contains a default value, `dataclasses` will call the descriptor's `__get__` method using its class access form (i.e. `descriptor.__get__(obj=None, type=cls)`. If the descriptor returns a value in this case, it will be used as the field's default. On the other hand, if the descriptor raises *`AttributeError`* in this situation, no default value will be provided for the field.

```
class IntConversionDescriptor:
  def __init__(self, *, default):
    self._default = default

  def __set_name__(self, owner, name):
    self._name = "_" + name

  def __get__(self, obj, type):
    if obj is None:
      return self._default

    return getattr(obj, self._name, self._default)

  def __set__(self, obj, value):
    setattr(obj, self._name, int(value))

@dataclass
class InventoryItem:
  quantity_on_hand: IntConversionDescriptor = IntConversionDescriptor(default=100)

i = InventoryItem()
print(i.quantity_on_hand)   # 100
i.quantity_on_hand = 2.5    # calls __set__ with 2.5
print(i.quantity_on_hand)   # 2
```

Note that if a field is annotated with a descriptor type, but is not assigned a descriptor object as its default value, the field will act like a normal field.

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.__exit__()`. Se proporciona una implementación predeterminada para `object.__enter__()` que retorna `self` mientras que `object.__exit__()` 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, **kwds):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)
```

  The function can then be used like this:

```
>>> 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:
        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.

Los administradores de contexto definidos con *`asynccontextmanager()`* se pueden utilizar como decoradores o con declaraciones `async with`:

```
import time
from contextlib import asynccontextmanager

@asynccontextmanager
async def timeit():
    now = time.monotonic()
    try:
        yield
    finally:
        print(f'it took {time.monotonic() - now}s to run')

@timeit()
async def main():
    # ... async code ...
```

Cuando se utiliza como decorador, se crea implícitamente una nueva instancia de generador en cada llamada de función. Esto permite que los administradores de contexto de otro modo «únicos»

creados por *asynccontextmanager()* cumplan con el requisito de que los administradores de contexto admitan múltiples invocaciones para ser utilizados como decoradores.

Distinto en la versión 3.10: Los administradores de contexto asíncronos creados con *asynccontextmanager()* se pueden utilizar como decoradores.

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('https://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.**aclosing**(*thing*)

Retorna un administrador de contexto asíncrono que llama al método `aclose()` de *thing* una vez completado el bloque. Esto es básicamente equivalente a:

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def aclosing(thing):
    try:
        yield thing
    finally:
        await thing.aclose()
```

Significativamente, `aclosing()` admite la limpieza determinista de generadores asincrónicos cuando salen temprano por `break` o una excepción. Por ejemplo:

```
from contextlib import aclosing

async with aclosing(my_generator()) as values:
    async for value in values:
        if value == 42:
            break
```

Este patrón garantiza que el código de salida asíncrono del generador se ejecute en el mismo contexto que sus iteraciones (de modo que las excepciones y las variables de contexto funcionen como se esperaba, y el código de salida no se ejecute después de la vida útil de alguna tarea de la que depende).

Nuevo en la versión 3.10.

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
```

También se puede utilizar como sustituto de asynchronous context managers:

```
async def send_http(session=None):
    if not session:
        # If no http session, create it with aiohttp
        cm = aiohttp.ClientSession()
    else:
        # Caller is responsible for closing the session
        cm = nullcontext(session)

    async with cm as session:
        # Send http requests with session
```

Nuevo en la versión 3.7.

Distinto en la versión 3.10: Se agregó compatibilidad con *asynchronous context manager*.

`contextlib.`**`suppress`**`(*exceptions)`

Retorna un administrador de contexto que suprime cualquiera de las excepciones especificadas si ocurren 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')
```

(continué en la próxima página)

(proviene de la página anterior)

```
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`. La secuencia de reemplazo se retorna desde el método `__enter__` y, por lo tanto, está disponible como destino de la declaración `with`:

```
with redirect_stdout(io.StringIO()) as 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
```

The class can then be used like this:

```
>>> @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.**AsyncContextDecorator**

Similar a *ContextDecorator* pero solo para funciones asincrónicas.

Ejemplo de AsyncContextDecorator:

```
from asyncio import run
from contextlib import AsyncContextDecorator

class mycontext(AsyncContextDecorator):
    async def __aenter__(self):
        print('Starting')
        return self

    async def __aexit__(self, *exc):
        print('Finishing')
        return False
```

The class can then be used like this:

```
>>> @mycontext()
... async def function():
...     print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing

>>> async def function():
...     async with mycontext():
...         print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing
```

Nuevo en la versión 3.10.

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
```

The __enter__() method returns the *ExitStack* instance, and performs no additional operations.

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 retrolla-

madas. 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*, ***kwds*)
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.

coroutine 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*, ***kwds*)
Similar a `callback()` pero espera una función de rutina.

coroutine 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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):

    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, **kwds):
        super().__init__()
        self.callback(callback, *args, **kwds)

    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 subprocesos. *redirect_stdout()*, por ejemplo, definitivamente no es seguro para subprocesos, 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:
...     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 metaclase `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 metaclase. Con esta clase, una clase de base abstracta puede ser creada simplemente derivándola desde `ABC` evitando el uso de metaclases algunas veces confusos, por ejemplo:

```
from abc import ABC

class MyABC(ABC):
    pass
```

Tenga en cuenta que el tipo de `ABC` sigue siendo `ABCMeta`, por lo tanto, heredar de `ABC` requiere las precauciones habituales con respecto al uso de metaclases, ya que la herencia múltiple puede dar lugar a conflictos de metaclases. También se puede definir una clase base abstracta pasando la palabra clave metaclase y usando `ABCMeta` directamente, por ejemplo:

```
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 metaclase 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 metaclase de `ABCMeta` tienen el siguiente método:

register(*subclass*)

Registre la *subclase* como una «subclase virtual» de esta CBA. Por ejemplo:

```
from abc import ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)
```

(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)

```
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 *subclase* 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 *subclase* se considera una subclase de esta CBA. Si retorna `False`, la *subclase* 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)

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 metaclase de la clase sea *ABCMeta* o se derive de esta. Una clase que tiene una metaclase 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.

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, solo se admite mediante la función *update_abstractmethods()*. *abstractmethod()* solo afecta a las subclases derivadas mediante herencia regular; Las «subclases virtuales» registradas con el método register() de ABC no se ven 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, arg1):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg2):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg3):
        ...

    @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.`**`abstractclassmethod`**
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, arg):
        ...
```

`@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(arg):
        ...
```

`@abc.`**`abstractproperty`**
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.

abc.**update_abstractmethods**(*cls*)
Una función para recalcular el estado de abstracción de una clase abstracta. Se debe llamar a esta función si los métodos abstractos de una clase se han implementado o cambiado después de su creación. Por lo general, esta función debe llamarse desde dentro de un decorador de clases.

Retorna *cls*, para permitir su uso como decorador de clases.

Si *cls* no es una instancia de *ABCMeta*, no hace nada.

Nota: Esta función asume que las superclases de *cls* ya están actualizadas. No actualiza subclases.

Nuevo en la versión 3.10.

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 traceback (a menos que se haya lanzado *SystemExit*) y se guarda la información de la excepción. Después de que todos los gestores de salida hayan tenido la oportunidad de ejecutarse, la última excepción a ser lanzada 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 en el apagado del intérprete. *unregister()* silenciosamente no hace nada si *func* no se registró previamente. Si *func* se ha registrado más de una vez, se eliminarán todas las apariciones de esa función en la pila de llamadas de *atexit*. Se utilizan comparaciones de igualdad (“”==””) internamente durante la cancelación del registro, por lo que las referencias de funciones no necesitan tener identidades coincidentes.

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

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()`.

Ver también:

Module `faulthandler` Used to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal.

Module `pdb` Interactive source code debugger for Python programs.

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`**(*exc*, */*[, *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):`
- imprime el tipo de excepción y *value* después del seguimiento de la 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.

Desde Python 3.10, en lugar de pasar *value* y *tb*, se puede pasar un objeto de excepción como primer argumento. Si se proporcionan *value* y *tb*, el primer argumento se ignora para proporcionar compatibilidad con versiones anteriores.

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*.

Distinto en la versión 3.10: El parámetro *etype* ha cambiado de nombre a *exc* y ahora es solo posicional.

`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**(*exc*, */*[, *value*])
Formatee la parte de excepción de un rastreo utilizando un valor de excepción como el proporcionado por `sys.last_value`. El valor de retorno es una lista de cadenas, cada una termina en una nueva línea. Normalmente, la lista contiene una sola cadena; sin embargo, para las excepciones *SyntaxError*, contiene varias líneas que (cuando se imprimen) muestran información detallada sobre dónde ocurrió el error de sintaxis. El mensaje que indica qué excepción ocurrió es siempre la última cadena de la lista.

Desde Python 3,10, en lugar de pasar *value*, un objeto de excepción se puede pasar como primer argumento. Si se proporciona *value*, el primer argumento se ignora el fin de proporcionar compatibilidad hacia atrás.

Distinto en la versión 3.10: El parámetro *etype* ha cambiado de nombre a *exc* y ahora es solo posicional.

traceback.**format_exception**(*exc*, */*[, *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*.

Distinto en la versión 3.10: El comportamiento y la firma de esta función se modificaron para coincidir con *print_exception()*.

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*, *compact=False*)
Captura una excepción para su posterior procesado. *limit*, *lookup_lines* y *capture_locals* son como para la clase *StackSummary*.

Si *compact* es verdadero, solo los datos requeridos por el método `format` de *TracebackException* se guardan en los atributos de clase. En particular, el campo `__context__` se calcula solo si `__cause__` es `None` y `__suppress_context__` es falso.

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.

end_lineno
For syntax errors - the end line number where the error occurred. Can be `None` if not present.

Nuevo en la versión 3.10.

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.

end_offset
For syntax errors - the end offset into the text where the error occurred. Can be `None` if not present.

Nuevo en la versión 3.10.

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 procesado. *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.

Distinto en la versión 3.10: Se agregó el parámetro *compact*.

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 objeto *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 tupa 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_life()

def bright_side_of_life():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print("*** print_tb:")
```

(continué en la próxima página)

(proviene de la página anterior)

```
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print("*** print_exception:")
    traceback.print_exception(exc_value, 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()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    print(repr(traceback.format_exception(exc_value)))
    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_life()
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_life()
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_life()\n',
 '  File "<doctest...>", line 7, in bright_side_of_life\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_life>]
*** format_tb:
['  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_life()\n',
 '  File "<doctest...>", line 7, in bright_side_of_life\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()
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')]
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print(repr(traceback.format_
↪stack()))\n']
```

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 print 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 StopIteration dentro de generadores*
anotaciones	3.7.0b1	TBD[1]	**PEP 563**: *Evaluación pospuesta de anotaciones*

Ver también:

future Cómo trata el compilador las importaciones futuras.

[1] `from __future__ import annotations` was previously scheduled to become mandatory in Python 3.10, but the Python Steering Council twice decided to delay the change (announcement for Python 3.10; announcement for Python 3.11). No final decision has been made yet. See also **PEP 563** and **PEP 649**.

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*.

  Lanza un *evento de auditoría* `gc.get_objects` con el argumento `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.

El recolector de basura (GC por sus siglas en inglés) clasifica los objetos en tres generaciones dependiendo de cuántos barridos de colección hayan sobrevivido. Los nuevos objetos se colocan en la generación más joven (generación `0`). Si un objeto sobrevive a una colección, se traslada a la siguiente generación anterior. Dado que la generación `2` es la generación más antigua, los objetos de esa generación permanecen allí después de una colección. Para decidir cuándo ejecutar, el recopilador realiza un seguimiento del número de asignaciones y desasignaciones de objetos desde la última recopilación. Cuando el número de asignaciones menos el número de desasignaciones supera el *treshold0*, comienza la recopilación. Inicialmente solo se examina la generación `0`. Si la generación `0` se ha examinado más de *treshold1* veces desde que se examinó la generación `1`, también se examina la generación `1`. Con la tercera generación, las cosas son un poco más complicadas, consulte Recopilación de la generación más antigua para obtener más información.

`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.

Lanza un *evento de auditoría* `gc.get_referrers` con el argumento `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.

Lanza un *evento de auditoría* `gc.get_referents` con el argumento `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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

Nuevo en la versión 3.1.

gc.**is_finalized**(*obj*)
Retorna `True` si el objeto dado ha sido finalizado por el recolector de basura, `False` en caso contrario.

```
>>> x = None
>>> class Lazarus:
...     def __del__(self):
...         global x
...         x = self
...
>>> lazarus = Lazarus()
>>> gc.is_finalized(lazarus)
False
>>> del lazarus
>>> gc.is_finalized(x)
True
```

Nuevo en la versión 3.9.

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	__doc__	cadena de caracteres de documentación
	__file__	nombre de archivo (falta para los módulos incorporados)
clase	__doc__	cadena de caracteres de documentación
	__name__	nombre con el que se definió esta clase
	__qualname__	nombre calificado
	__module__	nombre del módulo en el que se definió esta clase
método	__doc__	cadena de caracteres de documentación
	__name__	nombre con el que se definió este método
	__qualname__	nombre calificado
	__func__	objeto función que contiene la implementación del método
	__self__	instancia a la que este método está ligado, o `None`
	__module__	nombre del módulo en el cual este método fue definido
función	__doc__	cadena de caracteres de documentación
	__name__	nombre con el que se definió esta función
	__qualname__	nombre calificado
	__code__	objeto de código que contiene la función compilada *bytecode*
	__defaults__	tupla de cualquier valor por defecto para los parámetros de posición o de palabras clave
	__kwdefaults__	mapeo de cualquier valor predeterminado para parámetros de sólo palabras clave
	__globals__	namespace global en el que se definió esta función
	__builtins__	Espacio de nombres builtins
	__annotations__	mapeo de los nombres de parámetros a las anotaciones; la tecla `"return"` está reservada par
	__module__	nombre del módulo en el cual esta función fue definida
traceback	tb_frame	enmarcar el objeto a este nivel
	tb_lasti	índice del último intento de instrucción en código de bytes
	tb_lineno	número de línea actual en el código fuente de Python
	tb_next	el siguiente objeto de traceback interno (llamado por este nivel)
marco	f_back	el siguiente objeto exterior del marco (el que llama a este marco)
	f_builtins	construye el namespace visto por este marco
	f_code	objeto de código que se ejecuta en este marco
	f_globals	el namespace global visto por este marco
	f_lasti	índice del último intento de instrucción en código de bytes
	f_lineno	número de línea actual en el código fuente de Python
	f_locals	el namespace local visto por este marco
	f_trace	función de rastreo para este marco, o `None`
code	co_argcount	número de argumentos (sin incluir los argumentos de palabras clave, * o ** args)
	co_code	cadena de bytecode compilados en bruto
	co_cellvars	tupla de nombres de variables de celda (referenciados por contener alcances)
	co_consts	tupla de constantes utilizadas en el bytecode
	co_filename	nombre del archivo en el que este objeto código fue creado
	co_firstlineno	número de la primera línea del código fuente de Python
	co_flags	mapa de bits de los flags `CO_*`, leer más *aquí*
	co_lnotab	mapeo codificado de los números de línea a los índices de bytecode
	co_freevars	tupla de nombres de variables libres (referenciados a través del cierre de una función)
	co_posonlyargcount	número de argumentos solo posicionales
	co_kwonlyargcount	número de argumentos de sólo palabras clave (sin incluir el ** arg)
	co_name	nombre con el que se definió este objeto de código
	co_names	tuple of names other than arguments and function locals

con

Tabla 1 – proviene de la página anterior

Tipo	Atributo	Descripción
	co_nlocals	número de variables locales
	co_stacksize	se requiere espacio en la pila de máquina virtual
	co_varnames	tupla de nombres de argumentos y variables locales
generador	__name__	nombre
	__qualname__	nombre calificado
	gi_frame	marco
	gi_running	¿Está el generador en ejecución?
	gi_code	code
	gi_yieldfrom	el objeto siendo iterado por `yield from`, o `None`
corutina	__name__	nombre
	__qualname__	nombre calificado
	cr_await	objeto al que se espera, o `None`
	cr_frame	marco
	cr_running	¿Está la corutina en ejecución?
	cr_code	code
	cr_origin	donde se creó la corutina, o `None`. Ver *sys.set_coroutine_origin_tracking_d*
incorporado	__doc__	cadena de caracteres de documentación
	__name__	nombre original de esta función o método
	__qualname__	nombre calificado
	__self__	instancia a la que está ligada un método, o `None`

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.

Distinto en la versión 3.10: Agrega el atributo `__builtins__` a funciones.

`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 metaclase cuando el argumento sea una clase y esos atributos hayan sido listados en la costumbre de la metaclase `__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.

Detalles de implementación de CPython: 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.

Detalles de implementación de CPython: 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*)
Get the documentation string for an object, cleaned up with *cleandoc()*. If the documentation string for an object is not provided and the object is a class, a method, a property or a descriptor, retrieve the documentation string from the inheritance hierarchy. Return None if the documentation string is invalid or missing.

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*)
Try to guess which module an object was defined in. Return None if the module cannot be determined.

inspect.**getsourcefile**(*object*)
Return the name of the Python source file in which an object was defined or None if no way can be identified to get the source. This will fail with a *TypeError* if the object is a built-in module, class, or function.

inspect.**getsourcelines**(*object*)
Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An *OSError* is raised if the source code cannot be retrieved. A *TypeError* is raised if the object is a built-in module, class, or function.

Distinto en la versión 3.3: *OSError* se lanza en lugar de *IOError*, ahora un alias del primero.

inspect.**getsource**(*object*)
Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An *OSError* is raised if the source code cannot be retrieved. A *TypeError* is raised if the object is a built-in module, class, or function.

Distinto en la versión 3.3: *OSError* se lanza 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*, *globals=None*, *locals=None*, *eval_str=False*)
Retorna un objeto *Signature* para el callable dado:

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> 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()*.

Para los objetos definidos en módulos que utilizan anotaciones convertidas a cadenas (from __future__ import annotations), *signature()* intentará invertir la conversión a cadenas de las anotaciones automáticamente usando *inspect.get_annotations()*. Los parámetros global, locals, y eval_str se pasan a *inspect.get_annotations()* al resolver las anotaciones; consulte la documentación de *inspect.get_annotations()* para obtener instrucciones sobre cómo utilizar estos parámetros.

Lanza *ValueError* si no se puede proporcionar ninguna signatura, y *TypeError* si ese tipo de objeto no es compatible. Además, si las anotaciones están convertidas a cadenas y `eval_str` no es falso, las llamadas `eval()` para invertir la conversión a cadenas de las anotaciones podrían generar cualquier tipo de excepción.

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.)

Nuevo en la versión 3.10: Parámetros `globals`, `locals`, y `eval_str`

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: Signature objects are picklable and *hashable*.

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 sobreescribir 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*, *globalns=None*, *localns=None*)
Retorna un objeto *Signature* (o su subclase) para un determinado `obj` invocable. Pasa `follow_wrapped=False` para obtener una signatura de `obj` sin desenvolver su cadena `__wrapped__`. `globalns` y `localns` serán usados como los espacios de nombre cuando se resuelvan las anotaciones.

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.

Nuevo en la versión 3.10: Parámetros `globalns` y `localns`.

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: Parameter objects are picklable and *hashable*.

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.

Detalles de implementación de CPython: 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
Describes how argument values are bound to the parameter. The possible values are accessible via *Parameter* (like `Parameter.KEYWORD_ONLY`), and support comparison and ordering, in the following order:

Nombre	Significado
POSITIONAL_ONLY	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.
POSITIONAL_OR_KEYWORD	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)
VAR_POSITIONAL	Una tupla de argumentos posicionales que no están ligados a ningún otro parámetro. Esto corresponde a un parámetro `*args` en una definición de función Python.
KEYWORD_ONLY	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 `*` o `*args` en una definición de función Python.
VAR_KEYWORD	Un dictado de argumentos de palabras clave que no están ligadas a ningún otro parámetro. Esto corresponde a un parámetro `**kwargs` 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'

>>> str(param.replace()) # Will create a shallow copy of 'param'
```

(continué en la próxima página)

(proviene de la página anterior)

```
'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 mutable de los nombres de los parámetros a los valores de los argumentos. Contiene solo argumentos vinculados explícitamente. 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.

Distinto en la versión 3.9: *arguments* ahora es de tipo *dict*. Anteriormente, era de tipo *collections.OrderedDict*.

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
{'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*, */*, **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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 desenvolvimiento 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 desenvolvimiento 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.

inspect.**get_annotations**(*obj*, *, *globals=None*, *locals=None*, *eval_str=False*)

Calcular el diccionario de anotaciones de un objeto.

`obj` puede ser un invocable, clase o módulo. Al pasar un objeto de cualquier otro tipo, se genera *TypeError*.

Retorna un diccionario. `get_annotations()` retorna un nuevo diccionario cada vez que se invoca; llamarlo dos veces en el mismo objeto retornara dos diccionarios diferentes pero equivalentes.

Esta función maneja varios detalles por usted:

- Si `eval_str` es verdadero, los valores del tipo `str` dejarán de ser cadenas usando *eval()*. La intención de este diseño es su uso con anotaciones convertidas a cadenas (`from __future__ import annotations`).
- Si `obj` no tiene un diccionario de anotaciones, retorna un diccionario vacío. (Las funciones y los métodos siempre tienen un diccionario de anotaciones; las clases, los módulos y otros tipos de intocables pueden no tenerlo).
- Ignora las anotaciones heredadas en las clases. Si una clase no tiene su propio diccionario de anotaciones, retorna un diccionario vacío.
- Todos los accesos a los miembros del objeto y a los valores del diccionario se realizan mediante `getattr()` y `dict.get()` para mayor seguridad.
- Always, always, always returns a freshly created dict.

`eval_str` controla si los valores del tipo `str` se sustituyen o no por el resultado de llamar a *eval()* sobre esos valores:

- Si eval_str es verdadero, *eval()* se llama a valores de tipo `str`. (Tenga en cuenta que `get_annotations` no detecta excepciones; si *eval()* genera una excepción, desenrollará la pila más allá de la llamada `get_annotations`).

- Si eval_str es false (el valor predeterminado), los valores del tipo `str` no cambian.

`globals` and `locals` se pasan a *`eval()`*; consulte la documentación de *`eval()`* para obtener más información. Si `globals` o ``locals` es `None`, esta función puede reemplazar ese valor con un valor predeterminado específico del contexto, supeditado a `type(obj)`:

- Si `obj` es un módulo, `globals` por defecto es `obj.__dict__`.
- Si `obj` es una clase, `globals` tiene como valor predeterminado `sys.modules[obj.__module__].__dict__` y `locals` es de forma predeterminada el espacio de nombres de clase `obj`.
- Si `obj` es un callable, `globals` por defecto es `obj.__globals__`, aunque si `obj` es una función envuelta (usando `functools.update_wrapper()`) primero se desenvuelve.

Llamar a `get_annotations` es una práctica recomendada para acceder al diccionario de anotaciones de cualquier objeto. Consulte annotations-howto para obtener más información sobre las prácticas recomendadas de anotaciones.

Nuevo en la versión 3.10.

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 tupla 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 tupla con nombre en lugar de una tupla.

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.

Detalles de implementación de CPython: 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.

Detalles de implementación de CPython: 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 `sitio.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 encabezado y otra de cola. Para la parte del encabezado, usa `sys.prefix` y `sys.exec_prefix`; se saltean los encabezados vacíos. 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 macOS). Para cada una de las distintas combinaciones de encabezado 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 «pyvenv.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 «pyvenv.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 el Python en ejecución. Puede ser `None` si aún no se ha llamado a *getusersitepackages()*. El valor predeterminado es `~/.local/lib/python`*X.Y*`/site-packages` para compilaciones de macOS sin marco y UNIX, `~/Library/Python/`*X.Y*`/lib/python/site-packages` para compilaciones de framework de macOS, y *%APPDATA%*`\Python\Python`*XY*`\site-packages` en Windows. Este directorio es un directorio de sitio, lo que significa que se procesarán los archivos `.pth` 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 macOS, `~/Library/Python/`*X.Y* para las compilaciones de framework de macOS 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**()
Retorna la ruta del directorio base del usuario *USER_SITE*. Si este aún no fue inicializado, esta función también lo configurará, respetando *USER_BASE*. Para determinar si los paquetes específicos del sitio de usuario fueron añadidos a `sys.path` debe usarse *ENABLE_USER_SITE*

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

CAPÍTULO 30

Intérpretes de Python personalizados

The modules described in this chapter allow writing interfaces similar to Python's interactive interpreter. If you want a Python interpreter that supports some special feature in addition to the Python language, you should look at the `code` module. (The `codeop` module is lower-level, used to support compiling a possibly incomplete chunk of Python code.)

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 es la cadena de caracteres fuente; *filename* es el nombre de archivo opcional desde el que se leyó la fuente, por defecto es `'<input>'`; y *symbol* es el símbolo de inicio gramatical opcional, que debería ser `'single'` (el predeterminado), `'eval'` o `'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 `OFError`. La implementación base lee de `sys.stdin`; una subclase puede reemplazar esto con una implementación diferente.

30.2 `codeop` — Compila código Python

Código fuente: Lib/codeop.py

El módulo `codeop` proporciona herramientas para emular el bucle principal del intérprete de Python (también conocido como read-eval-print), tal como se hace en el módulo `code`. Por lo tanto, este módulo no está diseñado para utilizarlo directamente; si desea incluir un ciclo de este tipo en su programa, probablemente es mejor utilizar el módulo `code` en su lugar.

Esta actividad consta de dos partes:

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.

El módulo `codeop` proporciona formas de realizar estas dos partes, tanto de forma independiente, como en conjunto.

Para hacer lo anterior:

`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.

Si hay un problema con *source*, se lanzará una excepción. *SyntaxError* se lanza si hay una sintaxis de Python no válida, y *OverflowError* o *ValueError* si hay un literal no válido.

The *symbol* argument determines whether *source* is compiled as a statement (`'single'`, the default), as a sequence of *statement* (`'exec'`) or as an *expression* (`'eval'`). Any other value will cause *ValueError* to be raised.

Nota: Es posible (pero no probable) que el analizador deje de analizar con un resultado exitoso antes de llegar al final de la fuente; en este caso, los símbolos finales pueden ignorarse en lugar de provocar un error. Por ejemplo, una barra invertida seguida de dos nuevas líneas puede ir seguida de basura arbitraria. Esto se solucionará una vez que la API para el analizador sea mejor.

---

class codeop.**Compile**

Las instancias de esta clase tienen `__call__()` métodos idénticos en firma a la función incorporada *compile()*, pero con la diferencia de que si la instancia compila el texto del programa que contiene una instrucción *__future__*, la instancia "recuerda" y compila todos los textos de programa posteriores con la declaración en vigor.

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.

CAPÍTULO 31

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.

Any files may be present in the ZIP archive, but importers are only invoked for `.py` and `.pyc` files. ZIP import of dynamic modules (`.pyd`, `.so`) is disallowed. Note that if an archive only contains `.py` files, Python will not attempt to modify the archive by adding the corresponding `.pyc` file, meaning that if a ZIP archive doesn't contain `.pyc` files, importing may be rather slow.

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.

***`importlib`* - La implementación de la maquinaria de importación** El paquete proporciona los protocolos relevantes para que los implementen todos los importadores.

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.

create_module(*spec*)
Implementación de *importlib.abc.Loader.create_module()* que retorna *None* para solicitar explícitamente la semántica predeterminada.

Nuevo en la versión 3.10.

exec_module(*module*)
Implementación de *importlib.abc.Loader.exec_module()*.

Nuevo en la versión 3.10.

find_loader(*fullname*, *path=None*)
Una implementación de *importlib.abc.PathEntryFinder.find_loader()*.

Obsoleto desde la versión 3.10: Utilizar en su lugar *find_spec()*.

find_module(*fullname*, *path=None*)
Busca un módulo especificado por *fullname*. *fullname* debe ser el nombre completo del módulo (punteado). Retorna la propia instancia zipimporter si el módulo fue encontrado, o *None* si no. El argumento opcional *path* es ignorado — está ahí por compatibilidad con el protocolo del importador.

Obsoleto desde la versión 3.10: Utilizar en su lugar *find_spec()*.

find_spec(*fullname*, *target=None*)
Una implementación de *importlib.abc.PathEntryFinder.find_spec()*.

Nuevo en la versión 3.10.

get_code(*fullname*)
Retorna el objeto de código para el módulo especificado. Lanza *ZipImportError* si el módulo no se pudo importar.

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 se pudo importar.

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 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*)
Carga 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.

Obsoleto desde la versión 3.10: Utilizar en su lugar `exec_module()`.

invalidate_caches()
Limpia la caché interna de información sobre los archivos que se encuentran dentro del archivo ZIP.

Nuevo en la versión 3.10.

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__)
```

This will add to the package's __path__ all subdirectories of directories on *sys.path* named after the package. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

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 que envuelve el algoritmo de importación «clásico» de Python.

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.

If fullname contains a `'.'`, the finders will be for the package containing fullname, otherwise they will be all registered top level finders (i.e. those on both *sys.meta_path* and *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=''*)
Yields *ModuleInfo* for all submodules on *path*, or, if *path* is `None`, all top-level modules on *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`.

pkgutil.**resolve_name**(*name*)
Resuelve un nombre a un objeto.

Esta funcionalidad es usada en numerosos lugares en la librería estándar (ver bpo-12915) - y existe funcionalidad equivalente también en librerías de terceros ampliamente usadas, tales como setuptools, Django y Pyramid.

Se espera que *name* sea una cadena de caracteres en uno de los siguientes formatos, donde W representa un identificador de Python válido, y un punto representa literalmente un punto en las siguientes pseudo-expresiones regulares:

- `W(.W)*`
- `W(.W)*:(W(.W)*)?`

La primera forma existe sólo para mantener compatibilidad con versiones anteriores. Asume que parte del nombre con puntos es un paquete, y el resto es un objeto en algún lugar dentro de ese paquete, posiblemente anidado dentro de otros objetos. Dado que el lugar donde el paquete termina y la jerarquía de objetos comienza no puede ser inferida por inspección, con esta forma se debe realizar repetidos intentos de importación.

En la segunda forma, el usuario hace claro el punto de división al proveer un signo de dos puntos: el nombre con puntos a la izquierda de los dos puntos es el paquete a ser importado, y el nombre con puntos a la derecha es la jerarquía de nombres dentro de ese paquete. Sólo una importación se necesaria con esta forma. Si termina con dos puntos, entonces se retorna un objeto módulo.

La función retornará un objeto (el cual puede ser un módulo), o lanzará una de las siguientes excepciones:

`ValueError` – si *name* no tiene un formato reconocido.

`ImportError` – si una importación falló cuando no lo debería haber hecho.

`AttributeError` – Si un fallo ocurrió mientras se atravesaba la jerarquía de objetos dentro del paquete importado para obtener el objeto deseado.

Nuevo en la versión 3.9.

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 modulo *runpy* es usado para localizar y correr módulos *Python* sin importarlo primero. Su uso principal es implementar la opción `-m` cambiando la linea 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 modulo *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 **PEP 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 modulo *realmente* importado (es decir, __spec__.name siempre será un *mod_name* o mod_name + '.__main__, jamas *run_name*).

__file__, __cached__, __loader__ y __package__ son basados en la especificación del modulo 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 esta ejecutado. 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 subprocesos. Otros subprocesos 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 linea de comandos.

Distinto en la versión 3.1: Se agrego la capacidad de ejecutar paquetes buscando un submódulo __main__.

Distinto en la versión 3.2: Se agrego 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**(*path_name*, *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* valida (comúnmente es un archivo *zip* o un directorio), la entrada se agrega primero al comienzo de sys.path. La función busca y ejecuta un modulo *__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* precompilado), entonces __file__ se establecerá en la ruta proporcionada, y __spec__, __cached__, __loader__ y __package__ se establecerán todos en *None*.

Si la ruta proporciona es una referencia a una entrada *sys.path* valida, entonces `__spec__` se establece apropiadamente para la importación del modulo `__main__``(es decir, ``__spec__.name` siempre deberá ser `__main__`). `__file__`, `__cached__`, `__loader__` y `__package__` estarán basadas en la especificación del modulo establecidas como normal.

A number of alterations are also made to the *sys* module. Firstly, `sys.path` may be altered as described above. `sys.argv[0]` is updated with the value of `path_name` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. All modifications to items in *sys* are reverted before the function returns.

Tenga en cuenta que, diferente a *run_module()*, las alteraciones hecha 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 linea 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* valida 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 linea de comandos *CPython*

La función *importlib.import_module()*

31.5 `importlib` — La implementación de `import`

Nuevo en la versión 3.1.

Código fuente: Lib/importlib/__init__.py

31.5.1 Introducción

El propósito del paquete *importlib* es doble. Uno es proveer la implementación de la declaración de `import`, (y así, por extensión, el método *__import__()*) en el código fuente de Python. Esto provee una implementación de la `import` la cual es compatible con cualquier interprete de Python. También provee una implementación que es más fácil de comprender que otras implementaciones, en lenguajes diferentes a Python.

Dos, los componentes incluidos para implementar `import` están expuestos en este paquete para que sea más fácil para los usuarios crear sus propios objetos (conocidos de forma genérica como *importer*) para participar en el proceso de importación.

Ver también:

import La referencia en el lenguaje para la declaración de `import`.

Especificaciones de paquetes Especificaciones originales de los paquetes. Algunas semánticas han cambiado desde que este documento fue escrito (ejemplo, redirección de acuerdo a `None` en *sys.modules*).

La función *__import__()* La declaración de `import` es una decoración sintáctica para esta función.

PEP 235 Importar en sistemas que no distinguen entre mayúsculas y minúsculas

PEP 263 Definiendo las codificaciones del código fuente de Python

PEP 302 Nuevos ganchos de importación

pep:*328* Importaciones: Multilíneas, y absolutos/relativos

PEP 366 Importaciones relativas, explicitas, del módulo principal

PEP 420 Paquetes implícitos en el espacio de nombres

PEP 451 Un tipo de ModuleSpec para el sistema de importación

PEP 488 Eliminación de archivos PYO

PEP 489 Inicialización de extensión de módulo en múltiples fases

PEP 552 Pycs determinísticos

PEP 3120 Usando UTF-8 como la codificación fuente por defecto

PEP 3147 Repositorio de directorios PYC

31.5.2 Funciones

`importlib.`**`__import__`**(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)
Una implementación de la función *__import__()* incorporada.

Nota: La importación programática los módulos debe usar *import_module()* en lugar de esta función.

`importlib.`**`import_module`**(*name*, *package=None*)
Importar un módulo. El argumento llamado *name* especifica qué módulo importar en términos absolutos o relativos (ejemplo, puede ser `pkg.mod` o `..mod`). Si el nombre fuera especificado en términos relativos, entonces el argumento llamado *package* debe ser igual al nombre del paquete que será el ancla para resolver el nombre del paquete (ejemplo `import_module('..mod', 'pkg.subpkg')` importará `pkg.mod`).

La función *import_module()* actúa como un envoltorio simplificador alrededor de *importlib.__import__()*. Esto quiere decir que las semánticas de la función son derivadas de *importlib.__import__()*. La diferencia más importante entre las dos funciones es que *import_module()* retorna el paquete especificado o el módulo (ejemplo `pkg.mod`), mientras que *__import__()* retorna el paquete o módulo del nivel superior (ejemplo `pkg`).

Si está importando dinámicamente un módulo que se creó desde que el intérprete comenzó la ejecución (por ejemplo, creó un archivo fuente de Python), es posible que deba llamar a *invalidate_caches()* para que el nuevo módulo sea detectado por el sistema de importación.

Distinto en la versión 3.3: Paquetes padres son importados automáticamente.

`importlib.`**`find_loader`**(*name*, *path=None*)
Encuentra el cargador de un módulo, opcionalmente con el especificado en *path*.Si el módulo esta en *sys.modules*, entonces retorna el *sys.modules[name].__loader__*‘ (a menos que el cargador sea `None` o no haya uno especificado, en tal caso se lanza un *ValueError*).Si no se encuentra ahí, se hace una búsqueda usando *sys.meta_path*. Se retorna `None` si no se encuentra un cargador.

Un nombre con puntos no tiene sus ascendientes importados implícitamente, ya que eso requeriría cargarlo y eso podría no ser deseado. Para importar un sub-módulo correctamente debes importar todos los paquetes ascendientes del sub-módulo y pase el argumento correcto a *path*.

Nuevo en la versión 3.3.

Distinto en la versión 3.4: Si el `__loader__` no está configurado, lanza un *ValueError*, igual a si el atributo fuera `None`.

Obsoleto desde la versión 3.4: Utilice *importlib.util.find_spec()* en su lugar.

importlib.**invalidate_caches**()
Invalide los cache internos de ubicadores encontrados en *sys.meta_path*.Si un buscador implementa `invalidate_caches()` entonces será llamado para realizar la invalidación.Esta función debe ser llamada si cualquier módulo ha sido creado/instalado mientras tu programa esta siendo ejecutado para garantizar que todos los buscadores noten la existencia del nuevo módulo.

Nuevo en la versión 3.3.

importlib.**reload**(*module*)
Recarga un *modulo* previamente importado. El argumento debe ser un objeto módulo, por lo que debe haber sido importado exitosamente. Esto es útil cuando has editado el código fuente de un archivo usando un editor externo y deseas probar la nueva versión sin abandonar el interprete de Python. El valor retornado es el objeto módulo (que puede ser diferente si la reimportación crea un nuevo objeto en *sys.modules*).

Cuando *reload()* es ejecutada:

- El código de un módulo de Python es recompilado y el código del módulo reejecutado, definiendo un nuevo conjunto de objetos que son asignados a los nombres de los módulos en el diccionario, reusando el *loader* que originalmente carga los módulos. El método `init` de los módulos de extension no es llamado de nuevo.
- Al igual que con todos los demás objetos en Python, los objetos antiguos solo se recuperan después de que sus recuentos de referencias caen a cero.
- Los nombres en el espacio de nombres del módulo se actualizan para señalar cualquier objeto nuevo o modificado.
- Otras referencias a los objetos antiguos (como los nombres externos al módulo) no se vuelven a vincular para hacer referencia a los nuevos objetos y deben actualizarse en cada espacio de nombres donde se produzcan 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 anularán las antiguas definiciones, por lo que generalmente esto no es un problema. Si la nueva versión de un módulo no define un nombre que fue definido por la versión anterior, la definición anterior permanece. Esta característica se puede utilizar en beneficio del módulo si mantiene una tabla global o caché de objetos — con una declaración `try` puede probar la presencia de la tabla y omitir su inicialización si lo desea:

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

Por lo general, no es muy útil recargar módulos integrados o cargados dinámicamente. No se recomienda recargar *sys*, *__main__*, *builtins* y otros módulos clave. En muchos casos, los módulos de extensión no están diseñados para inicializarse más de una vez y pueden fallar de manera arbitraria cuando se vuelven a cargar.

Si un módulo importa objetos de otro módulo usando `from ... import ...`, al llamar a *reload()* para el otro módulo no redefine los objetos importados de él — una forma de evitar esto es volver a ejecutar la instrucción `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 las definiciones de método de las instancias — continúan usando la definición de clase anterior. Lo mismo ocurre con las clases derivadas.

Nuevo en la versión 3.4.

Distinto en la versión 3.7: *ModuleNotFoundError* se lanza cuando el módulo que se está recargando carece de *ModuleSpec*.

31.5.3 `importlib.abc` – Clases base abstractas relacionadas con la importación

Código fuente: Lib/importlib/abc.py

El módulo *`importlib.abc`* contiene todas las clases base abstractas principales utilizadas por `import`. También se proporcionan algunas subclases de las clases base abstractas centrales para ayudar a implementar los ABC centrales.

Jerarquía ABC:

```
object
 +-- Finder (deprecated)
 +-- MetaPathFinder
 +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

class importlib.abc.**Finder**
Una clase base abstracta que representa *finder*.

Obsoleto desde la versión 3.3: Utilice *`MetaPathFinder`* o *`PathEntryFinder`* en su lugar.

abstractmethod find_module(*fullname*, *path=None*)
Un método abstracto para encontrar un *loader* para el módulo especificado. Originalmente especificado en **PEP 302**, este método estaba destinado a ser utilizado en *`sys.meta_path`* y en el subsistema de importación basado en rutas.

Distinto en la versión 3.4: Retorna `None` cuando se llama en lugar de generar *`NotImplementedError`*.

Obsoleto desde la versión 3.10: Implemente *`MetaPathFinder.find_spec()`* o *`PathEntryFinder.find_spec()`* en su lugar.

class importlib.abc.**MetaPathFinder**
An abstract base class representing a *meta path finder*.

Nuevo en la versión 3.3.

Distinto en la versión 3.10: Ya no hereda de *`Finder`*.

find_spec(*fullname*, *path*, *target=None*)
Un método abstracto para encontrar un *spec* para el módulo especificado. Si se trata de una importación de nivel superior, el *path* será `None`. De lo contrario, esta es una búsqueda de un subpaquete o módulo y *path* será el valor de `__path__` del paquete principal. Si no se puede encontrar una especificación, se retorna `None`. Cuando se pasa, `target` es un objeto de módulo que el buscador puede usar para hacer una suposición más informada sobre qué especificación retornar. *`importlib.util.spec_from_loader()`* puede ser útil para implementar `MetaPathFinders` concretos.

Nuevo en la versión 3.4.

find_module(*fullname*, *path*)
Un método heredado para encontrar un *loader* para el módulo especificado. Si se trata de una importación de nivel superior, el *path* será `None`. De lo contrario, esta es una búsqueda de un subpaquete o módulo y *path* será el valor de `__path__` del paquete principal. Si no se puede encontrar un cargador, se retorna `None`.

Si se define *`find_spec()`*, se proporciona una funcionalidad compatible con versiones anteriores.

Distinto en la versión 3.4: Retorna `None` cuando se llama en lugar de generar *`NotImplementedError`*. Puede usar *`find_spec()`* para proporcionar funcionalidad.

Obsoleto desde la versión 3.4: Use *find_spec()* en su lugar.

invalidate_caches()
Un método opcional que, cuando se llama, debería invalidar cualquier caché interno utilizado por el buscador. Utilizado por *importlib.invalidate_caches()* al invalidar los cachés de todos los buscadores en *sys.meta_path*.

Distinto en la versión 3.4: Retorna `None` cuando se llama en lugar de `NotImplemented`.

class importlib.abc.**PathEntryFinder**
Una clase base abstracta que representa un *buscador de entradas de ruta*. Aunque tiene algunas similitudes con *MetaPathFinder*, `PathEntryFinder` está diseñado para usarse solo dentro del subsistema de importación basado en rutas proporcionado por *importlib.machinery.PathFinder*.

Nuevo en la versión 3.3.

Distinto en la versión 3.10: Ya no hereda de *Finder*.

find_spec(*fullname*, *target=None*)
Un método abstracto para encontrar un *spec* para el módulo especificado. El buscador buscará el módulo solo dentro del *path entry* a la que está asignado. Si no se puede encontrar una especificación, se retorna `None`. Cuando se pasa, `target` es un objeto de módulo que el buscador puede usar para hacer una suposición más informada sobre qué especificación retornar. *importlib.util.spec_from_loader()* puede ser útil para implementar `PathEntryFinders` concretos.

Nuevo en la versión 3.4.

find_loader(*fullname*)
Un método heredado para encontrar un *loader* para el módulo especificado. Retorna una tupla de 2 de `(loader, portion)` donde `portion` es una secuencia de ubicaciones del sistema de archivos que contribuyen a parte de un paquete de espacio de nombres. El cargador puede ser `None` mientras se especifica `portion` para indicar la contribución de las ubicaciones del sistema de archivos a un paquete de espacio de nombres. Se puede usar una lista vacía para `portion` para indicar que el cargador no es parte de un paquete de espacio de nombres. Si `loader` es `None` y `portion` es la lista vacía, entonces no se encontró ningún cargador o ubicación para un paquete de espacio de nombres (es decir, no se pudo encontrar nada para el módulo).

Si se define *find_spec()*, se proporciona una funcionalidad compatible con versiones anteriores.

Distinto en la versión 3.4: Retorna `(None, [])` en lugar de lanzar *NotImplementedError*. Usa *find_spec()* cuando está disponible para proporcionar funcionalidad.

Obsoleto desde la versión 3.4: Use *find_spec()* en su lugar.

find_module(*fullname*)
Una implementación concreta de *Finder.find_module()* que es equivalente a `self.find_loader(fullname)[0]`.

Obsoleto desde la versión 3.4: Use *find_spec()* en su lugar.

invalidate_caches()
Un método opcional que, cuando se llama, debería invalidar cualquier caché interno utilizado por el buscador. Usado por *importlib.machinery.PathFinder.invalidate_caches()* al invalidar las cachés de todos los buscadores en caché.

class importlib.abc.**Loader**
Una clase base abstracta para un *loader*. Consulte **PEP 302** para obtener la definición exacta de cargador.

Los cargadores que deseen admitir la lectura de recursos deben implementar un método `get_resource_reader(fullname)` según lo especificado por *importlib.abc.ResourceReader*.

Distinto en la versión 3.7: Introdujo el método opcional `get_resource_reader()`.

create_module(*spec*)
Un método que retorna el objeto de módulo que se utilizará al importar un módulo. Este método puede

retornar `None`, lo que indica que se debe llevar a cabo la semántica de creación de módulos predeterminada.

Nuevo en la versión 3.4.

Distinto en la versión 3.5: A partir de Python 3.6, este método no será opcional cuando se defina *exec_module()*.

exec_module(*module*)
Un método abstracto que ejecuta el módulo en su propio espacio de nombres cuando se importa o se vuelve a cargar un módulo. El módulo ya debería estar inicializado cuando se llama a `exec_module()`. Cuando existe este método, se debe definir *create_module()*.

Nuevo en la versión 3.4.

Distinto en la versión 3.6: *create_module()* también debe definirse.

load_module(*fullname*)
Un método heredado para cargar un módulo. Si el módulo no se puede cargar, se lanza *ImportError*; de lo contrario, se retorna el módulo cargado.

Si el módulo solicitado ya existe en *sys.modules*, ese módulo debe usarse y recargarse. De lo contrario, el cargador debe crear un nuevo módulo e insertarlo en *sys.modules* antes de que comience la carga, para evitar la recursividad de la importación. Si el cargador insertó un módulo y la carga falla, el cargador debe eliminarlo de *sys.modules*; los módulos que ya están en *sys.modules* antes de que el cargador comenzara a ejecutarse deben dejarse en paz (ver *importlib.util.module_for_loader()*).

El cargador debe establecer varios atributos en el módulo. (Tenga en cuenta que algunos de estos atributos pueden cambiar cuando se recarga un módulo):

- **__name__** El nombre del módulo.
- **__file__** La ruta hacia donde se almacenan los datos del módulo (no configurada para módulos integrados).
- **__cached__** La ruta a la que se debe almacenar una versión compilada del módulo (no se establece cuando el atributo sería inapropiado).
- **__path__** Una lista de cadenas de caracteres que especifican la ruta de búsqueda dentro de un paquete. Este atributo no se establece en módulos.
- **__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__** El cargador utilizado para cargar el módulo. El decorador *importlib.util.module_for_loader()* puede manejar los detalles de `__package__`.

Cuando *exec_module()* está disponible, se proporciona una funcionalidad compatible con versiones anteriores.

Distinto en la versión 3.4: Lanza *ImportError* cuando se llama en lugar de *NotImplementedError*. Funcionalidad proporcionada cuando *exec_module()* está disponible.

Obsoleto desde la versión 3.4: La API recomendada para cargar un módulo es *exec_module()* (y *create_module()*). Los cargadores deberían implementarlo en lugar de load_module(). La maquinaria de importación se encarga de todas las demás responsabilidades de load_module() cuando se implementa exec_module().

module_repr(*module*)
Un método heredado que, cuando se implementa, calcula y retorna la repr del módulo dado, como una cadena. El repr() predeterminado del tipo de módulo utilizará el resultado de este método según corresponda.

Nuevo en la versión 3.3.

Distinto en la versión 3.4: Hecho opcional en vez de un método abstracto (*abstractmethod*)

Obsoleto desde la versión 3.4: La maquinaria de importación ahora se encarga de esto automáticamente.

class importlib.abc.**ResourceReader**

Reemplazada por TraversableResources

Un *abstract base class* para proporcionar la capacidad de leer *resources*.

Desde la perspectiva de este ABC, un *resource* es un artefacto binario que se envía dentro de un paquete. Por lo general, esto es algo así como un archivo de datos que se encuentra junto al archivo `__init__.py` del paquete. El propósito de esta clase es ayudar a abstraer el acceso a dichos archivos de datos de modo que no importe si el paquete y sus archivos de datos se almacenan en un, por ejemplo, zip en comparación con el sistema de archivos.

Para cualquiera de los métodos de esta clase, se espera que un argumento *resource* sea un *path-like object* que representa conceptualmente solo un nombre de archivo. Esto significa que no se deben incluir rutas de subdirectorio en el argumento *resource*. Esto se debe a que la ubicación del paquete para el que está ubicado el lector actúa como el «directorio». Por lo tanto, la metáfora de los directorios y los nombres de los archivos son los paquetes y los recursos, respectivamente. Esta es también la razón por la que se espera que las instancias de esta clase se correlacionen directamente con un paquete específico (en lugar de representar potencialmente varios paquetes o un módulo).

Se espera que los cargadores que deseen admitir la lectura de recursos proporcionen un método llamado `get_resource_reader(fullname)` que retorna un objeto que implementa la interfaz de este ABC. Si el módulo especificado por fullname no es un paquete, este método debería devolver `None`. Un objeto compatible con este ABC solo debe retornarse cuando el módulo especificado es un paquete.

Nuevo en la versión 3.7.

abstractmethod open_resource(*resource*)

Retorna un, *file-like object* abierto para la lectura binaria del *resource*.

Si no se puede encontrar el recurso, se lanza `FileNotFoundError`.

abstractmethod resource_path(*resource*)

Retorna la ruta del sistema de archivos al *resource*.

Si el recurso no existe concretamente en el sistema de archivos, lanza `FileNotFoundError`.

abstractmethod is_resource(*name*)

Retorna `True` si el *name* nombrado se considera un recurso. `FileNotFoundError` se lanza si *name* no existe.

abstractmethod contents()

Retorna un *iterable* de cadenas de caracteres sobre el contenido del paquete. Tenga en cuenta que no es necesario que todos los nombres retornados por el iterador sean recursos reales, por ejemplo, es aceptable retornar nombres para los que `is_resource()` sería falso.

Al permitir que se retornen nombres que no son de recursos es para permitir situaciones en las que se conoce a priori cómo se almacenan un paquete y sus recursos y los nombres que no son de recursos serían útiles. Por ejemplo, se permite el retorno de nombres de subdirectorios para que cuando se sepa que el paquete y los recursos están almacenados en el sistema de archivos, esos nombres de subdirectorios se puedan usar directamente.

El método abstracto retorna un iterable de ningún elemento.

class importlib.abc.**ResourceLoader**

Una clase base abstracta para un *loader* que implementa el protocolo opcional **PEP 302** para cargar recursos arbitrarios desde el back-end de almacenamiento.

Obsoleto desde la versión 3.7: Este ABC está en desuso a favor de admitir la carga de recursos a través de `importlib.abc.ResourceReader`.

abstractmethod get_data(*path*)

Un método abstracto para devolver los bytes de los datos ubicados en *path*. Los cargadores que tienen un back-end de almacenamiento similar a un archivo que permite almacenar datos arbitrarios pueden

implementar este método abstracto para dar acceso directo a los datos almacenados. *OSError* se lanza si no se puede encontrar el *path*. Se espera que la *path* se construya utilizando el atributo `__file__` de un módulo o un elemento de un paquete `__path__`.

Distinto en la versión 3.4: Lanza *OSError* en vez de *NotImplementedError*.

class importlib.abc.**InspectLoader**

Una clase base abstracta para un *loader* que implementa el protocolo opcional **PEP 302** para cargadores que inspeccionan módulos.

get_code(*fullname*)

Retorna el objeto código para un módulo, o `None` si el módulo no tiene un objeto código (como sería el caso, por ejemplo, para un módulo integrado). Lanza un *ImportError* si el cargador no puede encontrar el módulo solicitado.

Nota: Si bien el método tiene una implementación predeterminada, se sugiere que se anule si es posible para mejorar el rendimiento.

Distinto en la versión 3.4: Ya no es un método abstracto y se proporciona una implementación concreta.

abstractmethod get_source(*fullname*)

Un método abstracto para retornar la fuente de un módulo. Se retorna como una cadena de caracteres de texto usando *universal newlines*, traduciendo todos los separadores de línea reconocidos en caracteres `'\n'`. Retorna `None` si no hay una fuente disponible (por ejemplo, un módulo integrado). Lanza *ImportError* si el cargador no puede encontrar el módulo especificado.

Distinto en la versión 3.4: Lanza *ImportError* en vez de *NotImplementedError*.

is_package(*fullname*)

Un método opcional para retornar un valor verdadero si el módulo es un paquete, un valor falso en caso contrario. Se lanza *ImportError* si el *cargador* no puede encontrar el módulo.

Distinto en la versión 3.4: Lanza *ImportError* en vez de *NotImplementedError*.

static source_to_code(*data*, *path='<string>'*)

Cree un objeto de código a partir de la fuente de Python.

El argumento *data* puede ser cualquier cosa que admita la función *compile()* (es decir, cadena de caracteres o bytes). El argumento *path* debe ser la «ruta» de donde se originó el código fuente, que puede ser un concepto abstracto (por ejemplo, ubicación en un archivo zip).

Con el objeto de código subsiguiente, uno puede ejecutarlo en un módulo ejecutando `exec(code, module.__dict__)`.

Nuevo en la versión 3.4.

Distinto en la versión 3.5: Hace el método estático.

exec_module(*module*)

Implementación de *Loader.exec_module()*.

Nuevo en la versión 3.4.

load_module(*fullname*)

Implementación de *Loader.load_module()*.

Obsoleto desde la versión 3.4: use *exec_module()* en su lugar.

class importlib.abc.**ExecutionLoader**

Una clase base abstracta que hereda de *InspectLoader* que, cuando se implementa, ayuda a que un módulo se ejecute como un script. El ABC representa un protocolo opcional **PEP 302**.

abstractmethod get_filename(*fullname*)

Un método abstracto que retorna el valor de `__file__` para el módulo especificado. Si no hay una ruta disponible, se lanza *ImportError*.

Si el código fuente está disponible, entonces el método debe devolver la ruta al archivo fuente, independientemente de si se utilizó un código de bytes para cargar el módulo.

Distinto en la versión 3.4: Lanza *ImportError* en vez de *NotImplementedError*.

class importlib.abc.**FileLoader**(*fullname*, *path*)

Una clase base abstracta que hereda de *ResourceLoader* y *ExecutionLoader*, proporcionando implementaciones concretas de *ResourceLoader.get_data()* y *ExecutionLoader.get_filename()*.

El argumento *fullname* es un nombre completamente resuelto del módulo que el cargador debe manejar. El argumento *path* es la ruta al archivo del módulo.

Nuevo en la versión 3.3.

name

El nombre del módulo que puede manejar el cargador.

path

Ruta al archivo del módulo.

load_module(*fullname*)

Llama a super's `load_module()`.

Obsoleto desde la versión 3.4: Utilice *Loader.exec_module()* en su lugar.

abstractmethod get_filename(*fullname*)

Retorna *path*.

abstractmethod get_data(*path*)

Lee *path* como un archivo binario y devuelve los bytes de él.

class importlib.abc.**SourceLoader**

Una clase base abstracta para implementar la carga de archivos fuente (y opcionalmente bytecode). La clase hereda tanto de *ResourceLoader* como de *ExecutionLoader*, lo que requiere la implementación de:

- *ResourceLoader.get_data()*
- ***ExecutionLoader.get_filename()*** Solo debe devolver la ruta al archivo de origen; la carga sin fuente no es compatible.

Los métodos abstractos definidos por esta clase son para agregar soporte de archivo de código de bytes opcional. No implementar estos métodos opcionales (o hacer que se lance *NotImplementedError*) hace que el cargador solo funcione con el código fuente. La implementación de los métodos permite que el cargador trabaje con archivos fuente *y* código de bytes; no permite la carga *sin fuente* donde solo se proporciona un código de bytes. Los archivos de código de bytes son una optimización para acelerar la carga al eliminar el paso de análisis del compilador de Python, por lo que no se expone ninguna API específica de código de bytes.

path_stats(*path*)

Método abstracto opcional que devuelve un *dict* que contiene metadatos sobre la ruta especificada. Las claves de diccionario admitidas son:

- `'mtime'` (obligatorio): un número entero o de punto flotante que representa la hora de modificación del código fuente;
- `'size'` (opcional): el tamaño en bytes del código fuente.

Cualquier otra clave del diccionario se ignora para permitir futuras extensiones. Si no se puede manejar la ruta, se genera *OSError*.

Nuevo en la versión 3.3.

Distinto en la versión 3.4: Lanza *OSError* en vez de *NotImplementedError*.

path_mtime(*path*)

Método abstracto opcional que retorna la hora de modificación de la ruta especificada.

Obsoleto desde la versión 3.3: Este método está obsoleto en favor de `path_stats()`. No tiene que implementarlo, pero aún está disponible para fines de compatibilidad. Lanza `OSError` si la ruta no se puede manejar.

Distinto en la versión 3.4: Lanza `OSError` en vez de `NotImplementedError`.

set_data(*path*, *data*)
Método abstracto opcional que escribe los bytes especificados en una ruta de archivo. Los directorios intermedios que no existan se crearán automáticamente.

Cuando la escritura en la ruta falla porque la ruta es de solo lectura (`errno.EACCES`/`PermissionError`), no propague la excepción.

Distinto en la versión 3.4: Ya no lanza `NotImplementedError` cuando se llama.

get_code(*fullname*)
Implementación concreta de `InspectLoader.get_code()`.

exec_module(*module*)
Implementación concreta de `Loader.exec_module()`.

Nuevo en la versión 3.4.

load_module(*fullname*)
Implementación concreta de `Loader.load_module()`.

Obsoleto desde la versión 3.4: Utilice `exec_module()` en su lugar.

get_source(*fullname*)
Implementación concreta de `InspectLoader.get_source()`.

is_package(*fullname*)
Implementación concreta de `InspectLoader.is_package()`. Se determina que un módulo es un paquete si su ruta de archivo (proporcionada por `ExecutionLoader.get_filename()`) es un archivo llamado `__init__` cuando se elimina la extensión del archivo **y** el nombre del módulo sí lo hace no termina en `__init__`.

class importlib.abc.**Traversable**
An object with a subset of `pathlib.Path` methods suitable for traversing directories and opening files.

For a representation of the object on the file-system, use `importlib.resources.as_file()`.

Nuevo en la versión 3.9.

abstractmethod name()
The base name of this object without any parent references.

abstractmethod iterdir()
Yield Traversable objects in self.

abstractmethod is_dir()
Return True if self is a directory.

abstractmethod is_file()
Return True if self is a file.

abstractmethod joinpath(*child*)
Return Traversable child in self.

abstractmethod __truediv__(*child*)
Return Traversable child in self.

abstractmethod open(*mode='r'*, **args*, ***kwargs*)
mode may be "r" or "rb" to open as text or binary. Return a handle suitable for reading (same as `pathlib.Path.open`).

When opening as text, accepts encoding parameters such as those accepted by `io.TextIOWrapper`.

read_bytes()
Read contents of self as bytes.

read_text(*encoding=None*)
Read contents of self as text.

Note: In Python 3.11 and later, this class is found in `importlib.resources.abc`.

class `importlib.abc.`**TraversableResources**
An abstract base class for resource readers capable of serving the `files` interface. Subclasses ResourceReader and provides concrete implementations of the ResourceReader's abstract methods. Therefore, any loader supplying TraversableResources also supplies ResourceReader.

Se espera que los cargadores que deseen admitir la lectura de recursos implementen esta interfaz.

Nuevo en la versión 3.9.

Note: In Python 3.11 and later, this class is found in `importlib.resources.abc`.

31.5.4 `importlib.resources` – Recursos

Código fuente: Lib/importlib/resources.py

Nuevo en la versión 3.7.

Este módulo aprovecha el sistema de importación de Python para proporcionar acceso a *resources* dentro de *packages*. Si puede importar un paquete, puede acceder a los recursos dentro de ese paquete. Los recursos se pueden abrir o leer, ya sea en modo binario o texto.

Los recursos son similares a los archivos dentro de los directorios, aunque es importante tener en cuenta que esto es solo una metáfora. Los recursos y paquetes **no** tienen que existir como archivos y directorios físicos en el sistema de archivos.

Nota: Este módulo proporciona una funcionalidad similar a pkg_resources Acceso Básico a Recursos sin la sobrecarga de rendimiento de ese paquete. Esto facilita la lectura de los recursos incluidos en los paquetes, con una semántica más estable y coherente.

El backport independiente de este módulo proporciona más información sobre usar importlib.resources y migrar desde pkg_resources a importlib.resources.

Los cargadores que deseen admitir la lectura de recursos deben implementar un método `get_resource_reader(fullname)` según lo especificado por *`importlib.abc.ResourceReader`*.

Se definen los siguientes tipos.

`importlib.resources.`**Package**
El tipo `Package` se define como `Union[str, ModuleType]`. Esto significa que cuando la función describe la aceptación de un `Package`, puede pasar una cadena de caracteres o un módulo. Los objetos de módulo deben tener un `__spec__.submodule_search_locations` que se pueda resolver que no sea `None`.

`importlib.resources.`**Resource**
Este tipo describe los nombres de los recursos que se pasan a las distintas funciones de este paquete. Esto se define como `Union[str, os.PathLike]`.

Están disponibles las siguientes funciones.

`importlib.resources.`**files**(*package*)
Returns an *`importlib.abc.Traversable`* object representing the resource container for the package (think directory) and its resources (think files). A Traversable may contain other containers (think subdirectories).

package es un nombre o un objeto de módulo que cumple con los requisitos de `Package`.

Nuevo en la versión 3.9.

importlib.resources.**as_file**(*traversable*)
Given a *importlib.abc.Traversable* object representing a file, typically from *importlib.resources.files()*, return a context manager for use in a `with` statement. The context manager provides a *pathlib.Path* object.

Salir del gestor de contexto limpia cualquier archivo temporal creado cuando se extrajo el recurso de, por ejemplo, un archivo zip.

Use `as_file` cuando los métodos Traversable (``read_text`, etc) sean insuficientes y se requiera un archivo real en el sistema de archivos.

Nuevo en la versión 3.9.

importlib.resources.**open_binary**(*package*, *resource*)
Abra para lectura binaria el *resource* dentro del *package*.

package es un nombre o un objeto de módulo que cumple con los requisitos de `Package`. *resource* es el nombre del recurso a abrir dentro de *package*; puede que no contenga separadores de ruta y puede que no tenga sub-recursos (es decir, no puede ser un directorio). Esta función retorna una instancia de `typing.BinaryIO`, un flujo de E/S binario abierto para lectura.

importlib.resources.**open_text**(*package*, *resource*, *encoding='utf-8'*, *errors='strict'*)
Se abre para leer el texto del *resource* dentro del *package*. De forma predeterminada, el recurso está abierto para lectura como UTF-8.

package es un nombre o un objeto de módulo que cumple con los requisitos de `Package`. *resource* es el nombre del recurso a abrir dentro de *package*; puede que no contenga separadores de ruta y puede que no tenga sub-recursos (es decir, no puede ser un directorio). *encoding* y *errors* tienen el mismo significado que con la función integrada *open()*.

Esta función retorna una instancia de `typing.TextIO`, un flujo de E/S de texto abierto para lectura.

importlib.resources.**read_binary**(*package*, *resource*)
Lee y retorna el contenido del *resource* dentro del *package* como `bytes`.

package es un nombre o un objeto de módulo que cumple con los requisitos de `Package`. *resource* es el nombre del recurso a abrir dentro de *package*; puede que no contenga separadores de ruta y puede que no tenga sub-recursos (es decir, no puede ser un directorio). Esta función retorna el contenido del recurso como *bytes*.

importlib.resources.**read_text**(*package*, *resource*, *encoding='utf-8'*, *errors='strict'*)
Lee y retorna el contenido de *resource* dentro de *package* como un `str`. De forma predeterminada, los contenidos se leen como UTF-8 estricto.

package es un nombre o un objeto de módulo que cumple con los requisitos de `Package`. *resource* es el nombre del recurso a abrir dentro de *package*; puede que no contenga separadores de ruta y puede que no tenga sub-recursos (es decir, no puede ser un directorio). *encoding* y *errors* tienen el mismo significado que con la función integrada *open()*. Esta función retorna el contenido del recurso como *str*.

importlib.resources.**path**(*package*, *resource*)
Retorna la ruta al *resource* como una ruta real del sistema de archivos. Esta función retorna un administrador de contexto para usar en una declaración `with`. El administrador de contexto proporciona un objeto *pathlib.Path*.

Salir del administrador de contexto limpia cualquier archivo temporal creado cuando el recurso necesita ser extraído, por ejemplo, un archivo zip.

package es un nombre o un objeto de módulo que cumple con los requisitos de `Package`. *resource* es el nombre del recurso a abrir dentro de *package*; puede que no contenga separadores de ruta y puede que no tenga sub-recursos (es decir, no puede ser un directorio).

importlib.resources.**is_resource**(*package*, *name*)
Retorna `True` si hay un recurso llamado *name* en el paquete; de lo contrario, `False`. ¡Recuerde que los directorios *no* son recursos! *package* es un nombre o un objeto de módulo que cumple con los requisitos de `Package`.

importlib.resources.**contents**(*package*)
Retorna un iterable sobre los elementos nombrados dentro del paquete. El iterable retorna recursos *str* (por ejemplo, archivos) y no-recursos (por ejemplo, directorios). El iterable no recurre a subdirectorios.

package es un nombre o un objeto de módulo que cumple con los requisitos de Package.

31.5.5 importlib.machinery – Importadores y enlaces de ruta

Código fuente: Lib/importlib/machinery.py

Este módulo contiene varios objetos que ayudan import buscar y cargar módulos.

importlib.machinery.**SOURCE_SUFFIXES**
Una lista de cadenas de caracteres que representan los sufijos de archivo reconocidos para los módulos de origen.

Nuevo en la versión 3.3.

importlib.machinery.**DEBUG_BYTECODE_SUFFIXES**
Una lista de cadenas que representan los sufijos de archivo para módulos de código de bytes no optimizados.

Nuevo en la versión 3.3.

Obsoleto desde la versión 3.5: Utilice *BYTECODE_SUFFIXES* en su lugar.

importlib.machinery.**OPTIMIZED_BYTECODE_SUFFIXES**
Una lista de cadenas de caracteres que representan los sufijos de archivo para módulos de código de bytes optimizados.

Nuevo en la versión 3.3.

Obsoleto desde la versión 3.5: Utilice *BYTECODE_SUFFIXES* en su lugar.

importlib.machinery.**BYTECODE_SUFFIXES**
Una lista de cadenas de caracteres que representan los sufijos de archivo reconocidos para los módulos de código de bytes (incluido el punto inicial).

Nuevo en la versión 3.3.

Distinto en la versión 3.5: El valor ya no depende de __debug__.

importlib.machinery.**EXTENSION_SUFFIXES**
Una lista de cadenas de caracteres que representan los sufijos de archivo reconocidos para los módulos de extensión.

Nuevo en la versión 3.3.

importlib.machinery.**all_suffixes**()
Retorna una lista combinada de cadenas de caracteres que representan todos los sufijos de archivo para módulos reconocidos por la maquinaria de importación estándar. Este es un ayudante para el código que simplemente necesita saber si una ruta del sistema de archivos potencialmente se refiere a un módulo sin necesidad de detalles sobre el tipo de módulo (por ejemplo, *inspect.getmodulename()*).

Nuevo en la versión 3.3.

class importlib.machinery.**BuiltinImporter**
Un *importer* para módulos integrados. Todos los módulos integrados conocidos se enumeran en *sys.builtin_module_names*. Esta clase implementa los ABC *importlib.abc.MetaPathFinder* y *importlib.abc.InspectLoader*.

Esta clase solo define los métodos de clase para aliviar la necesidad de instanciación.

Distinto en la versión 3.5: Como parte de **PEP 489**, el importador integrado ahora implementa Loader.create_module() y Loader.exec_module()

class importlib.machinery.**FrozenImporter**
Un *importer* para módulos congelados. Esta clase implementa los ABC *importlib.abc.MetaPathFinder* y *importlib.abc.InspectLoader*.

Esta clase solo define los métodos de clase para aliviar la necesidad de instanciación.

Distinto en la versión 3.4: Métodos obtenidos `create_module()` y `exec_module()`.

class importlib.machinery.**WindowsRegistryFinder**
Finder para los módulos declarados en el registro de Windows. Esta clase implementa el *importlib.abc.MetaPathFinder* ABC.

Esta clase solo define los métodos de clase para aliviar la necesidad de instanciación.

Nuevo en la versión 3.3.

Obsoleto desde la versión 3.6: Utilice la configuración de *site* en su lugar. Es posible que las versiones futuras de Python no habiliten este buscador de forma predeterminada.

class importlib.machinery.**PathFinder**
Un *Finder* para *sys.path* y atributos del paquete `__path__`. Esta clase implementa el *importlib.abc.MetaPathFinder* ABC.

Esta clase solo define los métodos de clase para aliviar la necesidad de instanciación.

classmethod find_spec(*fullname*, *path=None*, *target=None*)
Método de clase que intenta encontrar un *spec* para el módulo especificado por *fullname* en *sys.path* o, si está definido, en *path*. Para cada entrada de ruta que se busca, se comprueba *sys.path_importer_cache*. Si se encuentra un objeto que no es falso, se utiliza como *path entry finder* para buscar el módulo que se está buscando. Si no se encuentra ninguna entrada en *sys.path_importer_cache*, entonces *sys.path_hooks* se busca un buscador para la entrada de ruta y, si se encuentra, se almacena en *sys.path_importer_cache* junto con ser consultado sobre el módulo. Si nunca se encuentra ningún buscador, entonces `None` se almacena en el caché y se retorna.

Nuevo en la versión 3.4.

Distinto en la versión 3.5: Si el directorio de trabajo actual, representado por una cadena de caracteres vacía, ya no es válido, se retorna"None" pero no se almacena ningún valor en *sys.path_importer_cache*.

classmethod find_module(*fullname*, *path=None*)
Una envoltura heredada alrededor de *find_spec()*.

Obsoleto desde la versión 3.4: Use *find_spec()* en su lugar.

classmethod invalidate_caches()
Llama *importlib.abc.PathEntryFinder.invalidate_caches()* en todos los buscadores almacenados en *sys.path_importer_cache* que definen el método. De lo contrario, las entradas en *sys.path_importer_cache* establecidas en `None` se eliminan.

Distinto en la versión 3.7: Se eliminan las entradas de `None` en *sys.path_importer_cache*.

Distinto en la versión 3.4: Llama a objetos en *sys.path_hooks* con el directorio de trabajo actual para `''` (es decir, la cadena de caracteres vacía).

class importlib.machinery.**FileFinder**(*path*, **loader_details*)
Una implementación concreta de *importlib.abc.PathEntryFinder* que almacena en caché los resultados del sistema de archivos.

El argumento *path* es el directorio que el buscador se encarga de buscar.

El argumento *loader_details* es un número variable de tuplas de 2 elementos, cada una de las cuales contiene un cargador y una secuencia de sufijos de archivo que el cargador reconoce. Se espera que los cargadores sean invocables que acepten dos argumentos del nombre del módulo y la ruta al archivo encontrado.

El buscador almacenará en caché el contenido del directorio según sea necesario, haciendo llamadas estadísticas para cada búsqueda de módulo para verificar que la caché no esté desactualizada. Debido a que la obsolescencia

de la caché se basa en la granularidad de la información de estado del sistema operativo del sistema de archivos, existe una condición de carrera potencial de buscar un módulo, crear un nuevo archivo y luego buscar el módulo que representa el nuevo archivo. Si las operaciones ocurren lo suficientemente rápido como para ajustarse a la granularidad de las llamadas estadísticas, la búsqueda del módulo fallará. Para evitar que esto suceda, cuando cree un módulo dinámicamente, asegúrese de llamar a `importlib.invalidate_caches()`.

Nuevo en la versión 3.3.

path
La ruta en la que buscará el buscador.

find_spec(*fullname*, *target=None*)
Intente encontrar la especificación para manejar *fullname* dentro de `path`.

Nuevo en la versión 3.4.

find_loader(*fullname*)
Intente encontrar el cargador para manejar *fullname* dentro de `path`.

Obsoleto desde la versión 3.10: Use `find_spec()` en su lugar.

invalidate_caches()
Borrar el caché interno.

classmethod path_hook(**loader_details*)
Un método de clase que devuelve un cierre para su uso en `sys.path_hooks`. Una instancia de `FileFinder` es retornada por el cierre usando el argumento de ruta dado al cierre directamente y *loader_details* indirectamente.

Si el argumento del cierre no es un directorio existente, se lanza `ImportError`.

class importlib.machinery.**SourceFileLoader**(*fullname*, *path*)
Una implementación concreta de `importlib.abc.SourceLoader` subclasificando `importlib.abc.FileLoader` y proporcionando algunas implementaciones concretas de otros métodos.

Nuevo en la versión 3.3.

name
El nombre del módulo que manejará este cargador.

path
La ruta al archivo de origen.

is_package(*fullname*)
Devuelve `True` si `path` parece ser para un paquete.

path_stats(*path*)
Implementación concreta de `importlib.abc.SourceLoader.path_stats()`.

set_data(*path*, *data*)
Implementación concreta de `importlib.abc.SourceLoader.set_data()`.

load_module(*name=None*)
Implementación concreta de `importlib.abc.Loader.load_module()` donde especificar el nombre del módulo a cargar es opcional.

Obsoleto desde la versión 3.6: Utilice `importlib.abc.Loader.exec_module()` en su lugar.

class importlib.machinery.**SourcelessFileLoader**(*fullname*, *path*)
Una implementación concreta de `importlib.abc.FileLoader` que puede importar archivos de código de bytes (es decir, no existen archivos de código fuente).

Tenga en cuenta que el uso directo de archivos de código de bytes (y, por lo tanto, no de archivos de código fuente) impide que sus módulos sean utilizables por todas las implementaciones de Python o las nuevas versiones de Python que cambian el formato de código de bytes.

Nuevo en la versión 3.3.

name
El nombre del módulo que manejará el cargador.

path
La ruta al archivo de código de bytes.

is_package(*fullname*)
Determina si el módulo es un paquete basado en *path*.

get_code(*fullname*)
Retorna el objeto de código para *name* creado a partir de *path*.

get_source(*fullname*)
Devuelve `None` ya que los archivos de código de bytes no tienen fuente cuando se usa este cargador.

load_module(*name=None*)

Implementación concreta de *importlib.abc.Loader.load_module()* donde especificar el nombre del módulo a cargar es opcional.

Obsoleto desde la versión 3.6: Utilice *importlib.abc.Loader.exec_module()* en su lugar.

class importlib.machinery.**ExtensionFileLoader**(*fullname*, *path*)
Una implementación concreta de *importlib.abc.ExecutionLoader* para módulos de extensión.

El argumento *fullname* especifica el nombre del módulo que el cargador debe admitir. El argumento *path* es la ruta al archivo del módulo de extensión.

Nuevo en la versión 3.3.

name
Nombre del módulo que admite el cargador.

path
Ruta al módulo de extensión.

create_module(*spec*)
Crea el objeto de módulo a partir de la especificación dada de acuerdo con **PEP 489**.

Nuevo en la versión 3.5.

exec_module(*module*)
Inicializa el objeto de módulo dado de acuerdo con **PEP 489**.

Nuevo en la versión 3.5.

is_package(*fullname*)
Retorna `True` si la ruta del archivo apunta al módulo `__init__` de un paquete basado en *EXTENSION_SUFFIXES*.

get_code(*fullname*)
Retorna `None` ya que los módulos de extensión carecen de un objeto de código.

get_source(*fullname*)
Retorna `None` ya que los módulos de extensión no tienen código fuente.

get_filename(*fullname*)
Retorna *path*.

Nuevo en la versión 3.4.

class importlib.machinery.**ModuleSpec**(*name*, *loader*, ***, *origin=None*, *loader_state=None*, *is_package=None*)
Una especificación para el estado relacionado con el sistema de importación de un módulo. Esto generalmente se expone como el atributo `__spec__` del módulo. En las descripciones siguientes, los nombres entre paréntesis dan el atributo correspondiente disponible directamente en el objeto del módulo, por ejemplo, `module.__spec__.origin == module.__file__`. Sin embargo, tenga en cuenta que, si bien los *values* suelen ser equivalentes, pueden diferir ya que no hay sincronización entre los dos objetos. Por lo tanto,

es posible actualizar el `__path__` del módulo en tiempo de ejecución, y esto no se reflejará automáticamente en `__spec__.submodule_search_locations`.

Nuevo en la versión 3.4.

name

(`__name__`)

A string for the fully qualified name of the module.

loader

(`__loader__`)

El *Loader* que debe usarse al cargar el módulo. *Finders* siempre debe establecer esto.

origin

(`__file__`)

Nombre del lugar desde el que se carga el módulo, por ejemplo «incorporado» (*builtin*) para los módulos incorporados y el nombre de archivo para los módulos cargados desde la fuente. Normalmente se debe establecer «origen», pero puede ser `None` (el valor predeterminado), lo que indica que no está especificado (por ejemplo, para paquetes de espacio de nombres).

submodule_search_locations

(`__path__`)

Lista de cadenas de caracteres de dónde encontrar submódulos, si es un paquete (`None` de lo contrario).

loader_state

Contenedor de datos adicionales específicos del módulo para usar durante la carga (o `None`).

cached

(`__cached__`)

Cadena de caracteres para el lugar donde se debe almacenar el módulo compilado (o `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

Booleano que indica si el atributo «origen» del módulo se refiere a una ubicación cargable.

31.5.6 `importlib.util` – Código de utilidad para importadores

Código fuente: Lib/importlib/util.py

Este módulo contiene los diversos objetos que ayudan en la construcción de un *importer*.

`importlib.util.`**MAGIC_NUMBER**
Los bytes que representan el número de versión del código de bytes. Si necesita ayuda para cargar/escribir código de bytes, considere *`importlib.abc.SourceLoader`*.

Nuevo en la versión 3.4.

`importlib.util.`**cache_from_source**(*path*, *debug_override=None*, ***, *optimization=None*)
Retorna la ruta **PEP 3147**/**PEP 488** al archivo compilado por bytes asociado con la *path* de origen. Por ejemplo, si *path* es `/foo/bar/baz.py`, el valor de retorno sería `/foo/bar/__pycache__/baz.cpython-32.pyc` para Python 3.2. La cadena de caracteres `cpython-32` proviene de la etiqueta

mágica actual (ver `get_tag()`; si `sys.implementation.cache_tag` no está definido, se lanzará *NotImplementedError*).

El parámetro *optimization* se utiliza para especificar el nivel de optimización del archivo de código de bytes. Una cadena de caracteres vacía no representa optimización, por lo que `/foo/bar/baz.py` con una *optimization* de `''` dará como resultado una ruta de código de bytes de `/foo/bar/__pycache__/baz.cpython-32.pyc`. `None` hace que se utilice el nivel de optimización del intérprete. Se usa la representación de cadena de caracteres de cualquier otro valor, por lo que `/foo/bar/baz.py` con una *optimization* de `2` conducirá a la ruta del código de bytes de `/foo/bar/__pycache__/baz.cpython-32.opt-2.pyc`. La representación de cadena de caracteres *optimization* solo puede ser alfanumérica, de lo contrario se lanza *ValueError*.

El parámetro *debug_override* está obsoleto y se puede usar para anular el valor del sistema para `__debug__`. Un valor `True` es el equivalente a establecer *optimization* en la cadena de caracteres vacía. Un valor `False` es lo mismo que establecer *optimization* en `1`. Si tanto *debug_override* como *optimization* no es `None`, entonces se lanza *TypeError*.

Nuevo en la versión 3.4.

Distinto en la versión 3.5: Se agregó el parámetro *optimization* y el parámetro *debug_override* quedó obsoleto.

Distinto en la versión 3.6: Acepta un *path-like object*.

`importlib.util.`**`source_from_cache`**(*path*)
Dado el *path* a un nombre de archivo **PEP 3147**, retorna la ruta del archivo del código fuente asociado. 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** o **PEP 488**, se lanza un *ValueError*. Si `sys.implementation.cache_tag` no está definido, se lanza *NotImplementedError*.

Nuevo en la versión 3.4.

Distinto en la versión 3.6: Acepta un *path-like object*.

`importlib.util.`**`decode_source`**(*source_bytes*)
Decodifica los bytes dados que representan el código fuente y los retorna como una cadena de caracteres con nuevas líneas universales (como lo requiere *importlib.abc.InspectLoader.get_source()*).

Nuevo en la versión 3.4.

`importlib.util.`**`resolve_name`**(*name*, *package*)
Resuelve un nombre de módulo relativo a uno absoluto.

Si **name** no tiene puntos iniciales, entonces **name** simplemente se retorna. Esto permite el uso como `importlib.util.resolve_name('sys', __spec__.parent)` sin hacer una verificación para ver si se necesita el argumento **package**.

ImportError se lanza si **name** es un nombre de módulo relativo pero **package** es un valor falso (por ejemplo, `None` o la cadena de caracteres vacía). También se lanza *ImportError* un nombre relativo que escaparía del paquete que lo contiene (por ejemplo, solicitando `..bacon` desde el paquete `spam`).

Nuevo en la versión 3.3.

Distinto en la versión 3.9: Para mejorar la coherencia con las declaraciones de importación, aumente *ImportError* en lugar de *ValueError* para intentos de importación relativa no válidos.

`importlib.util.`**`find_spec`**(*name*, *package=None*)
Busca el *spec* para un módulo, opcionalmente relativo al nombre del **package** especificado. Si el módulo está en *sys.modules*, se retorna `sys.modules[name].__spec__` (a menos que la especificación sea `None` o no esté establecida, en cuyo caso se lanza *ValueError*). De lo contrario, se realiza una búsqueda utilizando *sys.meta_path*. Se retorna `None` si no se encuentra ninguna especificación.

Si **name** es para un submódulo (contiene un punto), el módulo principal se importa automáticamente.

name y **package** funcionan igual que para `import_module()`.

Nuevo en la versión 3.4.

Distinto en la versión 3.7: Lanza *ModuleNotFoundError* en lugar de *AttributeError* si **package** no es de hecho un paquete (es decir, carece de un atributo __path__).

importlib.util.**module_from_spec**(*spec*)
Cree un nuevo módulo basado en **spec** y *spec.loader.create_module*.

Si *spec.loader.create_module* no retorna None, no se restablecerán los atributos preexistentes. Además, no se lanzará *AttributeError* si se activa mientras se accede a **spec** o se establece un atributo en el módulo.

Esta función es preferible a usar *types.ModuleType* para crear un nuevo módulo ya que **spec** se usa para establecer tantos atributos de importación controlados en el módulo como sea posible.

Nuevo en la versión 3.5.

@importlib.util.**module_for_loader**
Un *decorator* para *importlib.abc.Loader.load_module()* para manejar la selección del objeto de módulo adecuado para cargar. Se espera que el método decorado tenga una firma de llamada que tome dos argumentos posicionales (por ejemplo, load_module(self, module)) para los cuales el segundo argumento será el módulo **object** que usará el cargador. Tenga en cuenta que el decorador no funcionará con métodos estáticos debido a la suposición de dos argumentos.

El método decorado tomará el **name** del módulo que se cargará como se esperaba para un *loader*. Si el módulo no se encuentra en *sys.modules*, se construye uno nuevo. Independientemente de la procedencia del módulo, __loader__ se establece en **self** y __package__ se establece en función de lo que retorna *importlib.abc.InspectLoader.is_package()* (si está disponible) . Estos atributos se establecen incondicionalmente para admitir la recarga.

Si el método decorado lanza una excepción y se agrega un módulo a *sys.modules*, entonces el módulo se eliminará para evitar que un módulo parcialmente inicializado quede en *sys.modules*. Si el módulo ya estaba en *sys.modules* entonces se deja solo.

Distinto en la versión 3.3: __loader__ y __package__ se configuran automáticamente (cuando es posible).

Distinto en la versión 3.4: Establece __name__, __loader__ __package__ incondicionalmente para apoyar la recarga.

Obsoleto desde la versión 3.4: La maquinaria de importación ahora realiza directamente toda la funcionalidad proporcionada por esta función.

@importlib.util.**set_loader**
Un *decorator* para *importlib.abc.Loader.load_module()* para establecer el atributo __loader__ en el módulo retornado. Si el atributo ya está configurado, el decorador no hace nada. Se asume que el primer argumento posicional del método envuelto (es decir, self) es lo que se debe establecer en __loader__.

Distinto en la versión 3.4: Establece __loader__ si está configurado como None, como si el atributo no existiera.

Obsoleto desde la versión 3.4: La maquinaria de importación se encarga de esto automáticamente.

@importlib.util.**set_package**
Un *decorator* para *importlib.abc.Loader.load_module()* para establecer el atributo __package__ en el módulo retornado. Si __package__ está configurado y tiene un valor diferente a None, no se cambiará.

Obsoleto desde la versión 3.4: La maquinaria de importación se encarga de esto automáticamente.

importlib.util.**spec_from_loader**(*name*, *loader*, *, *origin=None*, *is_package=None*)
Una función de fábrica para crear una instancia de *ModuleSpec* basada en un cargador. Los parámetros tienen el mismo significado que para ModuleSpec. La función utiliza APIs disponibles de *loader*, tal como InspectLoader.is_package(), para completar cualquier información que falte en la especificación.

Nuevo en la versión 3.4.

importlib.util.**spec_from_file_location**(*name*, *location*, *, *loader=None*, *submodule_search_locations=None*)

Una función de fábrica para crear una instancia de *ModuleSpec* basada en la ruta a un archivo. La información que falte se completará en la especificación mediante el uso de las API de loader y la implicación de que el módulo estará basado en archivos.

Nuevo en la versión 3.4.

Distinto en la versión 3.6: Acepta un *path-like object*.

importlib.util.**source_hash**(*source_bytes*)

Retorna el hash de *source_bytes* como bytes. Un archivo `.pyc` basado en hash incrusta *source_hash()* del contenido del archivo fuente correspondiente en su encabezado.

Nuevo en la versión 3.7.

class importlib.util.**LazyLoader**(*loader*)

Una clase que pospone la ejecución del cargador de un módulo hasta que el módulo tiene acceso a un atributo.

Esta clase **solo** funciona con cargadores que definen *exec_module()* ya que se requiere control sobre qué tipo de módulo se usa para el módulo. Por esas mismas razones, el método del cargador *create_module()* debe retornar `None` o un tipo para el cual su atributo `__class__` se puede mutar junto con no usar *slots*. Finalmente, los módulos que sustituyen el objeto colocado en *sys.modules* no funcionarán ya que no hay forma de reemplazar correctamente las referencias del módulo en todo el intérprete de forma segura; *ValueError* se genera si se detecta tal sustitución.

Nota: Para proyectos donde el tiempo de inicio es crítico, esta clase permite minimizar potencialmente el costo de cargar un módulo si nunca se usa. Para proyectos en los que el tiempo de inicio no es esencial, el uso de esta clase se desaconseja **en gran medida** debido a que los mensajes de error creados durante la carga se posponen y, por lo tanto, ocurren fuera de contexto.

Nuevo en la versión 3.5.

Distinto en la versión 3.6: Comenzó a llamar *create_module()*, eliminando la advertencia de compatibilidad para *importlib.machinery.BuiltinImporter* y *importlib.machinery.ExtensionFileLoader*.

classmethod factory(*loader*)

A class 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 Ejemplos

Importar programáticamente

Para importar un módulo mediante programación, use *importlib.import_module()*.

```
import importlib

itertools = importlib.import_module('itertools')
```

Comprobando si se puede importar un módulo

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()*.

Note that if `name` is a submodule (contains a dot), *importlib.util.find_spec()* will import the parent module.

```
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:
    # 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")
```

Importar un archivo fuente directamente

Para importar un archivo fuente de Python directamente, use la siguiente receta (solo Python 3.5 y más reciente):

```
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)
```

Implementar importaciones diferidas

El ejemplo de abajo muestra cómo implementar importaciones diferidas:

```
>>> import importlib.util
>>> import sys
>>> def lazy_import(name):
...     spec = importlib.util.find_spec(name)
...     loader = importlib.util.LazyLoader(spec.loader)
...     spec.loader = loader
...     module = importlib.util.module_from_spec(spec)
...     sys.modules[name] = module
...     loader.exec_module(module)
...     return module
...
>>> lazy_typing = lazy_import("typing")
>>> #lazy_typing is a real module object,
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> #but it is not loaded in memory yet.
>>> lazy_typing.TYPE_CHECKING
False
```

Configurar un importador

Para personalizaciones profundas de la importación, normalmente desea implementar un *importador*. Esto significa administrar tanto el lado *finder* como *loader* de las cosas. Para los buscadores, hay dos sabores para elegir según sus necesidades: un *meta path finder* o un *path entry finder*. El primero es lo que pondrías en *sys.meta_path* mientras que el segundo es lo que creas usando un *path entry hook* en *sys.path_hooks* que funciona con sys .path entradas para crear potencialmente un buscador. Este ejemplo le mostrará cómo registrar sus propios importadores para que import los utilice (para crear un importador para usted, lea la documentación de las clases apropiadas definidas dentro de este paquete):

```
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))
```

Aproximando `importlib.import_module()`

La importación en sí está implementada en código Python, lo que permite exponer la mayor parte de la maquinaria de importación a través de importlib. Lo siguiente ayuda a ilustrar las diversas API que importlib expone al proporcionar una implementación aproximada de *importlib.import_module()* (Python 3.4 y más reciente para el uso de importlib, Python 3.6 y más reciente para otras partes del código).

```
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:
```

(continué en la próxima página)

(proviene de la página anterior)

```
        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 Usando `importlib.metadata`

Nuevo en la versión 3.8.

Distinto en la versión 3.10: `importlib.metadata` ya no es provisional.

Source code: Lib/importlib/metadata/__init__.py

`importlib.metadata` es una biblioteca que proporciona acceso a los metadatos del paquete instalado. Construida en parte sobre el sistema de importación de Python, esta biblioteca tiene la intención de reemplazar una funcionalidad similar ofrecida por la API del punto de entrada y la API de metadatos de `pkg_resources`. Junto con *`importlib.resources`* en Python 3.7 y versiones posteriores (respaldada como importlib_resources para versiones anteriores de Python), esto puede eliminar la necesidad de usar el paquete `pkg_resources`, antiguo y menos eficiente.

Por «paquete instalado» generalmente nos referimos a un paquete de terceros instalado en el directorio `site-packages` de Python a través de herramientas como pip. Específicamente, significa un paquete con un directorio reconocible `dist-info` o `egg-info` y metadatos definidos por **PEP 566** o sus especificaciones anteriores. De forma predeterminada, los metadatos del paquete pueden vivir en el sistema de archivos o en archivos zip en *`sys.path`*. A través de un mecanismo de extensión, los metadatos pueden vivir casi en cualquier lugar.

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:

```
$ python -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', 'Requires-Python', 'Provides-Extra',
↪'Provides-Extra', 'Requires-Dist', 'Requires-Dist']
```

(continúe en la próxima página)

(proviene de la página anterior)

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 una colección de todos los puntos de entrada. 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. También hay atributos `.module`, `.attr` y `.extras` para obtener los componentes del atributo `.value`.

Consultar todos los puntos de entrada:

```
>>> eps = entry_points()
```

La función `entry_points()` retorna un objeto `EntryPoints`, una secuencia de todos los objetos `EntryPoint` con atributos `names` y `groups` por conveniencia:

```
>>> sorted(eps.groups)
['console_scripts', 'distutils.commands', 'distutils.setup_keywords', 'egg_info.
↪writers', 'setuptools.installation']
```

`EntryPoints` tiene un método `select` para seleccionar puntos de entrada que coincidan con propiedades específicas. Seleccione los puntos de entrada en el grupo `console_scripts`:

```
>>> scripts = eps.select(group='console_scripts')
```

De manera equivalente, ya que `entry_points` para argumentos de palabra clave para seleccionar:

```
>>> scripts = entry_points(group='console_scripts')
```

Elige un script específico llamado «wheel» (que se encuentra en el proyecto *wheel*):

```
>>> 'wheel' in scripts.names
True
>>> wheel = scripts['wheel']
```

De manera equivalente, consulta por ese punto de entrada durante la selección:

```
>>> (wheel,) = entry_points(group='console_scripts', name='wheel')
>>> (wheel,) = entry_points().select(group='console_scripts', name='wheel')
```

Inspeccionar el punto de entrada resuelto:

```
>>> wheel
EntryPoint(name='wheel', value='wheel.cli:main', group='console_scripts')
>>> wheel.module
'wheel.cli'
>>> wheel.attr
'main'
>>> wheel.extras
[]
>>> main = wheel.load()
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> main
<function main at 0x103528488>
```

The `group` and `name` are arbitrary values defined by the package author and usually a client will wish to resolve all entry points for a particular group. Read the setuptools docs for more information on entry points, their definition, and usage.

Nota de compatibilidad

Los puntos de entrada «seleccionables» se introdujeron en `importlib_metadata` 3.6 y Python 3.10. Antes de esos cambios, `entry_points` no aceptaba parámetros y siempre retornaba un diccionario de puntos de entrada, cifrados por grupo. Para la compatibilidad, si no se pasa ningún parámetro a *entry_points*, se retorna un objeto `SelectableGroups`, implementando esa interfaz de diccionario. En el futuro, llamar a `entry_points` sin parámetros retornará un objeto `EntryPoints`. Los usuarios deberían confiar en la interfaz de selección para recuperar los puntos de entrada por grupo.

Metadatos de distribución

Cada distribución incluye algunos metadatos, que se pueden extraer utilizando la función `metadata()`:

```
>>> wheel_metadata = metadata('wheel')
```

Las claves de la estructura de datos retornada un `PackageMetadata`, 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.*'
```

`PackageMetadata` también presenta un atributo `json` que retorna todos los metadatos en un formulario compatible con JSON por **PEP 566**:

```
>>> wheel_metadata.json['requires_python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

Distinto en la versión 3.10: La `Descripción` ahora se incluye en los metadatos cuando se presenta a través de la carga útil. Se han eliminado los caracteres de continuación de línea.

Nuevo en la versión 3.10: El atributo `json` fue añadido.

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

También se puede obtener el conjunto completo de archivos contenidos dentro de una distribución. La función `files()` toma el nombre de un paquete de distribución y retorna todos los archivos instalados por esta distribución. Cada objeto de archivo retornado es un `PackagePath`, un objeto derivado de *pathlib.PurePath* con las propiedades adicionales `dist`, `size` y `hash` según indican los metadatos. Por ejemplo:

```
>>> 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
```

También puede usar el método `locate` para obtener la ruta absoluta al archivo:

```
>>> util.locate()
PosixPath('/home/gustav/example/lib/site-packages/wheel/util.py')
```

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.

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'"]
```

Distribuciones de paquetes

Un método conveniente para resolver la distribución o distribuciones (en el caso de un paquete de espacio de nombres) para paquetes o módulos de Python de nivel superior:

```
>>> packages_distributions()
{'importlib_metadata': ['importlib-metadata'], 'yaml': ['PyYAML'], 'jaraco': [
↪'jaraco.classes', 'jaraco.functools'], ...}
```

Nuevo en la versión 3.10.

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'
```

El conjunto completo de metadatos disponible no está descripto aquí. Consultar **PEP 566** para detalles adicionales.

31.6.4 Extendiendo el algoritmo de búsqueda

Debido a que los metadatos de los paquetes no están disponibles a través de las búsquedas en *`sys.path`*, o en los cargadores de paquetes directamente, los metadatos de un paquete se encuentran a través del sistema de importación finders. Para encontrar los metadatos de un paquete de distribución, `importlib.metadata` consulta la lista de *meta path finders* en *`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.

La clase abstracta *`importlib.abc.MetaPathFinder`* define la interfaz que se espera de los buscadores por el sistema de importación de Python. `importlib.metadata` amplía este protocolo buscando una `find_distributions` opcional invocable en los buscadores desde *`sys.meta_path`* y presenta esta interfaz extendida como la clase base abstracta `DistributionFinder`, que define este método abstracto:

```
@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 subclasificar `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.

CAPÍTULO 32

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 `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()*.

32.1.1 Gramática abstracta

La gramática abstracta se define actualmente de la siguiente manera:

```
-- ASDL's 4 builtin types are:
-- identifier, int, string, constant

module Python
{
    mod = Module(stmt* body, type_ignore* type_ignores)
        | Interactive(stmt* body)
        | Expression(expr body)
        | FunctionType(expr* argtypes, expr returns)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns,
```

(continué en la próxima página)

(proviene de la página anterior)

```
                      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)

          | Match(expr subject, match_case* cases)

          | 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)
          | Nonlocal(identifier* names)
          | Expr(expr value)
          | Pass | Break | Continue

          -- 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
          | 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, expr 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)

          -- can appear only in Subscript
          | Slice(expr? lower, expr? upper, expr? step)

           -- 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

    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

    comprehension = (expr target, expr iter, expr* ifs, int is_async)

    excepthandler = 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)
               attributes (int lineno, int col_offset, int? end_lineno, int? end_
↪col_offset)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)
             attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

    withitem = (expr context_expr, expr? optional_vars)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    match_case = (pattern pattern, expr? guard, stmt* body)

    pattern = MatchValue(expr value)
            | MatchSingleton(constant value)
            | MatchSequence(pattern* patterns)
            | MatchMapping(expr* keys, pattern* patterns, identifier? rest)
            | MatchClass(expr cls, pattern* patterns, identifier* kwd_attrs,␣
↪pattern* kwd_patterns)

            | MatchStar(identifier? name)
            -- The optional "rest" MatchMapping parameter handles capturing extra␣
↪mapping keys

            | MatchAs(pattern? pattern, identifier? name)
            | MatchOr(pattern* patterns)

             attributes (int lineno, int col_offset, int end_lineno, int end_col_
↪offset)

    type_ignore = TypeIgnore(int lineno, string tag)
}
```

32.1.2 Clases Nodo

class ast.**AST**

This is the base of all AST node classes. The actual node classes are derived from the `Parser/Python.asdl` file, which is reproduced *above*. They are defined in the `_ast` C module and re-exported in *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*. *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 *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.

Distinto en la versión 3.9: Los índices simples se representan por su valor, los segmentos extendidos se representan como tuplas.

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, crear sus instancias retornará una instancia de una clase diferente.

Obsoleto desde la versión 3.9: Las clases antiguas `ast.Index` y `ast.ExtSlice` todavía están disponibles, pero se eliminarán en futuras versiones de Python. Mientras tanto, crear sus instancias retornará una instancia de una clase diferente.

Nota: Las descripciones de las clases de nodo específicas mostradas aquí fueron adaptadas inicialmente del fantástico proyecto Green Tree Snakes y todos sus contribuidores.

Literales

class `ast.`**Constant**(*value*)

Un valor constante. El atributo `value` del literal `Constant` contiene el objeto de Python que este representa. Los valores representados pueden ser de tipos simple como un número, una cadena de caracteres o `None`; pero también pueden ser de tipos de contenedores inmutables (tuplas y `frozensets`) si todos sus elementos son constantes.

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class `ast.`**FormattedValue**(*value*, *conversion*, *format_spec*)

Nodo que representa un único campo de formato en una `f-string`. Si la cadena de caracteres contiene un único campo de formato y nada más, el nodo puede estar aislado de otra manera aparece en *JoinedStr*.

- `value` es cualquier nodo de expresión (como un literal, una variable o una llamada a función).
- `conversion` es un entero:
  - -1: sin formato

  - 115: `!s` formato de cadena de caracteres
  - 114: `!r` formato repr
  - 97: `!a` formato ascii

- `format_spec` es un nodo *JoinedStr* que representa el formato del valor, o `None` si no se ha especificado un formato. Ambos, `conversion` y `format_spec`, pueden estar especificados al mismo tiempo.

class ast.**JoinedStr**(*values*)

Un f-string que comprende una serie de nodos *FormattedValue* y *Constant*.

```
>>> print(ast.dump(ast.parse('f"sin({a}) is {sin(a):.3}"', mode='eval'),␣
↪indent=4))
Expression(
    body=JoinedStr(
        values=[
            Constant(value='sin('),
            FormattedValue(
                value=Name(id='a', ctx=Load()),
                conversion=-1),
            Constant(value=') is '),
            FormattedValue(
                value=Call(
                    func=Name(id='sin', ctx=Load()),
                    args=[
                        Name(id='a', ctx=Load())],
                    keywords=[]),
                conversion=-1,
                format_spec=JoinedStr(
                    values=[
                        Constant(value='.3')]))])))
```

class ast.**List**(*elts*, *ctx*)

class ast.**Tuple**(*elts*, *ctx*)

Una lista o tupla. `elts` contiene una lista de nodos que representa a los elementos. `ctx` es *Store* si el contenedor es un objetivo de asignación (por ejemplo `(x,y)=something`), y *Load* en cualquier otro caso.

```
>>> print(ast.dump(ast.parse('[1, 2, 3]', mode='eval'), indent=4))
Expression(
    body=List(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
>>> print(ast.dump(ast.parse('(1, 2, 3)', mode='eval'), indent=4))
Expression(
    body=Tuple(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
```

class ast.**Set**(*elts*)

Un set. `elts` contiene una lista de nodos que representa a un set de elementos.

```
>>> print(ast.dump(ast.parse('{1, 2, 3}', mode='eval'), indent=4))
Expression(
    body=Set(
        elts=[
```

(continué en la próxima página)

(proviene de la página anterior)

```
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)]))
```

class ast.**Dict**(*keys*, *values*)

Un diccionario. `keys` y `values` contienen listas de nodos que representan las claves y los valores respectivamente en el orden correspondiente (el orden que retornaría `dictionary.keys()` y `dictionary.values()`).

Cuando se desempaqueta un diccionario utilizando literales de diccionario, la expresión a ser expandida va en la lista `values`, con `None` en la posición correspondiente en `keys`.

```
>>> print(ast.dump(ast.parse('{"a":1, **d}', mode='eval'), indent=4))
Expression(
    body=Dict(
        keys=[
            Constant(value='a'),
            None],
        values=[
            Constant(value=1),
            Name(id='d', ctx=Load())]))
```

Variables

class ast.**Name**(*id*, *ctx*)

Un nombre de variable. `id` contiene el nombre de una cadena de caracteres y `ctx` es uno de los siguientes tipos.

class ast.**Load**
class ast.**Store**
class ast.**Del**

Referencias a variables que pueden ser usadas para cargar el valor de una variable, asignar un nuevo valor o borrarlo. Las referencias a variables reciben un contexto para distinguir entre estos casos.

```
>>> print(ast.dump(ast.parse('a'), indent=4))
Module(
    body=[
        Expr(
            value=Name(id='a', ctx=Load()))],
    type_ignores=[])

>>> print(ast.dump(ast.parse('a = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store())],
            value=Constant(value=1))],
    type_ignores=[])

>>> print(ast.dump(ast.parse('del a'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='a', ctx=Del())])],
    type_ignores=[])
```

class ast.**Starred**(*value*, *ctx*)

Una referencia a variable `*var`. `value` contiene la variable, típicamente un nodo *Name*. Este tipo puede ser

usado cuando se construye un nodo *Call* con `*args`.

```
>>> print(ast.dump(ast.parse('a, *b = it'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Starred(
                            value=Name(id='b', ctx=Store()),
                            ctx=Store())],
                    ctx=Store())],
            value=Name(id='it', ctx=Load()))],
    type_ignores=[])
```

Expresiones

class ast.**Expr**(*value*)

Cuando una expresión, como un llamado a función, aparece como una declaración por sí misma sin que su valor de retorno se use o se almacene, está dentro de este contenedor. `value` contiene uno de los otros nodos en esta sección, un nodo *Constant*, *Name*, *Lambda*, *Yield* o *YieldFrom*.

```
>>> print(ast.dump(ast.parse('-a'), indent=4))
Module(
    body=[
        Expr(
            value=UnaryOp(
                op=USub(),
                operand=Name(id='a', ctx=Load()))))],
    type_ignores=[])
```

class ast.**UnaryOp**(*op*, *operand*)

Una operación unaria. `op` es el operador y `operand` es cualquier nodo de expresión.

class ast.**UAdd**
class ast.**USub**
class ast.**Not**
class ast.**Invert**

Tokens de operador unario. *Not* es la palabra clave `not`, *Invert* es el operador ~.

```
>>> print(ast.dump(ast.parse('not x', mode='eval'), indent=4))
Expression(
    body=UnaryOp(
        op=Not(),
        operand=Name(id='x', ctx=Load())))
```

class ast.**BinOp**(*left*, *op*, *right*)

Una operación binaria (como la suma o división(. `op` es el operador, y `left` y `right` son cualquier nodo de expresión.

```
>>> print(ast.dump(ast.parse('x + y', mode='eval'), indent=4))
Expression(
    body=BinOp(
        left=Name(id='x', ctx=Load()),
        op=Add(),
        right=Name(id='y', ctx=Load())))
```

class ast.**Add**
class ast.**Sub**

class ast.**Mult**
class ast.**Div**
class ast.**FloorDiv**
class ast.**Mod**
class ast.**Pow**
class ast.**LShift**
class ast.**RShift**
class ast.**BitOr**
class ast.**BitXor**
class ast.**BitAnd**
class ast.**MatMult**
Tokens de operador binario.

class ast.**BoolOp**(*op*, *values*)
Una operación booleana, "or" y "and". `op` es *Or* o *And*. `values` son los valores involucrados. Operaciones consecutivas con el mismo operador, como `a or b or c`, colapsan en un nodo con varios valores.

Esto no incluye `not`, el cual es un *UnaryOp*.

```
>>> print(ast.dump(ast.parse('x or y', mode='eval'), indent=4))
Expression(
    body=BoolOp(
        op=Or(),
        values=[
            Name(id='x', ctx=Load()),
            Name(id='y', ctx=Load())]))
```

class ast.**And**
class ast.**Or**
Tokens de operador booleano.

class ast.**Compare**(*left*, *ops*, *comparators*)
Una comparación de dos o más valores. `left` es el primer valor en la comparación, `ops` es la lista de operadores, y `comparators` es la lista de valores después de el primer elemento en la comparación.

```
>>> print(ast.dump(ast.parse('1 <= a < 10', mode='eval'), indent=4))
Expression(
    body=Compare(
        left=Constant(value=1),
        ops=[
            LtE(),
            Lt()],
        comparators=[
            Name(id='a', ctx=Load()),
            Constant(value=10)]))
```

class ast.**Eq**
class ast.**NotEq**
class ast.**Lt**
class ast.**LtE**
class ast.**Gt**
class ast.**GtE**
class ast.**Is**
class ast.**IsNot**
class ast.**In**
class ast.**NotIn**
Tokens de operador de comparación.

class ast.**Call**(*func*, *args*, *keywords*, *starargs*, *kwargs*)
Un llamado a función. `func` is la función, la cual suele ser un objeto *Name* o *Attribute*. De los argumentos:

- `args` contiene una lista de argumentos pasados por posición.

- `keywords` contiene una lista de objetos *`keyword`* que representan argumentos pasados por nombre clave.

Cuando se crea un nodo `Call`, `args` y `keywords` son requeridos pero pueden ser listas vacías. `starargs` y `kwargs` son opcionales.

```
>>> print(ast.dump(ast.parse('func(a, b=c, *d, **e)', mode='eval'), indent=4))
Expression(
    body=Call(
        func=Name(id='func', ctx=Load()),
        args=[
            Name(id='a', ctx=Load()),
            Starred(
                value=Name(id='d', ctx=Load()),
                ctx=Load())],
        keywords=[
            keyword(
                arg='b',
                value=Name(id='c', ctx=Load())),
            keyword(
                value=Name(id='e', ctx=Load()))]))
```

class `ast.`**keyword**(*arg*, *value*)

Un argumento de palabra clave para una llamada de función o definición de clase. `arg` es una cadena de caracteres sin formato del nombre del parámetro, `valor` es un nodo para pasar.

class `ast.`**IfExp**(*test*, *body*, *orelse*)

Una expresión como `a if b else c`. Cada campo contiene un único nodo, por lo que en el siguiente ejemplo, todos son nodos *`Name`*.

```
>>> print(ast.dump(ast.parse('a if b else c', mode='eval'), indent=4))
Expression(
    body=IfExp(
        test=Name(id='b', ctx=Load()),
        body=Name(id='a', ctx=Load()),
        orelse=Name(id='c', ctx=Load())))
```

class `ast.`**Attribute**(*value*, *attr*, *ctx*)

Acceso a atributos, por ejemplo `d.keys`. `value` es un nodo, típicamente un *`Name`*. `attr` es una simple cadena de caracteres que da el nombre del atributo, y `ctx` es *`Load`*, *`Store`* o *`Del`* de acuerdo a cómo se actúe sobre el atributo.

```
>>> print(ast.dump(ast.parse('snake.colour', mode='eval'), indent=4))
Expression(
    body=Attribute(
        value=Name(id='snake', ctx=Load()),
        attr='colour',
        ctx=Load()))
```

class `ast.`**NamedExpr**(*target*, *value*)

Una expresión con nombre. Este nodo AST es producido por el operador de expresiones de asignación (también conocido como el operador walrus). A diferencia del nodo *`Assign`* en el cual el primer argumento puede ser varios nodos, en este caso `target` y `value` deben ser nodos únicos.

```
>>> print(ast.dump(ast.parse('(x := 4)', mode='eval'), indent=4))
Expression(
    body=NamedExpr(
        target=Name(id='x', ctx=Store()),
        value=Constant(value=4)))
```

Subindexado

class ast.**Subscript**(*value*, *slice*, *ctx*)
Un subíndice, como `l[1].value` es el objeto subindicado (usualmente una secuencia o mapeo). `slice` es un índice, un segmento o una clave. Este puede ser una *Tuple* y contener un *Slice*. `ctx` es *Load*, *Store* or *Del* de acuerdo a la acción tomada con el subíndice.

```
>>> print(ast.dump(ast.parse('l[1:2, 3]', mode='eval'), indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Tuple(
            elts=[
                Slice(
                    lower=Constant(value=1),
                    upper=Constant(value=2)),
                Constant(value=3)],
            ctx=Load()),
        ctx=Load()))
```

class ast.**Slice**(*lower*, *upper*, *step*)
Una segmentación regular (en la forma `lower:upper` o `lower:upper:step`). Puede ocurrir solamente dentro del campo *slice* de *Subscript*, ya sea directamente o como un elemento de *Tuple*.

```
>>> print(ast.dump(ast.parse('l[1:2]', mode='eval'), indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Slice(
            lower=Constant(value=1),
            upper=Constant(value=2)),
        ctx=Load()))
```

Comprensiones

class ast.**ListComp**(*elt*, *generators*)
class ast.**SetComp**(*elt*, *generators*)
class ast.**GeneratorExp**(*elt*, *generators*)
class ast.**DictComp**(*key*, *value*, *generators*)
Listas y sets por comprensión, expresiones de generadores, y diccionarios por comprensión. `elt` (o `key` y `value`) es un único nodo que representa la parte que va a ser evaluada por cada item.

`generators` es una lista de nodos *comprehension*.

```
>>> print(ast.dump(ast.parse('[x for x in numbers]', mode='eval'), indent=4))
Expression(
    body=ListComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                ifs=[],
                is_async=0)]))
>>> print(ast.dump(ast.parse('{x: x**2 for x in numbers}', mode='eval'),␣
↪indent=4))
Expression(
    body=DictComp(
        key=Name(id='x', ctx=Load()),
        value=BinOp(
```

(continué en la próxima página)

(proviene de la página anterior)

```
            left=Name(id='x', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                ifs=[],
                is_async=0)]))
>>> print(ast.dump(ast.parse('{x for x in numbers}', mode='eval'), indent=4))
Expression(
    body=SetComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                ifs=[],
                is_async=0)]))
```

class ast.**comprehension**(*target*, *iter*, *ifs*, *is_async*)

Una cláusula `for` en una comprensión. `target` es la referencia a usarse por cada elemento - típicamente un nodo *Name* o *Tuple*. `iter` es el objeto por el cual se itera. `ifs` es una lista de expresiones de prueba: cada cláusula `for` puede tener múltiples `ifs`.

`is_async` indica que una compresión es asíncrona (usando `async for` en lugar de `for`). El valor es un entero (0 o 1).

```
>>> print(ast.dump(ast.parse('[ord(c) for line in file for c in line]', mode=
↪'eval'),
...                indent=4)) # Multiple comprehensions in one.
Expression(
    body=ListComp(
        elt=Call(
            func=Name(id='ord', ctx=Load()),
            args=[
                Name(id='c', ctx=Load())],
            keywords=[]),
        generators=[
            comprehension(
                target=Name(id='line', ctx=Store()),
                iter=Name(id='file', ctx=Load()),
                ifs=[],
                is_async=0),
            comprehension(
                target=Name(id='c', ctx=Store()),
                iter=Name(id='line', ctx=Load()),
                ifs=[],
                is_async=0)]))

>>> print(ast.dump(ast.parse('(n**2 for n in it if n>5 if n<10)', mode='eval'),
...                indent=4)) # generator comprehension
Expression(
    body=GeneratorExp(
        elt=BinOp(
            left=Name(id='n', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='n', ctx=Store()),
```

(continué en la próxima página)

(proviene de la página anterior)

```
                iter=Name(id='it', ctx=Load()),
                ifs=[
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Lt()],
                        comparators=[
                            Constant(value=10)])],
                is_async=0)]))

>>> print(ast.dump(ast.parse('[i async for i in soc]', mode='eval'),
...                indent=4)) # Async comprehension
Expression(
    body=ListComp(
        elt=Name(id='i', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='i', ctx=Store()),
                iter=Name(id='soc', ctx=Load()),
                ifs=[],
                is_async=1)]))
```

Declaraciones

class ast.**Assign**(*targets*, *value*, *type_comment*)

Una asignación. `targets` es una lista de nodos, y `value` es un nodo único.

Nodos múltiples en `targets` representa asignar el mismo valor a cada uno. El desempaquetado se representa poniendo una *`Tuple`* o *`List`* en `targets`.

type_comment

`type_comment` es una cadena de caracteres opcional con la anotación de tipos como comentario.

```
>>> print(ast.dump(ast.parse('a = b = 1'), indent=4)) # Multiple assignment
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store()),
                Name(id='b', ctx=Store())],
            value=Constant(value=1))],
    type_ignores=[])

>>> print(ast.dump(ast.parse('a,b = c'), indent=4)) # Unpacking
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Name(id='b', ctx=Store())],
                    ctx=Store())],
```

(continué en la próxima página)

(proviene de la página anterior)

```
            value=Name(id='c', ctx=Load()))],
    type_ignores=[])
```

class ast.**AnnAssign**(*target*, *annotation*, *value*, *simple*)

Una asignación con una anotación de tipos. `target` es un nodo único y puede ser un *Name*, a *Attribute* o un *Subscript*. `annotation` es la anotación, como un nodo *Constant* o *Name*. `value` es un único nodo opcional. `simple` es un booleano que es True para un nodo *Name* en *target* que no aparece entre paréntesis y por ende son nombres puros y no expresiones.

```
>>> print(ast.dump(ast.parse('c: int'), indent=4))
Module(
    body=[
        AnnAssign(
            target=Name(id='c', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=1)],
    type_ignores=[])

>>> print(ast.dump(ast.parse('(a): int = 1'), indent=4)) # Annotation with␣
↪parenthesis
Module(
    body=[
        AnnAssign(
            target=Name(id='a', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            value=Constant(value=1),
            simple=0)],
    type_ignores=[])

>>> print(ast.dump(ast.parse('a.b: int'), indent=4)) # Attribute annotation
Module(
    body=[
        AnnAssign(
            target=Attribute(
                value=Name(id='a', ctx=Load()),
                attr='b',
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)],
    type_ignores=[])

>>> print(ast.dump(ast.parse('a[1]: int'), indent=4)) # Subscript annotation
Module(
    body=[
        AnnAssign(
            target=Subscript(
                value=Name(id='a', ctx=Load()),
                slice=Constant(value=1),
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)],
    type_ignores=[])
```

class ast.**AugAssign**(*target*, *op*, *value*)

Asignación aumentada, como `a+=1`. En el siguiente ejemplo, `target` es un nodo *Name* para `x` (con el contexto *Store*), `op` es *Add* y `value` es un *Constant* con valor 1.

El atributo `target` no puede ser de clase *Tuple* o *List*, a diferencia de los objetivos de *Assign*.

```
>>> print(ast.dump(ast.parse('x += 2'), indent=4))
Module(
```

(continué en la próxima página)

(proviene de la página anterior)

```
    body=[
        AugAssign(
            target=Name(id='x', ctx=Store()),
            op=Add(),
            value=Constant(value=2))],
    type_ignores=[])
```

class ast.**Raise**(*exc*, *cause*)

Una declaración `raise`. `exc` es el objeto de excepción a ser lanzado, normalmente un *Call* or *Name*, o `None` para un `raise` solo. `cause` es la parte opcional para `y` en `raise x from y`.

```
>>> print(ast.dump(ast.parse('raise x from y'), indent=4))
Module(
    body=[
        Raise(
            exc=Name(id='x', ctx=Load()),
            cause=Name(id='y', ctx=Load()))],
    type_ignores=[])
```

class ast.**Assert**(*test*, *msg*)

Una aserción. `test` contiene la condición, como un nodo *Compare*. `msg` contiene el mensaje de fallo.

```
>>> print(ast.dump(ast.parse('assert x,y'), indent=4))
Module(
    body=[
        Assert(
            test=Name(id='x', ctx=Load()),
            msg=Name(id='y', ctx=Load()))],
    type_ignores=[])
```

class ast.**Delete**(*targets*)

Contiene una declaración `del`. `targets` es una lista de nodos, como nodos *Name*, *Attribute* o *Subscript*.

```
>>> print(ast.dump(ast.parse('del x,y,z'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='x', ctx=Del()),
                Name(id='y', ctx=Del()),
                Name(id='z', ctx=Del())])],
    type_ignores=[])
```

class ast.**Pass**

Una declaración `pass`.

```
>>> print(ast.dump(ast.parse('pass'), indent=4))
Module(
    body=[
        Pass()],
    type_ignores=[])
```

Otras declaraciones que solo son aplicables dentro de funciones o bucles descritos en otras secciones.

Importaciones

class ast.**Import**(*names*)
Una declaración de importación. `names` es una lista de nodos *alias*.

```
>>> print(ast.dump(ast.parse('import x,y,z'), indent=4))
Module(
    body=[
        Import(
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')])],
    type_ignores=[])
```

class ast.**ImportFrom**(*module*, *names*, *level*)
Representa `form x import y`. `module` es una cadena de caracteres sin formato del nombre "from", sin puntos, o `None` para declaraciones como `from . import foo`. `level` es un entero que contiene el nivel relativo de la importación (0 significa una importación absoluta).

```
>>> print(ast.dump(ast.parse('from y import x,y,z'), indent=4))
Module(
    body=[
        ImportFrom(
            module='y',
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')],
            level=0)],
    type_ignores=[])
```

class ast.**alias**(*name*, *asname*)
Ambos parámetros son cadenas de caracteres sin formato para los nombres. `asname` puede ser `None` si se va a usar el nombre regular.

```
>>> print(ast.dump(ast.parse('from ..foo.bar import a as b, c'), indent=4))
Module(
    body=[
        ImportFrom(
            module='foo.bar',
            names=[
                alias(name='a', asname='b'),
                alias(name='c')],
            level=2)],
    type_ignores=[])
```

Control de flujo

Nota: Cláusulas opcionales como"else" se guardan como una lista vacía si no están presentes.

class ast.**If**(*test*, *body*, *orelse*)
Una declaración `if`. `test` contiene un único nodo, como un nodo *Compare*. `body` y `orelse` contiene cada uno una lista de nodos.

Cláusulas `elif` no tienen una representación especial en AST, pero pueden aparecer como nodos extra *If* dentro de la sección `orelse` del nodo anterior.

```
>>> print(ast.dump(ast.parse("""
... if x:
...     ...
... elif y:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        If(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                If(
                    test=Name(id='y', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))],
                    orelse=[
                        Expr(
                            value=Constant(value=Ellipsis))])])],
    type_ignores=[])
```

class ast.**For**(*target*, *iter*, *body*, *orelse*, *type_comment*)

Un bucle `for`. `target` contiene la(s) variable(s) donde asigna el bucle como un único nodo *Name*, *Tuple* o *List*. `iter` contiene el item por el cual se va recorrer como un único nodo. `body` y `orelse` contienen una lista de nodos a ejecutar. Aquellos en `orelse` son ejecutados si el bucle termina normalmente, en contra de si terminan utilizando la declaración `break`.

type_comment

`type_comment` es una cadena de caracteres opcional con la anotación de tipos como comentario.

```
>>> print(ast.dump(ast.parse("""
... for x in y:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='x', ctx=Store()),
            iter=Name(id='y', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])],
    type_ignores=[])
```

class ast.**While**(*test*, *body*, *orelse*)

Un bucle `while`. `test` contiene la condición, como un nodo *Compare*.

```
>> print(ast.dump(ast.parse("""
... while x:
...     ...
... else:
...     ...
```

(continué en la próxima página)

(proviene de la página anterior)

```
... """), indent=4))
Module(
    body=[
        While(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])],
    type_ignores=[])
```

class ast.**Break**

class ast.**Continue**

Las declaraciones `break` y `continue`.

```
>>> print(ast.dump(ast.parse("""\
... for a in b:
...     if a > 5:
...         break
...     else:
...         continue
...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='a', ctx=Store()),
            iter=Name(id='b', ctx=Load()),
            body=[
                If(
                    test=Compare(
                        left=Name(id='a', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    body=[
                        Break()],
                    orelse=[
                        Continue()])],
            orelse=[])],
    type_ignores=[])
```

class ast.**Try**(*body*, *handlers*, *orelse*, *finalbody*)

Bloques `try`. Todos los atributos son listas de nodos a ejecutar, excepto para `handlers`, el cual es una lista de nodos *`ExceptHandler`*.

```
>>> print(ast.dump(ast.parse("""
... try:
...     ...
... except Exception:
...     ...
... except OtherException as e:
...     ...
... else:
...     ...
... finally:
...     ...
... """), indent=4))
```

(continué en la próxima página)

(proviene de la página anterior)

```
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                ExceptHandler(
                    type=Name(id='OtherException', ctx=Load()),
                    name='e',
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))],
            finalbody=[
                Expr(
                    value=Constant(value=Ellipsis))])],
    type_ignores=[])
```

class ast.**ExceptHandler**(*type*, *name*, *body*)

Una sola cláusula `except`. `type` es el tipo de excepción con el que coincidirá, normalmente un nodo *Name* (o `None` para una cláusula `except:` generalizada). `name` es una cadena sin formato para que el nombre contenga la excepción, o `None` si la cláusula no tiene `as foo`. `body` es una lista de nodos.

```
>>> print(ast.dump(ast.parse("""\
... try:
...     a + 1
... except TypeError:
...     pass
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=BinOp(
                        left=Name(id='a', ctx=Load()),
                        op=Add(),
                        right=Constant(value=1)))],
            handlers=[
                ExceptHandler(
                    type=Name(id='TypeError', ctx=Load()),
                    body=[
                        Pass()])],
            orelse=[],
            finalbody=[])],
    type_ignores=[])
```

class ast.**With**(*items*, *body*, *type_comment*)

Un bloque `with`. `items` es una lista de nodos *withitem* que representan los administradores de contexto, y `body` es el bloque con sangría dentro del contexto.

type_comment

`type_comment` es una cadena de caracteres opcional con la anotación de tipos como comentario.

class ast.**withitem**(*context_expr*, *optional_vars*)
Un administrador de contexto único en un bloque `with`. `context_expr` es el administrador de contexto, a menudo un nodo *Call*. `optional_vars` es un *Name*, *Tuple* o *List* para la parte `as foo`, o `None` si no se usa.

```
>>> print(ast.dump(ast.parse("""\
... with a as b, c as d:
...    something(b, d)
... """), indent=4))
Module(
    body=[
        With(
            items=[
                withitem(
                    context_expr=Name(id='a', ctx=Load()),
                    optional_vars=Name(id='b', ctx=Store())),
                withitem(
                    context_expr=Name(id='c', ctx=Load()),
                    optional_vars=Name(id='d', ctx=Store()))],
            body=[
                Expr(
                    value=Call(
                        func=Name(id='something', ctx=Load()),
                        args=[
                            Name(id='b', ctx=Load()),
                            Name(id='d', ctx=Load())],
                        keywords=[]))])],
    type_ignores=[])
```

La coincidencia de patrones

class ast.**Match**(*subject*, *cases*)
Una declaración `match`. `subject` contiene el sujeto de la coincidencia (el objeto que se compara con los casos) y `cases` contiene un iterable de nodos *match_case* con los diferentes casos.

class ast.**match_case**(*pattern*, *guard*, *body*)
Un patrón de caso único en una declaración `match`. `pattern` contiene el patrón de coincidencia con el que se comparará el sujeto. Tenga en cuenta que los nodos *AST* producidos para patrones difieren de los producidos para expresiones, incluso cuando comparten la misma sintaxis.

El atributo `guard` contiene una expresión que se evaluará si el patrón coincide con el sujeto.

`body` contiene una lista de nodos para ejecutar si el patrón coincide y el resultado de evaluar la expresión de protección es verdadero.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] if x>0:
...         ...
...     case tuple():
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchAs(name='x')]),
```

(continué en la próxima página)

(proviene de la página anterior)

```
                    guard=Compare(
                        left=Name(id='x', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=0)]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='tuple', ctx=Load()),
                        patterns=[],
                        kwd_attrs=[],
                        kwd_patterns=[]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])],
    type_ignores=[])
```

class ast.**MatchValue**(*value*)

Un patrón de valor o literal de coincidencia que se compara por igualdad. `value` es un nodo de expresión. Los nodos de valores permitidos están restringidos como se describe en la documentación de la declaración de coincidencia. Este patrón tiene éxito si el sujeto de la coincidencia es igual al valor evaluado.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case "Relevant":
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchValue(
                        value=Constant(value='Relevant')),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])],
    type_ignores=[])
```

class ast.**MatchSingleton**(*value*)

Un patrón literal de coincidencia que se compara por identidad. `value` es el singleton que se va a comparar con: `None`, `True` o `False`. Este patrón tiene éxito si el sujeto de la coincidencia es la constante dada.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case None:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSingleton(value=None),
                    body=[
                        Expr(
```

(continué en la próxima página)

(proviene de la página anterior)

```
                                value=Constant(value=Ellipsis))])])],
    type_ignores=[])
```

class ast.**MatchSequence**(*patterns*)

Un patrón de secuencia de coincidencia. `patterns` contiene los patrones que se compararán con los elementos del sujeto si el sujeto es una secuencia. Coincide con una secuencia de longitud variable si uno de los subpatrones es un nodo `MatchStar`; de lo contrario, coincide con una secuencia de longitud fija.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
                                value=Constant(value=1)),
                            MatchValue(
                                value=Constant(value=2))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])],
    type_ignores=[])
```

class ast.**MatchStar**(*name*)

Coincide con el resto de la secuencia en un patrón de secuencia de coincidencia de longitud variable. Si `name` no es `None`, una lista que contiene los elementos de secuencia restantes está vinculada a ese nombre si el patrón de secuencia general es exitoso.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2, *rest]:
...         ...
...     case [*_]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
                                value=Constant(value=1)),
                            MatchValue(
                                value=Constant(value=2)),
                            MatchStar(name='rest')]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchSequence(
                        patterns=[
```

(continué en la próxima página)

(proviene de la página anterior)

```
                            MatchStar()]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])],
    type_ignores=[])
```

class ast.**MatchMapping**(*keys*, *patterns*, *rest*)

Un patrón de mapeo de coincidencias. `keys` es una secuencia de nodos de expresión. `patterns` es una secuencia correspondiente de nodos de patrón. `rest` es un nombre opcional que se puede especificar para capturar los elementos de mapeo restantes. Las expresiones clave permitidas están restringidas como se describe en la documentación de la declaración de coincidencia.

Este patrón tiene éxito si el sujeto es un mapeo, todas las expresiones clave evaluadas están presentes en el mapeo y el valor correspondiente a cada clave coincide con el subpatrón correspondiente. Si `rest` no es `None`, un dict que contiene los elementos de mapeo restantes se vincula a ese nombre si el patrón de mapeo general es exitoso.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case {1: _, 2: _}:
...         ...
...     case {**rest}:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchMapping(
                        keys=[
                            Constant(value=1),
                            Constant(value=2)],
                        patterns=[
                            MatchAs(),
                            MatchAs()]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchMapping(keys=[], patterns=[], rest='rest'),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])],
    type_ignores=[])
```

class ast.**MatchClass**(*cls*, *patterns*, *kwd_attrs*, *kwd_patterns*)

Un patrón de clase coincidente. `cls` es una expresión que da la clase nominal que se va a emparejar. `patterns` es una secuencia de nodos de patrón que se compararán con la secuencia definida por la clase de atributos de coincidencia de patrones. `kwd_attrs` es una secuencia de atributos adicionales que deben coincidir (especificados como argumentos de palabra clave en el patrón de clase), `kwd_patterns` son los patrones correspondientes (especificados como valores de palabras clave en el patrón de clase).

Este patrón tiene éxito si el sujeto es una instancia de la clase nominada, todos los patrones posicionales coinciden con los atributos definidos por la clase correspondientes y cualquier atributo de palabra clave especificado coincide con su patrón correspondiente.

Nota: las clases pueden definir una propiedad que retorna self para hacer coincidir un nodo de patrón con la instancia que se está comparando. Varios tipos incorporados también se combinan de esa manera, como se describe en la documentación de la declaración de coincidencia.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case Point2D(0, 0):
...         ...
...     case Point3D(x=0, y=0, z=0):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point2D', ctx=Load()),
                        patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))],
                        kwd_attrs=[],
                        kwd_patterns=[]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point3D', ctx=Load()),
                        patterns=[],
                        kwd_attrs=[
                            'x',
                            'y',
                            'z'],
                        kwd_patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])],
    type_ignores=[])
```

class ast.**MatchAs**(*pattern*, *name*)

Una coincidencia «como patrón», patrón de captura o patrón comodín. `pattern` contiene el patrón de coincidencia con el que se comparará el sujeto. Si el patrón es `None`, el nodo representa un patrón de captura (es decir, un nombre simple) y siempre tendrá éxito.

El atributo `name` contiene el nombre que se vinculará si el patrón tiene éxito. Si `name` es `None`, `pattern` también debe ser `None` y el nodo representa el patrón comodín.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] as y:
...         ...
...     case _:
...         ...
... """), indent=4))
Module(
    body=[
```

(continué en la próxima página)

(proviene de la página anterior)

```
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchAs(
                        pattern=MatchSequence(
                            patterns=[
                                MatchAs(name='x')]),
                        name='y'),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchAs(),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])],
    type_ignores=[])
```

class ast.**MatchOr**(*patterns*)

Una coincidencia «o patrón». Un patrón-o hace coincidir cada uno de sus subpatrones con el sujeto, hasta que uno tiene éxito. Entonces se considera que el patrón-o tiene éxito. Si ninguno de los subpatrones tiene éxito, el patrón o falla. El atributo `patterns` contiene una lista de nodos de patrones de coincidencia que se compararán con el sujeto.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] | (y):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchOr(
                        patterns=[
                            MatchSequence(
                                patterns=[
                                    MatchAs(name='x')]),
                            MatchAs(name='y')]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])],
    type_ignores=[])
```

Definiciones de función y clase

class ast.**FunctionDef**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*)

Una definición de función.

- `name` es una cadena de caracteres sin formato del nombre de la función.
- `args` es un nodo *`arguments`*.
- `body` es la lista de nodos dentro de la función.
- `decorator_list` es la lista de decoradores que se aplicarán, almacenados en el exterior primero (es decir, el primero de la lista se aplicará al final).
- `returns` es la anotación de retorno.

type_comment
type_comment es una cadena de caracteres opcional con la anotación de tipos como comentario.

class ast.**Lambda**(*args*, *body*)
lambda es una definición de función mínima que se puede utilizar dentro de una expresión. A diferencia de *FunctionDef*, body tiene un solo nodo.

```
>>> print(ast.dump(ast.parse('lambda x,y: ...'), indent=4))
Module(
    body=[
        Expr(
            value=Lambda(
                args=arguments(
                    posonlyargs=[],
                    args=[
                        arg(arg='x'),
                        arg(arg='y')],
                    kwonlyargs=[],
                    kw_defaults=[],
                    defaults=[]),
                body=Constant(value=Ellipsis)))],
    type_ignores=[])
```

class ast.**arguments**(*posonlyargs*, *args*, *vararg*, *kwonlyargs*, *kw_defaults*, *kwarg*, *defaults*)
Los argumentos para una función.

- posonlyargs, args y kwonlyargs son listas de nodos *arg*.
- vararg y kwarg son nodos *arg* únicos, en referencia a los parámetros *args, **kwargs.
- kw_defaults es una lista de valores predeterminados para argumentos de solo palabras clave. Si uno es None, se requiere el argumento correspondiente.
- defaults es una lista de valores predeterminados para argumentos que se pueden pasar posicionalmente. Si hay menos valores predeterminados, corresponden a los últimos n argumentos.

class ast.**arg**(*arg*, *annotation*, *type_comment*)
Un solo argumento en una lista. arg es una cadena sin formato del nombre del argumento, annotation es su anotación, como un nodo Str o *Name*.

type_comment
type_comment es una cadena opcional con la anotación de tipo como comentario

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... def f(a: 'annotation', b=1, c=2, *d, e, f=3, **g) -> 'return annotation':
...     pass
... """), indent=4))
Module(
    body=[
        FunctionDef(
            name='f',
            args=arguments(
                posonlyargs=[],
                args=[
                    arg(
                        arg='a',
                        annotation=Constant(value='annotation')),
                    arg(arg='b'),
                    arg(arg='c')],
                vararg=arg(arg='d'),
                kwonlyargs=[
                    arg(arg='e'),
```

(continué en la próxima página)

(proviene de la página anterior)

```
                    arg(arg='f')],
                kw_defaults=[
                    None,
                    Constant(value=3)],
                kwarg=arg(arg='g'),
                defaults=[
                    Constant(value=1),
                    Constant(value=2)]),
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())],
            returns=Constant(value='return annotation'))],
    type_ignores=[])
```

class ast.**Return**(*value*)

Una declaración `return`.

```
>>> print(ast.dump(ast.parse('return 4'), indent=4))
Module(
    body=[
        Return(
            value=Constant(value=4))],
    type_ignores=[])
```

class ast.**Yield**(*value*)

class ast.**YieldFrom**(*value*)

Una expresión `yield` o `yield from`. Debido a que se trata de expresiones, deben incluirse en un nodo *Expr* si no se utiliza el valor retornado.

```
>>> print(ast.dump(ast.parse('yield x'), indent=4))
Module(
    body=[
        Expr(
            value=Yield(
                value=Name(id='x', ctx=Load()))))],
    type_ignores=[])

>>> print(ast.dump(ast.parse('yield from x'), indent=4))
Module(
    body=[
        Expr(
            value=YieldFrom(
                value=Name(id='x', ctx=Load()))))],
    type_ignores=[])
```

class ast.**Global**(*names*)

class ast.**Nonlocal**(*names*)

Declaraciones `global` y `nonlocal`. `names` es una lista de cadenas sin formato.

```
>>> print(ast.dump(ast.parse('global x,y,z'), indent=4))
Module(
    body=[
        Global(
            names=[
                'x',
                'y',
                'z'])],
    type_ignores=[])
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> print(ast.dump(ast.parse('nonlocal x,y,z'), indent=4))
Module(
    body=[
        Nonlocal(
            names=[
                'x',
                'y',
                'z'])],
    type_ignores=[])
```

class ast.**ClassDef**(*name*, *bases*, *keywords*, *starargs*, *kwargs*, *body*, *decorator_list*)

Una definición de clase.

- `name` es una cadena sin formato para el nombre de la clase
- `bases` es una lista de nodos para clases base especificadas explícitamente.
- `keywords` es una lista de nodos *`keyword`*, principalmente para "metaclase". Otras palabras clave se pasarán a la metaclase, según PEP-3115.
- `starargs` y `kwargs` son cada uno un solo nodo, como en una llamada de función. Los starargs se expandirán para unirse a la lista de clases base y los kwargs se pasarán a la metaclase.
- `body` es una lista de nodos que representan el código dentro de la definición de clase.
- `decorator_list` es una lista de nodos, como en *`FunctionDef`*.

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... class Foo(base1, base2, metaclass=meta):
...     pass
... """), indent=4))
Module(
    body=[
        ClassDef(
            name='Foo',
            bases=[
                Name(id='base1', ctx=Load()),
                Name(id='base2', ctx=Load())],
            keywords=[
                keyword(
                    arg='metaclass',
                    value=Name(id='meta', ctx=Load()))],
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())])],
    type_ignores=[])
```

Async y await

class ast.**AsyncFunctionDef**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*)
Una definición de función `async def`. Tiene los mismos campos que *FunctionDef*.

class ast.**Await**(*value*)
Una expresión `await`. `value` es lo que espera. Solo válido en el cuerpo de un *AsyncFunctionDef*.

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
... """), indent=4))
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(
                posonlyargs=[],
                args=[],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func', ctx=Load()),
                            args=[],
                            keywords=[]))))],
            decorator_list=[])],
    type_ignores=[])
```

class ast.**AsyncFor**(*target*, *iter*, *body*, *orelse*, *type_comment*)
class ast.**AsyncWith**(*items*, *body*, *type_comment*)
Bucles `async for` y administradores de contexto `async with`. Tienen los mismos campos que *For* y *With*, respectivamente. Solo válido en el cuerpo de un *AsyncFunctionDef*.

Nota: Cuando *ast.parse()* analiza una cadena, los nodos de operador (subclases de `ast.operator`, `ast.unaryop`, `ast.cmpop`, `ast.boolop` y `ast.expr_context`) en el árbol retornado serán singletons. Los cambios en uno se reflejarán en todas las demás ocurrencias del mismo valor (por ejemplo, *ast.Add*).

32.1.3 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 alto es `sys.version_info[0:2]`.

Si la fuente contiene un carácter nulo ("0"), se lanza *ValueError*.

Advertencia: Tenga en cuenta que analizar correctamente el código fuente en un objeto AST no garantiza que el código fuente proporcionado sea un código Python válido que se pueda ejecutar, ya que el paso de compilación puede lanzar más excepciones *SyntaxError*. Por ejemplo, la fuente `return 42` genera un nodo AST válido para una declaración de retorno, pero no se puede compilar solo (debe estar dentro de un nodo de función).

En particular, *ast.parse()* no realizará ninguna verificación de alcance, lo que hace el paso de compilación.

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.`**`unparse`**(*ast_obj*)

Analice un objeto *ast.AST* y genere una cadena con código que produciría un objeto *ast.AST* equivalente si se analiza con *ast.parse()*.

Advertencia: La cadena de código producida no será necesariamente igual al código original que generó el objeto *ast.AST* (sin ninguna optimización del compilador, como tuplas constantes / frozensets).

Advertencia: Intentar descomprimir una expresión muy compleja daría como resultado *RecursionError*.

Nuevo en la versión 3.9.

`ast.`**`literal_eval`**(*node_or_string*)

Evaluate an expression node or a string containing only a Python literal or container display. The string or node provided may only consist of the following Python literal structures: strings, bytes, numbers, tuples, lists, dicts, sets, booleans, `None` and `Ellipsis`.

This can be used for evaluating strings containing Python values without the need to parse the values oneself. It is not capable of evaluating arbitrarily complex expressions, for example involving operators or indexing.

This function had been documented as «safe» in the past without defining what that meant. That was misleading. This is specifically designed not to execute Python code, unlike the more general *eval()*. There is no namespace, no name lookups, or ability to call out. But it is not free from attack: A relatively small input can lead to memory exhaustion or to C stack exhaustion, crashing the process. There is also the possibility for excessive CPU consumption denial of service on some inputs. Calling it on untrusted data is thus not recommended.

Advertencia: It is possible to crash the Python interpreter due to stack depth limitations in Python's AST compiler.

Puede generar *ValueError*, *TypeError*, *SyntaxError*, *MemoryError* y *RecursionError* dependiendo de la entrada mal formada.

Distinto en la versión 3.2: Ahora permite bytes y establece literales.

Distinto en la versión 3.9: Ahora admite la creación de conjuntos vacíos con `'set()'`.

Distinto en la versión 3.10: Para las entradas de cadena, los espacios iniciales y las tabulaciones ahora se eliminan.

`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*)
Incremente 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=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*, ***, *indent=None*)
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.

Si *indent* es un entero no negativo o una cadena de caracteres, entonces el árbol será impreso de forma linda con ese nivel de sangría. Un nivel de sangría de 0, negativo, o `""` solo insertará nuevas líneas. `None` (el valor por defecto) selecciona la representación de línea simple. Al usar un entero positivo se sangrará esa cantidad de espacios como sangría. Si *indent* es una cadena de caracteres (como `"\t"`), esa cadena se usa para sangrar cada nivel.

Distinto en la versión 3.9: Añadida la opción *indent*.

32.1.4 Banderas del compilador

Los siguientes indicadores pueden pasarse a *compile()* para cambiar los efectos en la compilación de un programa:

ast.**PyCF_ALLOW_TOP_LEVEL_AWAIT**
Habilita el soporte para `await`, `async for`, `async with` y comprensiones asíncronas de nivel superior.

Nuevo en la versión 3.8.

ast.**PyCF_ONLY_AST**
Genera y retorna un árbol de sintaxis abstracto en lugar de retornar un objeto de código compilado.

ast.**PyCF_TYPE_COMMENTS**
Habilita el soporte para comentarios de tipo de estilo **PEP 484** y **PEP 526** (`# type: <type>`, `# type: ignore <stuff>`).

Nuevo en la versión 3.8.

32.1.5 Uso en línea de comandos

Nuevo en la versión 3.9.

El módulo *ast* puede ser ejecutado como un script desde la línea de comandos. Es tan simple como:

```
python -m ast [-m <mode>] [-a] [infile]
```

Las siguientes opciones son aceptadas:

-h, --help
Muestra el mensaje de ayuda y sale.

-m <mode>
--mode <mode>
Especifica qué tipo de código debe ser compilado, como el argumento *mode* en *parse()*.

--no-type-comments
No analizar los comentarios de tipo.

-a, --include-attributes
Incluye atributos como números de línea y sangrías.

-i <indent>
--indent <indent>
Sangría de nodos en AST (número de espacios).

Si `infile` es especificado, su contenido es analizado a AST y mostrado en stdout. De otra forma, el contenido es leído desde stdin.

Ver también:

Green Tree Snakes, un recurso de documentación externo, tiene buenos detalles sobre cómo trabajar con Python AST.

ASTTokens anota ASTs de Python con la posición de tokens y texto en el código fuente que los genera. Esto es de ayuda para herramientas que hacen transformaciones de código fuente.

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 analiza código como Árboles de Sintaxis Concreta que se ven como ASTs y mantienen todos los detalles de formato. Es útil para construir herramientas de refactor automáticas y linters.

Parso es un analizador de Python que soporta recuperación de errores y análisis sintáctico de ida y vuelta para las diferentes versiones de Python (en múltiples versiones de Python). Parso también es capaz de enlistar múltiples errores de sintaxis en tu archivo de Python.

32.2 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.

32.2.1 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()`.

32.2.2 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()`.

`get_identifiers`()
Return a view object containing the names of symbols in the table. See the *documentation of view objects*.

`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()
Retorna `True` si el símbolo está anotado.

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*.

32.3 token— Constantes usadas con árboles de sintaxis de Python

Código fuente: Lib/token.py

Este módulo proporciona constantes que representan los valores numéricos de los nodos hoja del árbol de análisis (tokens terminales). Consulte el archivo `Grammar/Tokens` en la distribución de Python para conocer las definiciones de los nombres en el contexto de la gramática del lenguaje. Los valores numéricos específicos a los que se asignan los nombres pueden cambiar entre las 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 de token para `"("`.

token.**RPAR**
Valor de token para `")"`.

token.**LSQB**
Valor de token para `"["`.

token.**RSQB**
Valor de token para `"]"`.

token.**COLON**
Valor de token para ":".

token.**COMMA**
Valor de token para ",".

token.**SEMI**
Valor de token para ";".

token.**PLUS**
Valor de token para "+".

token.**MINUS**
Valor de token para "-".

token.**STAR**
Valor de token para "*".

token.**SLASH**
Valor de token para "/".

token.**VBAR**
Valor de token para "|".

token.**AMPER**
Valor de token para "&".

token.**LESS**
Valor de token para "<".

token.**GREATER**
Valor de token para ">".

token.**EQUAL**
Valor de token para "=".

token.**DOT**
Valor de token para ".".

token.**PERCENT**
Valor de token para "%".

token.**LBRACE**
Valor de token para "{".

token.**RBRACE**
Valor de token para "}".

token.**EQEQUAL**
Valor de token para "==".

token.**NOTEQUAL**
Valor de token para "!=".

token.**LESSEQUAL**
Valor de token para "<=".

token.**GREATEREQUAL**
Valor de token para ">=".

token.**TILDE**
Valor de token para "~".

token.**CIRCUMFLEX**
Valor de token para "^".

token.**LEFTSHIFT**
Valor de token para "<<".

token.**RIGHTSHIFT**
Valor de token para ">>".

token.**DOUBLESTAR**
Valor de token para "**".

token.**PLUSEQUAL**
Valor de token para "+=".

token.**MINEQUAL**
Valor de token para "-=".

token.**STAREQUAL**
Valor de token para "*=".

token.**SLASHEQUAL**
Valor de token para "/=".

token.**PERCENTEQUAL**
Valor de token para "%=".

token.**AMPEREQUAL**
Valor de token para "&=".

token.**VBAREQUAL**
Valor de token para "|=".

token.**CIRCUMFLEXEQUAL**
Valor de token para "^=".

token.**LEFTSHIFTEQUAL**
Valor de token para "<<=".

token.**RIGHTSHIFTEQUAL**
Valor de token para ">>=".

token.**DOUBLESTAREQUAL**
Valor de token para "**=".

token.**DOUBLESLASH**
Valor de token para "//".

token.**DOUBLESLASHEQUAL**
Valor de token para "//=".

token.**AT**
Valor de token para "@".

token.**ATEQUAL**
Valor de token para "@=".

token.**RARROW**
Valor de token para "->".

token.**ELLIPSIS**
Valor de token para "...".

token.**COLONEQUAL**
Valor de token para ":=".

token.**OP**

token.**AWAIT**

token.**ASYNC**

token.**TYPE_IGNORE**

token.**TYPE_COMMENT**

token.**SOFT_KEYWORD**

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).

32.4 keyword — Pruebas para palabras clave en Python

Código fuente: Lib/keyword.py

This module allows a Python program to determine if a string is a keyword or soft keyword.

keyword.**iskeyword**(*s*)
Retorna True si *s* es una palabra clave Python.

keyword.**kwlist**
Secuencia que contiene todos las palabras clave definidos para el intérprete. Si cualquier palabra clave es definida para estar activa sólo cuando las declaraciones particulares *__future__* están vigentes, estas se incluirán también.

keyword.**issoftkeyword**(*s*)
Return True if *s* is a Python soft keyword.

Nuevo en la versión 3.9.

keyword.**softkwlist**
Sequence containing all the soft keywords defined for the interpreter. If any soft keywords are defined to only be active when particular *__future__* statements are in effect, these will be included as well.

Nuevo en la versión 3.9.

32.5 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()*.

Advertencia: Note that the functions in this module are only designed to parse syntactically valid Python code (code that does not raise when parsed using *ast.parse()*). The behavior of the functions in this module is **undefined** when providing invalid Python code and it can change at any point.

32.5.1 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()*.

Llamará 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.

32.5.2 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.

32.5.3 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             '('
```

(continué en la próxima página)

(proviene de la página anterior)

```
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:
        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)
```

32.6 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()* sí 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.

32.7 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.

32.7.1 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 el en 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.

Function.**is_async**
`True` para funciones que están definidas con el prefijo `async`, de otra manera `False`.

Nuevo en la versión 3.10.

32.7.2 Objetos Class

Las instancias de las 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 el en 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.

32.8 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=PycInvalidationMode.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 instead of *file* as the name of the source file from which source lines are obtained for display in exception tracebacks. 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 sobreescribe 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.

32.8.1 Interfaz de línea de comandos

Este módulo puede invocarse como un script para compilar varios archivos fuente. Los archivos nombrados en *filenames* se compilan y el código de bytes resultante se almacena en caché de la manera normal. Este programa no busca en una estructura de directorio para localizar archivos fuente; solo compila archivos nombrados explícitamente. El estado de salida es distinto de cero si uno de los archivos no se pudo compilar.

<file> ... <fileN>

-

Los argumentos posicionales son archivos para compilar. Si − es el único parámetro, la lista de archivos se toma de la entrada estándar.

-q, --quiet

Suprime la salida de errores.

Distinto en la versión 3.2: Agregado soporte para −.

Distinto en la versión 3.10: Agregado soporte para *-q*.

Ver también:

Módulo *compileall* Utilidades para compilar todos los archivos fuente Python en el árbol del directorio.

32.9 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.

32.9.1 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.

-s `strip_prefix`

-p `prepend_prefix`
: Elimina (`-s`) o agrega (`-p`) el prefijo dado de rutas registradas en los archivos `.pyc`. No se puede combinar con `-d`.

-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 sobreescribir 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.

-o level
: Compila con el nivel de optimización dado. Puede usarse varias veces para compilar varios niveles a la vez (por ejemplo, compileall -o 1 -o 2).

-e dir
: Ignora los enlaces simbólicos que apuntan fuera del directorio dado.

--hardlink-dupes
: Si dos archivos .pyc con diferente nivel de optimización tienen el mismo contenido, use enlaces físicos para consolidar los archivos duplicados.

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.

Distinto en la versión 3.9: Se agregaron las opciones -s, -p, -e y --hardlink-dupes. Se elevó el límite de recursividad predeterminado de 10 a *sys.getrecursionlimit()*. Se agregó la posibilidad de especificar la opción -o varias veces.

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.

32.9.2 Funciones públicas

compileall.**compile_dir**(*dir*, *maxlevels=sys.getrecursionlimit()*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=- 1*, *workers=1*, *invalidation_mode=None*, ***, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)
: 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 sys.getrecursionlimit().

  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 se proporciona *rx*, se llama a su método `search` en la ruta completa a cada archivo considerado para compilación, y si retorna un valor verdadero, se omite el archivo. Esto se puede usar para excluir archivos que coincidan con una expresión regular, dada como un objeto *re.Pattern*.

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 sobreescribir 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()*. Acepta también una secuencia de niveles de optimización que conducen a múltiples compilaciones de un archivo `.py` en una llamada.

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.

The *stripdir*, *prependdir* and *limit_sl_dest* arguments correspond to the `-s`, `-p` and `-e` options described above. They may be specified as `str` or *os.PathLike*.

Si *hardlink_dupes* es verdadero y dos archivos `.pyc` con diferente nivel de optimización tienen el mismo contenido, use enlaces físicos para consolidar los archivos duplicados.

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.

Distinto en la versión 3.9: Se agregaron los argumentos *stripdir*, *prependdir*, *limit_sl_dest* y *hardlink_dupes*. El valor predeterminado de *maxlevels* se cambió de `10` a `sys.getrecursionlimit()`

`compileall.`**`compile_file`**(*fullname*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*, ***, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)

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 se proporciona *rx*, a su método `search` se le pasa el nombre de ruta completo al archivo que se está compilando, y si retorna un valor verdadero, el archivo no se compila y se retorna `True`. Esto se puede usar para excluir archivos que coincidan con una expresión regular, dada como un objeto *re.Pattern*.

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 sobreescribir 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()*. Acepta también una secuencia de niveles de optimización que conducen a múltiples compilaciones de un archivo `.py` en una llamada.

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.

The *stripdir*, *prependdir* and *limit_sl_dest* arguments correspond to the `-s`, `-p` and `-e` options described above. They may be specified as `str` or *os.PathLike*.

Si *hardlink_dupes* es verdadero y dos archivos `.pyc` con diferente nivel de optimización tienen el mismo contenido, use enlaces físicos para consolidar los archivos duplicados.

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*.

Distinto en la versión 3.9: Se agregaron los argumentos *stripdir*, *prependdir*, *limit_sl_dest* y *hardlink_dupes*.

`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_curdir* 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 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*.

Para forzar la re-compilación del 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:

Módulo *py_compile* Compilación byte de un solo archivo fuente.

32.10 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.

Detalles de implementación de CPython: 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.

Distinto en la versión 3.10: The argument of jump, exception handling and loop instructions is now the instruction offset rather than the byte offset.

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).

32.10.1 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
```

32.10.2 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*)
This generator function uses the `co_lines` method of the code object *code* to find the offsets which are starts of lines in the source code. They are generated as `(offset, lineno)` pairs.

Distinto en la versión 3.6: Los números de línea pueden estar disminuyendo. Antes, siempre estaban aumentando.

Distinto en la versión 3.10: The **PEP 626** `co_lines` method is used instead of the `co_firstlineno` and `co_lnotab` attributes of the code object.

`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*.

32.10.3 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
Eleva los elementos de la segunda, tercera y cuarta pila una posición hacia arriba, se mueve de arriba hacia abajo a la posición cuatro.

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
Pushes `get_awaitable(TOS.__anext__())` to the stack. See `GET_AWAITABLE` for details about `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*)
Llama a `list.append(TOS1[-i], TOS)`. Se utiliza para implementar listas por comprensión.

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.

RERAISE
Re-lanza la excepción que se encuentra actualmente al tope de la pila. Si oparg no es cero, restaura `f_lasti` del marco actual a su valor cuando se lanza la excepción.

Nuevo en la versión 3.9.

WITH_EXCEPT_START
Llama a la función en la posición 7 de la pila con los tres elementos superiores de la pila como argumentos. Se usa para implementar la llamada `context_manager.__exit__(*exc_info())` cuando se ha producido una excepción en una sentencia `with`.

Nuevo en la versión 3.9.

LOAD_ASSERTION_ERROR
Inserta *AssertionError* en la pila. Utilizado por la declaración `assert`.

Nuevo en la versión 3.9.

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_EXCEPT_START*. Luego, `__enter__()` se llama, y un bloque finally que apunta a *delta* se apila. 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.

COPY_DICT_WITHOUT_KEYS
TOS es una tuple de llaves de mapeo, y TOS1 es el sujeto de la coincidencia. Reemplaza TOS con un *dict* formado por los ítems de TOS1, pero sin ninguna de las llaves en TOS.

Nuevo en la versión 3.10.

GET_LEN
Apila `len(TOS)` en la pila.

Nuevo en la versión 3.10.

MATCH_MAPPING
Si TOS es una instancia de *collections.abc.Mapping* (o, más técnicamente, si tiene el indicador `Py_TPFLAGS_MAPPING` establecido en su `tp_flags`), apila `True` en la pila. De lo contrario apila `False`.

Nuevo en la versión 3.10.

MATCH_SEQUENCE
Si TOS es una instancia de *collections.abc.Sequence* y *no* es una instancia de *str*/*bytes*/*bytearray* (o, más técnicamente, si tiene el indicador `Py_TPFLAGS_SEQUENCE` establecido en su `tp_flags`), apila `True` en la pila. De lo contrario apila `False`.

Nuevo en la versión 3.10.

MATCH_KEYS
TOS es una tuple de llaves de mapeo, y TOS1 es el sujeto de la coincidencia. Si TOS1 contiene todas las llaves en TOS, apila un *tuple* que contiene los valores correspondientes, seguido por `True`. De lo contrario apila `None`, seguido de `False`.

Nuevo en la versión 3.10.

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.

LIST_TO_TUPLE
Saca una lista de la pila y empuja una tupla que contiene los mismos valores.

Nuevo en la versión 3.9.

LIST_EXTEND (*i*)
Llama a `list.extend(TOS1[-i], TOS)`. Se utiliza para crear listas.

Nuevo en la versión 3.9.

SET_UPDATE (*i*)
Llama a `set.update(TOS1[-i], TOS)`. Se usa para construir decorados.

Nuevo en la versión 3.9.

DICT_UPDATE (*i*)
Llama a `dict.update(TOS1[-i], TOS)`. Se usa para construir dictados.

Nuevo en la versión 3.9.

DICT_MERGE
Como *DICT_UPDATE* pero lanza una excepción para claves duplicadas.

Nuevo en la versión 3.9.

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]`.

IS_OP (*invert*)
Realiza una comparación `is` o `is not` si `invert` es 1.

Nuevo en la versión 3.9.

CONTAINS_OP (*invert*)
Realiza una comparación `in` o `not in` si `invert` es 1.

Nuevo en la versión 3.9.

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_NOT_EXC_MATCH (*target*)
Comprueba si el segundo valor de la pila es una excepción que coincide con el TOS y salta si no lo es. Saca dos valores de la pila.

Nuevo en la versión 3.9.

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.

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 de la pila contiene una tupla con los nombres de los argumentos de la palabra clave, que deben ser cadenas de caracteres. Debajo están los valores para los argumentos de la palabra clave, en el orden correspondiente a la tupla. Debajo están los 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. Antes de que se llame al invocable, el objeto de mapeo y el objeto iterable se «desempaquetan» y su contenido se pasa como palabra 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 devuelto 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` una tupla de cadenas de caracteres que contiene anotaciones de parámetros
- `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)

Distinto en la versión 3.10: El valor indicador `0x04` es una tupla de cadena de caracteres en vez de un diccionario

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 predeterminado. *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.

MATCH_CLASS (*count*)
TOS es una tupla de nombres de atributos clave, TOS1 es la clase contra la cual se hace la coincidencia, y TOS2 es el sujeto de la coincidencia. *count* es el número de sub-patrones posicionales.

Desapila TOS. Si TOS2 es una instancia de TOS1 y tiene los atributos clave y posicionales requeridos por *count* y TOS, establece TOS como `True` y TOS1 es una tupla de atributos extraídos. De lo contrario, establece TOS como `False`.

Nuevo en la versión 3.10.

GEN_START (*kind*)

Pops TOS. The `kind` operand corresponds to the type of generator or coroutine. The legal kinds are 0 for generator, 1 for coroutine, and 2 for async generator.

Nuevo en la versión 3.10.

ROT_N (*count*)
Eleva los *count* elementos más altos de la pila una posición hacia arriba, y baja TOS hacia la posición *count*.

Nuevo en la versión 3.10.

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.

32.10.4 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.

32.11 `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`*.

32.11.1 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.

32.11.2 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.

CAPÍTULO 33

Servicios Específicos para MS Windows

Este capítulo describe los módulos que sólo están disponibles en plataformas MS Windows.

33.1 `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*.

33.1.1 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_NBRLCK**
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_osfhandle**(*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_osfhandle` con los argumentos `handle`, `flags`.

msvcrt.**get_osfhandle**(*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_osfhandle` con el argumento `fd`.

33.1.2 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 `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.

33.1.3 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*.

33.2 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*.

33.2.1 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.

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 is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_WOW64_64KEY`. On 32-bit Windows, the WOW64 constants are ignored. See *Access Rights* for other allowed values.

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 *`SetValueEx()`*)

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 clave 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`.

33.2.2 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**
Indicates that an application on 64-bit Windows should operate on the 64-bit registry view. On 32-bit Windows, this constant is ignored.

winreg.**KEY_WOW64_32KEY**
Indicates that an application on 64-bit Windows should operate on the 32-bit registry view. On 32-bit Windows, this constant is ignored.

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.

33.2.3 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.

33.3 :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:

PlaySound() *name*	Panel de control correspondiente nombre (*name*) 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`.

CAPÍTULO 34

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:

34.1 `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*.

34.1.1 Soporte de archivos grandes

Several operating systems (including AIX and Solaris) provide support for files that are larger than 2 GiB from a C programming model where `int` and `long` are 32-bit values. This is typically accomplished by defining the relevant size and offset types as 64-bit values. Such files are sometimes referred to as *large files*.

Large file support is enabled in Python when the size of an `off_t` is larger than a `long` and the `long long` is at least as large as an `off_t`. It may be necessary to configure and compile Python with certain compiler flags to enable this mode. For example, with Solaris 2.6 and 2.7 you need to do something like:

```
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
```

34.1.2 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*.

34.2 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	`pw_name`	Nombre de usuario
1	`pw_passwd`	Contraseña encriptada opcional
2	`pw_uid`	Identificación numérica de usuario
3	`pw_gid`	Identificación del grupo numérico
4	`pw_gecos`	Nombre de usuario o campo de comentarios
5	`pw_dir`	El directorio *home* del usuario
6	`pw_shell`	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: En Unix tradicional, el campo `pw_passwd` generalmente contiene una contraseña cifrada con un algoritmo derivado de DES (ver módulo *crypt*). Sin embargo, la mayoría de los sistemas operativos modernos utilizan un sistema llamado *shadow password*. En esos unices, el campo *pw_passwd* solo contiene un asterisco (`'*'`) o la letra `'x'` donde la contraseña cifrada se almacena en un archivo `/etc/shadow` que no es legible por todo el mundo. Si el campo *pw_passwd* contiene algo útil depende del sistema. Si está disponible, el módulo *spwd* debe usarse donde se requiera acceso a la contraseña encriptada.

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*** Una interfaz para la base de datos de contraseñas ocultas, similar a esta.

34.3 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.

Group database entries are reported as a tuple-like object, whose attributes correspond to the members of the group structure (Attribute field below, see <grp.h>):

Índice	Atributo	Significado
0	gr_name	el nombre del grupo
1	gr_passwd	El grupo contraseña (encriptado); usualmente vacío
2	gr_gid	el grupo ID numérico
3	gr_mem	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**(*id*)
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.

Distinto en la versión 3.10: *TypeError* is raised for non-integer arguments like floats or strings.

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.

34.4 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.

34.4.1 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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
```

34.5 `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.

34.6 `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.

El manejo de pseudo-terminales depende en gran medida de la plataforma. Este código se prueba principalmente en Linux, FreeBSD y macOS (se supone que funciona en otras plataformas POSIX, pero no se ha probado a fondo).

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.

Un bucle copia STDIN del proceso actual al hijo y los datos recibidos del niño a STDOUT del proceso actual. No se le indica al hijo si STDIN del proceso actual se cierra.

A las funciones *master_read* y *stdin_read* se les pasa un descriptor de archivo del cual deben leer, y siempre deben retornar una cadena de bytes. Para forzar el retorno de spawn antes de que el proceso hijo salga, se debe retornar un arreglo de bytes vacía para señalar el final del archivo.

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)

Retorna el valor del estado de salida de *os.waitpid()* en el proceso hijo.

`waitstatus_to_exitcode()` se puede utilizar para convertir el estado de salida en un código de salida.

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.

34.6.1 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```

34.7 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()*.

Distinto en la versión 3.9: On macOS, the fcntl module exposes the `F_GETPATH` constant, which obtains the path of a file from a file descriptor. On Linux(>=3.15), the fcntl module exposes the `F_OFD_GETLK`, `F_OFD_SETLK` and `F_OFD_SETLKW` constants, which are used when working with open file description locks.

Distinto en la versión 3.10: En Linux >= 2.6.11, el módulo fcntl expone las constantes `F_GETPIPE_SZ` y `F_SETPIPE_SZ`, las cuales permiten chequear y modificar el tamaño de un *pipe*, respectivamente.

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`**(*fd*, *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`**(*fd*, *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 lanzará 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('hhllhh', 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()`.

34.8 `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`.

34.8.1 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 lanzará 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_RTTIME**
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**
El tamaño máximo (en bytes) del espacio de intercambio que puede ser reservado o utilizado por todos los procesos de este ID de usuario. Este límite se aplica solo si se establece el bit 1 de vm.overcommit sysctl. Consulte tuning(7) para obtener una descripción completa de este 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.

resource.**RLIMIT_KQUEUES**
El número máximo de kqueues que este ID de usuario puede crear.

Disponibilidad: FreeBSD 11 o posterior.

Nuevo en la versión 3.10.

34.8.2 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
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	`ru_utime`	tiempo en modo usuario (flotante en segundos)
1	`ru_stime`	tiempo en modo sistema (flotante en segundos)
2	`ru_maxrss`	tamaño máximo del conjunto residente
3	`ru_ixrss`	tamaño de memoria compartida
4	`ru_idrss`	tamaño de memoria no compartida
5	`ru_isrss`	tamaño de la pila no compartida
6	`ru_minflt`	fallos de página que no requieran E/S
7	`ru_majflt`	fallos de página que requieran E/S
8	`ru_nswap`	número de intercambios
9	`ru_inblock`	bloque de operaciones de entrada
10	`ru_oublock`	bloque de operaciones de salida
11	`ru_msgsnd`	mensajes enviados
12	`ru_msgrcv`	mensajes recibidos
13	`ru_nsignals`	señales recibidas
14	`ru_nvcsw`	intercambios de contexto voluntarios
15	`ru_nivcsw`	intercambios de contexto involuntarios

Esta función lanzará 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.

34.9 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()*.

  If *openlog()* has not been called prior to the call to *syslog()*, *openlog()* will be called with no arguments.

  Lanza un *evento de auditoría* syslog.syslog con los argumentos priority, message.

  Distinto en la versión 3.2: In previous versions, *openlog()* would not be called automatically if it wasn't called prior to the call to *syslog()*, deferring to the syslog implementation to call openlog().

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: In previous versions, keyword arguments were not allowed, and *ident* was required.

`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`.

34.9.1 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...')
```

CAPÍTULO 35

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.

35.1 `aifc` — Lee y escribe archivos AIFF y AIFC

Código fuente: Lib/aifc.py

Obsoleto desde la versión 3.11: The *aifc* module is deprecated (see **PEP 594** for details).

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.

35.2 `asynchat` — Gestor de comandos/respuestas en *sockets* asíncronos

Código fuente: Lib/asynchat.py

Obsoleto desde la versión 3.6: *asynchat* will be removed in Python 3.12 (see **PEP 594** for details). Please use *asyncio* instead.

Nota: Este módulo existe únicamente por motivos de retrocompatibilidad. Para nuevo código, es recomendable usar *asyncio*.

Este módulo se construye en la infraestructura de *asyncore*, simplificando los clientes y servidores asíncronos y facilitando la gestión de protocolos cuyos elementos son terminados por cadenas de texto arbitrarias, o que son

de longitud variable. *asynchat* define la clase abstracta *async_chat* de la que se debe heredar, implementado los métodos `collect_incoming_data()` y `found_terminator()`. Utiliza el mismo bucle asíncrono que *asyncore*, y los dos tipos de canal, *asyncore.dispatcher* y *asynchat.async_chat*, se pueden mezclar libremente en el mapa de canal. Normalmente un canal de servidor *asyncore.dispatcher* genera nuevos objetos de canal *asynchat.async_chat*, al recibir peticiones de conexión entrantes.

class asynchat.**async_chat**

Esta clase es una subclase abstracta de *asyncore.dispatcher*. Para el uso práctico del código se debe heredar *async_chat*, definiendo los métodos significativos *collect_incoming_data()* y *found_terminator()*. Los métodos de *asyncore.dispatcher* se pueden utilizar, aunque no todos tienen sentido en un contexto de mensaje/respuesta.

Al igual que *asyncore.dispatcher*, *async_chat* define una serie de eventos generados por un análisis sobre las condiciones de _socket_, tras una llamada a `select()`. Una vez que el bucle de _polling_ haya sido iniciado, los métodos de los objetos *async_chat* son llamados por el _framework_ que procesa los eventos, sin que tengamos que programar ninguna acción a mayores.

Se pueden modificar dos atributos de clase, para mejorar el rendimiento o incluso hasta para ahorrar memoria.

ac_in_buffer_size

El tamaño del _buffer_ de entrada asíncrona (por defecto `4096`).

ac_out_buffer_size

El tamaño del _buffer_ de salida asíncrona (por defecto `4096`).

Al contrario que *asyncore.dispatcher*, *async_chat* permite definir una cola FIFO de productores (*producers*). Un productor necesita tener un solo método, `more()`, que debe devolver los datos que se vayan a transmitir en el canal. Cuando el método `more()` devuelve un objeto bytes vacío, significa que el productor ya se ha agotado (por ejemplo, que no contiene más datos). En este punto, el objeto *async_chat* elimina el productor de la cola y empieza a usar el siguiente productor, si existiese. Cuando la cola de productores está vacía, el método `handle_write()` no hace nada. El método *set_terminator()* de los objetos de canal se utiliza para describir cómo reconocer, en una transmisión entrante desde el punto remoto, el final de esta transmisión o un punto de ruptura importante en la misma.

Para construir una subclase funcional de *async_chat*, los métodos de entrada *collect_incoming_data()* and *found_terminator()* deben tratar los datos que el canal recibe asíncronamente. Los métodos se describen a continuación.

async_chat.**close_when_done**()

Añade un `None` en la cola de productores. Cuando este productor se extrae de la cola, hace que el canal se cierre.

async_chat.**collect_incoming_data**(*data*)

Llamado con *data*, conteniendo una cantidad arbitraria de datos recibidos. El método por defecto, que debe ser reemplazado, lanza una excepción *NotImplementedError*.

async_chat.**discard_buffers**()

En situaciones de emergencia, este método descarta cualquier dato albergado en los búfers de entrada y/o salida y en la cola del productor.

async_chat.**found_terminator**()

Llamado cuando el flujo de datos de entrada coincide con la condición de finalización establecida por *set_terminator()*. El método por defecto, que debe ser reemplazado, lanza una excepción *NotImplementedError*. Los datos de entrada en búfer deberían estar disponibles a través de un atributo de instancia.

async_chat.**get_terminator**()

Retorna el terminador actual del canal.

async_chat.**push**(*data*)

Añade datos en la cola del canal para asegurarse de su transmisión. Esto es todo lo que se necesita hacer para que el canal envíe los datos a la red, aunque es posible usar productores personalizados en esquemas más complejos para implementar características como encriptación o fragmentación.

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*)
Establece la condición de finalización que será reconocida en este canal. `term` puede ser uno de los tres tipos de valores posibles, correspondientes a tres formas diferentes de tratar los datos de protocolo entrantes.

tér-mino	Descripción
string	Llamará a *found_terminator()* cuando la cadena de caracteres se encuentre en el flujo de datos de entrada
integer	Llamará a *found_terminator()* cuando el número de caracteres indicado se haya recibido
`None`	El canal continúa recopilando datos indefinidamente

Téngase en cuenta que cualquier dato posterior al terminador estará disponible para ser leído por el canal después de llamar a *found_terminator()*.

35.2.1 Ejemplo de asynchat

El siguiente ejemplo parcial muestra cómo se pueden leer peticiones HTTP con *async_chat*. Un servidor web podría crear un objeto `http_request_handler` para cada conexión de cliente entrante. Téngase en cuenta que, inicialmente, el terminador del canal está configurado para detectar la línea vacía presente al final de las cabeceras HTTP, y una bandera indica que las cabeceras se están leyendo.

Una vez que las cabeceras se hayan leído, si la petición es de tipo POST (lo cual indica que hay más datos disponibles en el flujo de entrada), la cabecera `Content-Length:` se utiliza para establecer un terminador numérico para leer la cantidad de datos correcta en el canal.

El método `handle_request()` se llama una vez todas las entradas relevantes han sido serializadas (*marshalled*), tras establecer el terminador del canal a `None` para asegurarse de que cualquier dato extraño enviado por el cliente web es ignorado.

```
import asynchat

class http_request_handler(asynchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = 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":
```

(continué en la próxima página)

(proviene de la página anterior)

```
                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()
```

35.3 `asyncore` — controlador de socket asincrónico

Código fuente: Lib/asyncore.py

Obsoleto desde la versión 3.6: *asyncore* will be removed in Python 3.12 (see **PEP 594** for details). Please use *asyncio* instead.

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, *asynchat.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
`handle_connect()`	Implícito en el primer proceso de lectura o escritura
`handle_close()`	Implícito en un evento de lectura sin datos disponibles
`handle_accepted()`	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 asincrónico 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 asincrónico 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 lanzar *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 *asynchat.async_chat*.

class asyncore.**file_dispatcher**
Un 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**
Un 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.

35.3.1 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()
```

35.3.2 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()
```

35.4 audioop — Manipula datos de audio sin procesar

Obsoleto desde la versión 3.11: The *audioop* module is deprecated (see **PEP 594** for details).

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 mínimo, 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 máximo. 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 es una tupla que contiene el estado del codificador. El codificador retorna una tupla `(adpcmfrag, newstate)`, y el *newstate* debe pasarse a la siguiente llamada de *lin2adpcm()*. En la llamada inicial, se puede pasar `None` como estado. *adpcmfrag* es el fragmento codificado ADPCM empaquetado 2 valores de 4 bits por 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)
    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.

La 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)
```

35.5 `cgi` — Soporte de Interfaz de Entrada Común (CGI)

Código fuente: Lib/cgi.py

Obsoleto desde la versión 3.11: The *cgi* module is deprecated (see **PEP 594** for details and alternatives).

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.

35.5.1 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!")
```

35.5.2 Usando el módulo CGI

Empieza escribiendo `import cgi`.

Cuando escribas un nuevo script, considera añadir estas líneas:

```
import cgitb
cgitb.enable()
```

Esto activa un gestor de excepciones especial que mostrará informes detallados en el explorador web si se produce algún error. Si prefiere no mostrar los detalles de su programa a los usuarios de su script, puede tener los informes guardados en archivos en su lugar, como se muestra en el siguiente código:

```
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 llegar a los datos de formulario enviados usa la clase `FieldStorage`. Si el formulario contiene caracteres que no sean ASCII, usa el parámetro de palabra clave *encoding* establecido en el valor del codificado definido para el documento. Éste está usualmente contenido en la etiqueta META en la sección HEAD del documento HTML, o en el encabezado *Content-Type*. Esto lee los contenidos del formulario desde la entrada estándar o el ambiente (dependiendo de los valores de varias variables de entorno establecidas de acuerdo del estándar CGI). Dado que podría consumir entrada estándar, sólo se debería instanciar una sola 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 real-

mente 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`.

35.5.3 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 `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

[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.

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)
```

35.5.4 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='&'*)
Analiza una consulta en el entorno o desde un archivo (el archivo predeterminado es `sys.stdin`). Los parámetros *keep_blank_values*, *strict_parsing* y *separator* se pasan a *`urllib.parse.parse_qs()`* sin cambios.

`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.

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 3.10: Agregado el parámetro *separator*.

`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 hacia el 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.

35.5.5 Preocuparse por la seguridad

Hay una regla importante: si invoca un programa externo (a través de *os.system()*, *os.popen()* u otras funciones con una funcionalidad similar), asegúrese bien de no pasar strings arbitrarios recibidos desde el cliente hacia la 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.

35.5.6 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 `0o755` (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 `0o644` para legible y `0o666` 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).

35.5.7 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.

35.5.8 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 instalando una copia de este archivo de módulo (`cgi.py`) como un script CGI. Cuando se invoca como un script, el archivo cambiará 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.

35.5.9 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

35.6 `cgitb` — Administrador *traceback* para scripts CGI.

Código fuente: Lib/cgitb.py

Obsoleto desde la versión 3.11: The *cgitb* module is deprecated (see **PEP 594** for details).

El modulo *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()*.

35.7 chunk — Lee los datos de los trozos de IFF

Código fuente: Lib/chunk.py

Obsoleto desde la versión 3.11: The *chunk* module is deprecated (see **PEP 594** for details).

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	*n*	Bytes de datos, donde *n* es el tamaño dado en el campo anterior
8 + *n*	0 o 1	Se necesita un byte de relleno si *n* 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

[1] «EA IFF 85» *Standard for Interchange Format Files*, *Jerry Morrison*, *Electronic Arts*, enero 1985.

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`.

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.

35.8 `crypt` — Función para verificar contraseñas Unix

Código fuente: Lib/crypt.py

Obsoleto desde la versión 3.11: The `crypt` module is deprecated (see **PEP 594** for details and alternatives). The `hashlib` module is a potential replacement for certain use cases.

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.

35.8.1 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.

35.8.2 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.

35.8.3 Funciones del módulo

El módulo *crypt* define las siguientes funciones:

crypt.**crypt**(*word*, *salt=None*)
: *word* will usually be a user's password as typed at a prompt or in a graphical interface. The optional *salt* is either a string as returned from *mksalt()*, one of the `crypt.METHOD_*` values (though not all may be available on all platforms), or a full encrypted password including salt, as returned by this function. If *salt* is not provided, the strongest method available in *methods* will be used.

  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*)
Return a randomly generated salt of the specified method. If no *method* is given, the strongest method available in *methods* is used.

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*.

35.8.4 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: ')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise ValueError('no support for shadow passwords')
        cleartext = getpass.getpass()
        return compare_hash(crypt.crypt(cleartext, cryptedpasswd), cryptedpasswd)
    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")
```

35.9 :mod:”imghdr” — Determinar el tipo de imagen

Código fuente: :source:”Lib/imghdr.py”

Obsoleto desde la versión 3.11: The *imghdr* module is deprecated (see **PEP 594** for details and alternatives).

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**(*file*, *h=None*)
Prueba los datos de imagen contenidos en el archivo nombrado por *file* y retorna una cadena que describe el tipo de imagen. Si se proporciona *h* opcional, se ignora el argumento *file* y se supone que *h* contiene el flujo de bytes para probar.

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'
```

35.10 `imp` — Acceda a import internamente

Código fuente: Lib/imp.py

Obsoleto desde la versión 3.4: El módulo `imp` está obsoleto en favor de `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 la cadena de caracteres mágica que se utiliza para reconocer archivos de código compilados por bytes (archivos `.pyc`). (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 cuales describe un tipo particular de módulo. Cada triple tiene la forma `(suffix, mode, type)`, donde *suffix* es una cadena de caracteres que se agregará al nombre del módulo para formar el nombre de archivo a buscar, *mode* es la cadena de caracteres 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* es el tipo de archivo, que tiene uno de los valores `PY_SOURCE`, `PY_COMPILED`, o `C_EXTENSION`, que se describen 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 *Ejemplos* 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 *Ejemplos* 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
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, como lo retorna *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 está actualmente retenido, de lo contrario, `False`. En plataformas sin hilos, 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 construidos 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.

35.10.1 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]
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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()
```

35.11 `mailcap` — Manejo de archivos Mailcap

Código fuente: Lib/mailcap.py

Obsoleto desde la versión 3.11: The *mailcap* module is deprecated (see **PEP 594** for details). The *mimetypes* module provides an alternative.

Los archivos Mailcap se utilizan para configurar cómo las aplicaciones compatibles con MIME, como los lectores de correo y los navegadores web, reaccionan a los archivos con diferentes tipos de MIME. (El nombre «mailcap» se deriva de la frase «capacidad de correo»). Por ejemplo, un archivo mailcap puede contener una línea como `video/mpeg; xmpeg %s`. Luego, si el usuario encuentra un mensaje de correo electrónico o documento web con el tipo MIME: mimetype: *video / mpeg*, `%s` será reemplazado por un nombre de archivo (generalmente uno que pertenezca a un archivo temporal) y el programa **xmpeg** se puede iniciar automáticamente para ver el archivo.

El formato mailcap está documentado en **RFC 1524**, «Un mecanismo de configuración de agente de usuario para información de formato de correo multimedia», pero no es un estándar de Internet. Sin embargo, los archivos mailcap son compatibles con 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 3.10.8: 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'})
```

35.12 `msilib` — Leer y escribir archivos *Microsoft Installer*

Código fuente: Lib/msilib/__init__.py

Obsoleto desde la versión 3.11: The *msilib* module is deprecated (see **PEP 594** for details).

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

35.12.1 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

35.12.2 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

35.12.3 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

35.12.4 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

35.12.5 Errores

Todos los *wrappers* sobre funciones MSI lanzan MSIError; la cadena de caracteres dentro de la excepción contendrá más detalles.

35.12.6 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 sobreescrito, 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.

35.12.7 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

35.12.8 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

35.12.9 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

35.12.10 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.

35.13 nis — Interfaz a Sun's NIS (Páginas amarillas)

Obsoleto desde la versión 3.11: The *nis* module is deprecated (see **PEP 594** for details).

El modulo *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.

35.14 nntplib — Protocolo de cliente NNTP

Código fuente: Lib/nntplib.py

Obsoleto desde la versión 3.11: The *nntplib* module is deprecated (see **PEP 594** for details).

Este módulo define la clase *NNTP* que implementa el lado del cliente del Protocolo de transferencia de noticias por red. Se puede utilizar para implementar un lector o póster de noticias, o procesadores de noticias automatizados. Es compatible con **RFC 3977** así como con los antiguos **RFC 977** y **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']))
...
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*, *usenetrc=False*[, *timeout*])

Retorna un nuevo objeto *NNTP* que representa una conexión con el servidor NNTP ejecutándose en el host *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 *usenetrc* 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: *usenetrc* es ahora `False` por defecto.

Distinto en la versión 3.3: El soporte para la declaración `with` fue añadido.

Distinto en la versión 3.9: Si el parámetro *timeout* se establece en cero, lanzará un *ValueError* para evitar la creación de un socket sin bloqueo.

class nntplib.**NNTP_SSL**(*host*, *port=563*, *user=None*, *password=None*, *ssl_context=None*, *readermode=None*, *usenetrc=False*[, *timeout*])

Retorna un nuevo objeto *NNTP_SSL*, que representa una conexión cifrada con el servidor NNTP ejecutándose en el host *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 *Consideraciones de seguridad* 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*).

Distinto en la versión 3.9: Si el parámetro *timeout* se establece en cero, lanzará un *ValueError* para evitar la creación de un socket sin bloqueo.

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.

35.14.1 Objetos NNTP

Cuando están conectados, los objetos *NNTP* y *NNTP_SSL* admiten los siguientes métodos y atributos.

Atributos

NNTP.**nntp_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.**nntp_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 *Consideraciones de seguridad* 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
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.

35.14.2 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'
```

35.15 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
```

Además, los usuarios pueden ejecutar uno de los siguientes:

```
<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]`).

35.15.1 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 `execl()` 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`

These option syntaxes are not supported by *optparse*, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you're exclusively targeting Windows or certain legacy platforms (e.g. VMS, MS-DOS).

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 si misma. *optparse* no te impide implementar opciones requeridas, pero tampoco brinda mucha ayuda para ello.

Por ejemplo, considera esta hipotética linea 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.

35.15.2 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 linea 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 *Extendiendo 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" store a constant value, pre-set via *Option.const*

"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.

- options that take a value indicate this fact in their automatically generated help message, e.g. for the «mode» option:

  ```
  -m MODE, --mode=MODE
  ```

  Here, «MODE» is called the meta-variable: it stands for the argument that the user is expected to supply to `-m`/`--mode`. By default, *optparse* converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that's not what you want—for example, the `--filename` option explicitly sets `metavar="FILE"`, resulting in this automatically generated option description:

```
-f FILE, --filename=FILE
```

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.")
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 subclasificar 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()
```

35.15.3 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 si. 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" store a constant value, pre-set via *Option.const*

"store_true" almacena `True`

"store_false" almacena `False`

"append" agrega el argumento de esta opción a una lista

"append_const" append a constant value to a list, pre-set via *Option.const*

"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
↪'])
>>> 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 linea 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 lanzará 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 linea 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 si. 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 linea 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 si:

```
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")
```

At this point, *optparse* detects that a previously added option is already using the `-n` option string. Since `conflict_handler` is `"resolve"`, it resolves the situation by removing `-n` from the earlier option's list of option strings. Now `--dry-run` is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

It's possible to whittle away the option strings for a previously added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, *optparse* removes that option completely, so it doesn't show up in help text or anywhere else. Carrying on with our existing OptionParser:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

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")
  ```

35.15.4 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 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 proporciones 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)
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
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)
```

35.15.5 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 al 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 si: 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.

35.16 `ossaudiodev` — Acceso a dispositivos de audio compatibles con OSS

Obsoleto desde la versión 3.11: The *`ossaudiodev`* module is deprecated (see **PEP 594** for details).

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.

35.16.1 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 `.au` y `/dev/audio`)
`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 `/dev/audio`
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 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"`.

35.16.2 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 lanzará 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.**reccontrols**()
: 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)
```

35.17 pipes — Interfaz para tuberías de shell

Código fuente: Lib/pipes.py

Obsoleto desde la versión 3.11: The *pipes* module is deprecated (see **PEP 594** for details). Please use the *subprocess* module instead.

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()*.

Disponibilidad: Unix. No disponible en VxWorks.

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'
```

35.17.1 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.

35.18 `smtpd` — Servidor SMTP

Source code: Lib/smtpd.py

Este módulo ofrece varias clases para implementar servidores SMTP (correo electrónico).

Obsoleto desde la versión 3.6: *smtpd* will be removed in Python 3.12 (see **PEP 594** for details). The aiosmtpd package is a recommended replacement for this module. It is based on *asyncio* and provides a more straightforward API.

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.

35.18.1 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 instanciación.

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 de argumentos de palabras clave.

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.

35.18.2 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.

35.18.3 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.

35.18.4 Objetos MailmanProxy

class smtpd.**MailmanProxy**(*localaddr*, *remoteaddr*)

Obsoleto desde la versión 3.9, se eliminará en la versión 3.11: *MailmanProxy* está obsoleto, ya que depende del módulo Mailman el cual ya no existe.

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.

35.18.5 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
Holds the fully qualified domain name of the server as returned by *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):

Comando	Acción tomada
HELO	Acepta el saludo del cliente y lo almacena en *seen_greeting*. Establece el servidor en el modo de comando base.
EHLO	Acepta el saludo del cliente y lo almacena en *seen_greeting*. 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 *mailfrom*. En el modo de comando extendido, acepta el atributo **RFC 1870** SIZE y responde apropiadamente según el valor de *data_size_limit*.
RCPT	Acepta la sintaxis «RCPT TO:» y almacena las direcciones proporcionadas en la lista *rcpttos*.
RSET	Restablece *mailfrom*, *rcpttos* y *received_data*, pero no el saludo.
DATA	Establece el estado interno en `DATA` y almacena las líneas restantes del cliente en *received_data* hasta que se recibe el terminador `"\r\n.\r\n"`.
HELP	Retorna información mínima sobre la sintaxis del comando
VRFY	Retorna el código 252 (el servidor no sabe si la dirección es válida)
EXPN	Informa que el comando no está implementado.

35.19 sndhdr — Determinar el tipo de archivo de sonido

Código fuente: Lib/sndhdr.py

Obsoleto desde la versión 3.11: The *sndhdr* module is deprecated (see **PEP 594** for details and alternatives).

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*.

35.20 `spwd` — La base de datos de contraseñas ocultas

Obsoleto desde la versión 3.11: The *spwd* module is deprecated (see **PEP 594** for details and alternatives).

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	`sp_namp`	Nombre de inicio de sesión
1	`sp_pwdp`	Contraseña encriptada
2	`sp_lstchg`	Fecha del último cambio
3	`sp_min`	Número mínimo de días entre cambios
4	`sp_max`	Número máximo de días entre cambios
5	`sp_warn`	Número de días antes de que expire la contraseña para advertir al usuario sobre ello
6	`sp_inact`	Número de días desde que caduca la contraseña hasta que se deshabilita la cuenta
7	`sp_expire`	Número de días desde 1970-01-01 cuando expira la cuenta
8	`sp_flag`	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.

35.21 sunau — Lectura y escritura de ficheros Sun AU

Código fuente: Lib/sunau.py

Obsoleto desde la versión 3.11: The *sunau* module is deprecated (see **PEP 594** for details).

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 `.snd`.
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`.

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.

35.21.1 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.

35.21.2 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()`.

35.22 `telnetlib` — cliente Telnet

Código fuente: Lib/telnetlib.py

Obsoleto desde la versión 3.11: The *telnetlib* module is deprecated (see **PEP 594** for details and alternatives).

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.

35.22.1 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 (suboption begin/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).

No intente volver a abrir una instancia ya conectada.

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 sustituyen 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.

35.22.2 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: ")
    tn.write(password.encode('ascii') + b"\n")

tn.write(b"ls\n")
tn.write(b"exit\n")

print(tn.read_all().decode('ascii'))
```

35.23 `uu` — Codifica y decodifica archivos UUEncode

Código fuente: Lib/uu.py

Obsoleto desde la versión 3.11: The *uu* module is deprecated (see **PEP 594** for details). *base64* is a modern alternative.

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 descriptas 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.

35.24 xdrlib — Codificar y decodificar datos XDR

Código fuente: Lib/xdrlib.py

Obsoleto desde la versión 3.11: The `xdrlib` module is deprecated (see **PEP 594** for details).

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.

35.24.1 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.

35.24.2 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 desempaquetada en el búfer de datos.

Unpacker.**set_position**(*position*)
Establece la posición de desempaquetado 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 desempaquetado como una cadena de caracteres.

Unpacker.**done**()
Indica se ha completado el proceso de desempaquetado. Lanza una excepción *Error* si no se han desempaquetado todos los datos.

Además, cada tipo de datos que puede ser empaquetados con un *Packer*, puede ser desempaquetado con un *Unpacker*. Los métodos de desempaquetados son de la forma unpack_tipo(), y no tienen argumentos. Estos retornan el objeto desempaquetado.

Unpacker.**unpack_float**()
Desempaqueta un número de punto flotante de precisión simple.

Unpacker.**unpack_double**()
Desempaqueta un número de punto flotante de doble precisión, similar a *unpack_float()*.

Adicionalmente, los siguientes métodos desempaquetan cadenas de caracteres, bytes y datos opacos:

Unpacker.**unpack_fstring**(*n*)
Desempaqueta 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*)
Desempaqueta y retorna un flujo de datos opaco de longitud fija, similar a *unpack_fstring()*.

Unpacker.**unpack_string**()
Desempaqueta y retorna una cadena de caracteres de longitud variable. La longitud de la cadena de caracteres es desempaquetada primero como un entero sin signo, luego los datos de la cadena de caracteres son desempaquetados con el método *unpack_fstring()*.

Unpacker.**unpack_opaque**()
Desempaqueta y retorna un flujo de datos opaco de longitud variable, similar a *unpack_string()*.

Unpacker.**unpack_bytes**()
Desempaqueta y retorna una secuencia de bytes de longitud variable, similar a *unpack_string()*.

Los siguientes métodos soportan el desempaquetado de arreglos y listas:

Unpacker.**unpack_list**(*unpack_item*)
Desempaqueta y retorna una lista de elementos homogéneos. La lista se desempaqueta un elemento a la vez desempaquetando primero un indicador entero sin signo. Si el flag es 1, el elemento se desempaqueta 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 desempaquetar los elementos.

Unpacker.**unpack_farray**(*n*, *unpack_item*)
Desempaqueta 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 desempaquetar cada elemento.

Unpacker.**unpack_array**(*unpack_item*)
Desempaqueta 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.

35.24.3 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)
```

CAPÍTULO 36

Consideraciones de seguridad

Los siguientes módulos tienen consideraciones de seguridad específicas:

- `base64`: *consideraciones de seguridad de base64* en **RFC 4648**
- `cgi`: *consideraciones de seguridad de CGI*
- `hashlib`: *todos los constructores toman un argumento de solo palabra clave «usedforsecurity» que deshabilita algoritmos inseguros y bloqueados conocidos*
- `http.server` is not suitable for production use, only implementing basic security checks. See the *security considerations*.
- `logging`: *La configuración de registro usa eval()*
- `multiprocessing`: *Connection.recv() usa pickle*
- `pickle`: *Restricción de globals en pickle*
- `random` no debe usarse por motivos de seguridad, usa `secrets` en su lugar
- `shelve`: *el estante se basa en pickle y, por lo tanto, no es adecuado para tratar con fuentes no confiables*
- `ssl`: *consideraciones de seguridad SSL/TLS*
- `subprocess`: *Consideraciones sobre seguridad de subprocesos*
- `tempfile`: *mktemp está en desuso debido a la vulnerabilidad a condiciones de urgencia*
- `xml`: *vulnerabilidades XML*
- `zipfile`: *los archivos de .zip preparados maliciosamente pueden causar agotamiento del volumen del disco*

APÉNDICE A

Glosario

>>> 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 — Automated Python 2 to 3 code translation*.

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.

Consulte *variable annotation*, *function annotation*, **PEP 484** y **PEP 526**, que describen esta funcionalidad. Consulte también annotations-howto para conocer las mejores prácticas sobre cómo trabajar con anotaciones.

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 un sentencia `async with` al definir los métodos `__aenter__()` `__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 A value associated with an object which is usually referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

It is possible to give an object an attribute whose name is not an identifier as defined by identifiers, for example using *`setattr()`*, if the object allows it. Such an attribute will not be accessible using a dotted expression, and would instead need to be retrieved with *`getattr()`*.

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`.

referencia prestada In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Se recomienda llamar a `Py_INCREF()` en la *referencia prestada* para convertirla en una *referencia fuerte* in situ, excepto cuando el objeto no se puede destruir antes del último uso de la referencia prestada. La función `Py_NewRef()` se puede utilizar para crear una nueva *referencia fuerte*.

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 corren 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*.

callable A callable is an object that can be called, possibly with a set of arguments (see *argument*), with the following syntax:

```
callable(argument1, argument2, ...)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the `__call__()` method is also a callable.

retrollamada Una función de subrutina que se pasa como un argumento para ejecutarse en algún momento en el futuro.

clase 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 se 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(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

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 descriptores 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 obtener más información sobre los métodos de los descriptores, consulte descriptors o Guía práctica de uso de los descriptores.

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.

comprensión de diccionarios Una forma compacta de procesar todos o parte de los elementos en un iterable y retornar un diccionario con los resultados. `results = {n: n ** 2 for n in range(10)}` genera un diccionario que contiene la clave `n` asignada al valor `n ** 2`. Ver 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 *statement*s 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*.

codificación del sistema de archivos y manejador de errores Controlador de errores y codificación utilizado por Python para decodificar bytes del sistema operativo y codificar Unicode en el sistema operativo.

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 API pueden lanzar `UnicodeError`.

Las funciones `sys.getfilesystemencoding()` y `sys.getfilesystemencodeerrors()` se pueden utilizar para obtener la codificación del sistema de archivos y el controlador de errores.

La *codificación del sistema de archivos y el manejador de errores* se configuran al inicio de Python mediante la función `PyConfig_Read()`: consulte los miembros `filesystem_encoding` y `filesystem_errors` de `PyConfig`.

See also the *locale encoding*.

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.

Consulte *variable annotation* y **PEP 484**, que describen esta funcionalidad. Consulte también annotations-howto para conocer las mejores prácticas sobre cómo trabajar con anotaciones.

__future__ Un future statement, `from __future__ import <feature>`, indica al compilador que compile el módulo actual utilizando una sintaxis o semántica que se convertirá en estándar en una versión futura de Python. El módulo `__future__` documenta los posibles valores de *feature*. Al importar este módulo y evaluar sus variables, puede ver cuándo se agregó por primera vez una nueva característica al lenguaje y cuándo se convertirá (o se convirtió) en la predeterminada:

```
>>> 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**.

tipos genéricos A *type* that can be parameterized; typically a container class such as `list` or `dict`. Used for *type hints* and *annotations*.

For more details, see *generic alias types*, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

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.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

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 An Integrated Development and Learning Environment for Python. *IDLE* is a basic editor and interpreter environment which ships with the standard distribution of 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 intérpreter es que el módulo __main__ o el script que estaba corriendo termine su ejecución.

iterable Un objeto capaz de retornar sus miembros uno por vez. Ejemplos de iterables son todos los tipos de secuencias (como *list*, *str*, y *tuple*) y algunos de tipos no secuenciales, como *dict*, *objeto archivo*, y objetos de cualquier clase que defina con los métodos __iter__() o con un método __getitem__() que implementen la semántica de *Sequence*.

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*.

Detalles de implementación de CPython: CPython does not consistently apply the requirement that an iterator define __iter__().

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.

codificación de la configuración regional En Unix, es la codificación de la configuración regional LC_CTYPE. Se puede configurar con locale.setlocale(locale.LC_CTYPE, new_locale).

En Windows, es la página de códigos ANSI (por ejemplo, cp1252).

locale.getpreferredencoding(False) se puede utilizar para obtener la codificación de la configuración regional.

Python usa el *filesystem encoding and error handler* para convertir entre nombres de archivo Unicode y nombres de archivo en bytes.

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.

metaclase La clase de una clase. Las definiciones de clases crean nombres de clase, un diccionario de clase, y una lista de clases base. Las metaclases 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 metaclases a medida. La mayoría de los usuario nunca necesitarán esta herramienta, pero cuando la necesidad surge, las metaclases 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 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 hechas personalizadamente 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, `__getattribute__()`, 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 A Python *module* which can contain submodules or recursively, subpackages. Technically, a package is a Python module with a `__path__` attribute.

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 guardo 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 provisional 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 provisorias, 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 predeclarando 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.

La clase abstracta base `collections.abc.Sequence` define una interfaz mucho más rica que va más allá de sólo `__getitem__()` y `__len__()`, agregando `count()`, `index()`, `__contains__()`, y `__reversed__()`. Los tipos que implementan esta interfaz expandida pueden ser registrados explícitamente usando `register()`.

comprensión de conjuntos Una forma compacta de procesar todos o parte de los elementos en un iterable y retornar un conjunto con los resultados. `results = {c for c in 'abracadabra' if c not in 'abc'}` genera el conjunto de cadenas `{'r', 'd'}`. Ver 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 (suscrito) 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`.

referencia fuerte In Python's C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

La función `Py_NewRef()` se puede utilizar para crear una referencia fuerte a un objeto. Por lo general, se debe llamar a la función `Py_DECREF()` en la referencia fuerte antes de salir del alcance de la referencia fuerte, para evitar filtrar una referencia.

Consulte también *borrowed reference*.

codificación de texto A string in Python is a sequence of Unicode code points (in range `U+0000–U+10FFFF`). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as «encoding», and recreating the string from the sequence of bytes is known as «decoding».

There are a variety of different text serialization *codecs*, which are collectively referred to as «text encodings».

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:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

podría ser más legible así:

```
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.

Consulte *function annotation*, **PEP 484** y **PEP 526**, que describen esta funcionalidad. Consulte también annotations-howto para conocer las mejores prácticas sobre cómo trabajar con anotaciones.

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.

APÉNDICE B

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;
- the Docutils project for creating reStructuredText and the Docutils suite;
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

B.1 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!

APÉNDICE C

Historia y Licencia

C.1 Historia del software

Python fue creado a principios de la década de 1990 por Guido van Rossum en Stichting Mathematisch Centrum (CWI, ver https://www.cwi.nl/) en los Países Bajos como sucesor de un idioma llamado ABC. Guido sigue siendo el autor principal de Python, aunque incluye muchas contribuciones de otros.

En 1995, Guido continuó su trabajo en Python en la Corporation for National Research Initiatives (CNRI, consulte https://www.cnri.reston.va.us/) en Reston, Virginia, donde lanzó varias versiones del software.

En mayo de 2000, Guido y el equipo de desarrollo central de Python se trasladaron a BeOpen.com para formar el equipo de BeOpen PythonLabs. En octubre del mismo año, el equipo de PythonLabs se trasladó a Digital Creations (ahora Zope Corporation; consulte https://www.zope.org/). En 2001, se formó la Python Software Foundation (PSF, consulte https://www.python.org/psf/), una organización sin fines de lucro creada específicamente para poseer la propiedad intelectual relacionada con Python. Zope Corporation es miembro patrocinador del PSF.

Todas las versiones de Python son de código abierto (consulte https://opensource.org/ para conocer la definición de código abierto). Históricamente, la mayoría de las versiones de Python, pero no todas, también han sido compatibles con GPL; la siguiente tabla resume las distintas versiones.

Lanzamiento	Derivado de	Año	Dueño/a	¿compatible con GPL?
0.9.0 hasta 1.2	n/a	1991-1995	CWI	sí
1.3 hasta 1.5.2	1.2	1995-1999	CNRI	sí
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	sí
2.1.1	2.1+2.0.1	2001	PSF	sí
2.1.2	2.1.1	2002	PSF	sí
2.1.3	2.1.2	2002	PSF	sí
2.2 y superior	2.1.1	2001-ahora	PSF	sí

Nota: Compatible con GPL no significa que estemos distribuyendo Python bajo la GPL. Todas las licencias de Python, a diferencia de la GPL, le permiten distribuir una versión modificada sin que los cambios sean de código

abierto. Las licencias compatibles con GPL permiten combinar Python con otro software que se publica bajo la GPL; los otros no lo hacen.

Gracias a los muchos voluntarios externos que han trabajado bajo la dirección de Guido para hacer posibles estos lanzamientos.

C.2 Términos y condiciones para acceder o usar Python

El software y la documentación de Python están sujetos a *Acuerdo de licencia de PSF*.

A partir de Python 3.8.6, los ejemplos, recetas y otros códigos de la documentación tienen licencia doble según el Acuerdo de licencia de PSF y la *Licencia BSD de cláusula cero*.

Parte del software incorporado en Python está bajo diferentes licencias. Las licencias se enumeran con el código correspondiente a esa licencia. Consulte *Licencias y reconocimientos para software incorporado* para obtener una lista incompleta de estas licencias.

C.2.1 ACUERDO DE LICENCIA DE PSF PARA PYTHON | lanzamiento |

```
1. This LICENSE AGREEMENT is between the Python Software Foundation␣
↪("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise␣
↪using Python
   3.10.13 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 3.10.13 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 3.10.13 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 3.10.13 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
   3.10.13.

4. PSF is making Python 3.10.13 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 3.10.13 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.10.13
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A␣
↪RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.10.13, 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 3.10.13, Licensee␣
↪agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 ACUERDO DE LICENCIA DE BEOPEN.COM PARA PYTHON 2.0

ACUERDO DE LICENCIA DE CÓDIGO ABIERTO DE BEOPEN PYTHON VERSIÓN 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.
```

(continué en la próxima página)

(proviene de la página anterior)

```
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 ACUERDO DE LICENCIA CNRI PARA 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 ACUERDO DE LICENCIA CWI PARA PYTHON 0.9.0 HASTA 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 LICENCIA BSD DE CLÁUSULA CERO PARA CÓDIGO EN EL PYTHON | lanzamiento | DOCUMENTACIÓN

```
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 Licencias y reconocimientos para software incorporado

Esta sección es una lista incompleta, pero creciente, de licencias y reconocimientos para software de terceros incorporado en la distribución de Python.

C.3.1 Mersenne Twister

El módulo `_random` incluye código basado en una descarga de http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. Los siguientes son los comentarios textuales del código original:

```
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, https://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Servicios de socket asincrónicos

Los módulos *asynchat* y *asyncore* contienen el siguiente aviso:

```
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 Gestión de cookies

El módulo *http.cookies* contiene el siguiente aviso:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Seguimiento de ejecución

El módulo *trace* contiene el siguiente aviso:

```
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 funciones UUencode y UUdecode

El módulo *uu* contiene el siguiente aviso:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 Llamadas a procedimientos remotos XML

El módulo *xmlrpc.client* contiene el siguiente aviso:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

El módulo `test_epoll` contiene el siguiente aviso:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Seleccionar kqueue

El módulo *select* contiene el siguiente aviso para la interfaz kqueue:

```
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

El archivo `Python/pyhash.c` contiene la implementación de Marek Majkowski del algoritmo SipHash24 de Dan Bernstein. Contiene la siguiente nota:

```
<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>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod y 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 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. 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

Los módulos *hashlib*, *posix*, *ssl*, *crypt* utilizan la biblioteca OpenSSL para un rendimiento adicional si el sistema operativo la pone a disposición. Además, los instaladores de Windows y macOS para Python pueden incluir una copia de las bibliotecas de OpenSSL, por lo que incluimos una copia de la licencia de OpenSSL aquí:

```
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:
   *
   * 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    * 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,
    * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
    * included with this distribution is covered by the same copyright terms
    * except that the holder is Tim Hudson (tjh@cryptsoft.com).
    *
    * Copyright remains Eric Young's, and as such any Copyright notices in
    * the code are not to be removed.
    * If this package is used in a product, Eric Young should be given attribution
    * as the author of the parts of the library used.
    * This can be in the form of a textual message at program startup or
    * in documentation (online or textual) provided with the package.
    *
    * Redistribution and use in source and binary forms, with or without
    * modification, are permitted provided that the following conditions
    * are met:
    * 1. Redistributions of source code must retain the copyright
    *    notice, this list of conditions and the following disclaimer.
    * 2. Redistributions in binary form must reproduce the above copyright
    *    notice, this list of conditions and the following disclaimer in the
    *    documentation and/or other materials provided with the distribution.
    * 3. All advertising materials mentioning features or use of this software
    *    must display the following acknowledgement:
    *    "This product includes cryptographic software written by
    *     Eric Young (eay@cryptsoft.com)"
    *    The word 'cryptographic' can be left out if the rouines from the library
    *    being used are not cryptographic related :-).
    * 4. If you include any Windows specific code (or a derivative thereof) from
    *    the apps directory (application code) you must include an acknowledgement:
    *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
    *
    * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
    * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
    * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
    * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
    * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
    * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
    * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

(continué en la próxima página)

(proviene de la página anterior)

```
    * 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

La extensión `pyexpat` se construye usando una copia incluida de las fuentes de expatriados a menos que la construcción esté configurada `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

La extensión `_ctypes` se construye usando una copia incluida de las fuentes de libffi a menos que la construcción esté configurada `--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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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

La extensión *zlib* se crea utilizando una copia incluida de las fuentes de zlib si la versión de zlib encontrada en el sistema es demasiado antigua para ser utilizada para la compilación:

```
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:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

La implementación de la tabla hash utilizada por *tracemalloc* se basa en el proyecto cfuhash:

```
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
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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

El módulo `_decimal` se construye usando una copia incluida de la biblioteca libmpdec a menos que la construcción esté configurada `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 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 Conjunto de pruebas W3C C14N

El conjunto de pruebas C14N 2.0 en el paquete *test* (`Lib/test/xmltestdata/c14n-20/`) se recuperó del sitio web de W3C en https://www.w3.org/TR/xml-c14n2-testcases/ y se distribuye bajo la licencia BSD de 3 cláusulas:

```
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:
```

(continué en la próxima página)

(proviene de la página anterior)

```
* 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.
```

C.3.19 Audioop

The audioop module uses the code base in g771.c file of the SoX project:

```
Programming the AdLib/Sound Blaster
FM Music Chips
Version 2.0 (24 Feb 1992)
Copyright (c) 1991, 1992 by Jeffrey S. Lee
jlee@smylex.uucp
Warranty and Copyright Policy
This document is provided on an "as-is" basis, and its author makes
no warranty or representation, express or implied, with respect to
its quality performance or fitness for a particular purpose.  In no
event will the author of this document be liable for direct, indirect,
special, incidental, or consequential damages arising out of the use
or inability to use the information contained within.  Use of this
document is at your own risk.
This file may be used and copied freely so long as the applicable
copyright notices are retained, and no modifications are made to the
text of the document.  No money shall be charged for its distribution
beyond reasonable shipping, handling and duplication costs, nor shall
proprietary changes be made to this document so that it cannot be
distributed freely.  This document may not be included in published
material or commercial packages without the written consent of its
author.
```

APÉNDICE D

Derechos de autor

Python y esta documentación es:

Copyright © 2001-2023 Python Software Foundation. All rights reserved.

Derechos de autor © 2000 BeOpen.com. Todos los derechos reservados.

Derechos de autor © 1995-2000 Corporation for National Research Initiatives. Todos los derechos reservados.

Derechos de autor © 1991-1995 Stichting Mathematisch Centrum. Todos los derechos reservados.

Consulte *Historia y Licencia* para obtener información completa sobre licencias y permisos.

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 https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

Índice de Módulos Python

_

__future__, 1822
__main__, 1773
_thread, 916

a

abc, 1810
aifc, 1985
argparse, 680
array, 270
ast, 1889
asynchat, 1987
asyncio, 919
asyncore, 1990
atexit, 1814
audioop, 1994

b

base64, 1182
bdb, 1689
binascii, 1186
binhex, 1185
bisect, 266
builtins, 1773
bz2, 520

c

calendar, 234
cgi, 1997
cgitb, 2004
chunk, 2005
cmath, 328
cmd, 1443
code, 1849
codecs, 174
codeop, 1851
collections, 239
collections.abc, 257
colorsys, 1389
compileall, 1936
concurrent.futures, 882
configparser, 562
contextlib, 1796
contextvars, 912
copy, 286
copyreg, 481
cProfile, 1705
crypt (*Unix*), 2006
csv, 555
ctypes, 788
curses (*Unix*), 756
curses.ascii, 775
curses.panel, 778
curses.textpad, 774

d

dataclasses, 1785
datetime, 193
dbm, 486
dbm.dumb, 490
dbm.gnu (*Unix*), 488
dbm.ndbm (*Unix*), 489
decimal, 331
difflib, 144
dis, 1940
distutils, 1731
doctest, 1548

e

email, 1093
email.charset, 1145
email.contentmanager, 1123
email.encoders, 1147
email.errors, 1116
email.generator, 1106
email.header, 1142
email.headerregistry, 1117
email.iterators, 1150
email.message, 1094
email.mime, 1140
email.parser, 1102
email.policy, 1109
email.utils, 1148
encodings.idna, 190
encodings.mbcs, 190
encodings.utf_8_sig, 191
ensurepip, 1732

enum, 294
errno, 782

f

faulthandler, 1694
fcntl (*Unix*), 1977
filecmp, 444
fileinput, 437
fnmatch, 452
fractions, 358
ftplib, 1308
functools, 396

g

gc, 1824
getopt, 711
getpass, 755
gettext, 1391
glob, 451
graphlib, 313
grp (*Unix*), 1973
gzip, 517

h

hashlib, 585
heapq, 262
hmac, 595
html, 1191
html.entities, 1196
html.parser, 1192
http, 1299
http.client, 1301
http.cookiejar, 1350
http.cookies, 1347
http.server, 1341

i

idlelib, 1510
imaplib, 1316
imghdr, 2008
imp, 2009
importlib, 1862
importlib.abc, 1865
importlib.machinery, 1874
importlib.metadata, 1884
importlib.resources, 1872
importlib.util, 1878
inspect, 1827
io, 656
ipaddress, 1372
itertools, 381

j

json, 1152
json.tool, 1160

k

keyword, 1927

l

lib2to3, 1665
linecache, 453
locale, 1400
logging, 714
logging.config, 730
logging.handlers, 742
lzma, 524

m

mailbox, 1161
mailcap, 2014
marshal, 485
math, 320
mimetypes, 1179
mmap, 1087
modulefinder, 1859
msilib (*Windows*), 2015
msvcrt (*Windows*), 1957
multiprocessing, 834
multiprocessing.connection, 864
multiprocessing.dummy, 868
multiprocessing.managers, 855
multiprocessing.pool, 861
multiprocessing.shared_memory, 877
multiprocessing.sharedctypes, 853

n

netrc, 580
nis (*Unix*), 2021
nntplib, 2022
numbers, 317

o

operator, 406
optparse, 2028
os, 601
os.path, 431
ossaudiodev (*Linux, FreeBSD*), 2055

p

pathlib, 413
pdb, 1696
pickle, 465
pickletools, 1953
pipes (*Unix*), 2060
pkgutil, 1855
platform, 779
plistlib, 581
poplib, 1313
posix (*Unix*), 1971
pprint, 287
profile, 1705
pstats, 1707
pty (*Unix*), 1975
pwd (*Unix*), 1972
py_compile, 1934
pyclbr, 1932

pydoc, 1544

q

queue, 909
quopri, 1188

r

random, 360
re, 124
readline (*Unix*), 162
reprlib, 292
resource (*Unix*), 1979
rlcompleter, 166
runpy, 1860

s

sched, 908
secrets, 597
select, 1068
selectors, 1075
shelve, 482
shlex, 1448
shutil, 454
signal, 1078
site, 1844
smtpd, 2061
smtplib, 1323
sndhdr, 2064
socket, 1008
socketserver, 1333
spwd (*Unix*), 2065
sqlite3, 491
ssl, 1033
stat, 439
statistics, 368
string, 113
stringprep, 160
struct, 167
subprocess, 889
sunau, 2066
symtable, 1922
sys, 1749
sysconfig, 1769
syslog (*Unix*), 1983

t

tabnanny, 1932
tarfile, 539
telnetlib, 2068
tempfile, 446
termios (*Unix*), 1974
test, 1666
test.support, 1668
test.support.bytecode_helper, 1679
test.support.import_helper, 1682
test.support.os_helper, 1680
test.support.script_helper, 1677
test.support.socket_helper, 1677
test.support.threading_helper, 1679
test.support.warnings_helper, 1683
textwrap, 155
threading, 821
time, 670
timeit, 1711
tkinter, 1455
tkinter.colorchooser (*Tk*), 1468
tkinter.commondialog (*Tk*), 1472
tkinter.dnd (*Tk*), 1473
tkinter.filedialog (*Tk*), 1470
tkinter.font (*Tk*), 1468
tkinter.messagebox (*Tk*), 1472
tkinter.scrolledtext (*Tk*), 1473
tkinter.simpledialog (*Tk*), 1469
tkinter.tix, 1493
tkinter.ttk, 1474
token, 1924
tokenize, 1928
trace, 1716
traceback, 1816
tracemalloc, 1719
tty (*Unix*), 1975
turtle, 1409
turtledemo, 1441
types, 280
typing, 1511

u

unicodedata, 158
unittest, 1571
unittest.mock, 1601
urllib, 1270
urllib.error, 1297
urllib.parse, 1288
urllib.request, 1270
urllib.response, 1288
urllib.robotparser, 1298
uu, 2071
uuid, 1329

v

venv, 1733

w

warnings, 1779
wave, 1387
weakref, 272
webbrowser, 1257
winreg (*Windows*), 1959
winsound (*Windows*), 1968
wsgiref, 1260
wsgiref.handlers, 1265
wsgiref.headers, 1262
wsgiref.simple_server, 1263
wsgiref.util, 1260
wsgiref.validate, 1264

X

xdrlib, 2072
xml, 1197
xml.dom, 1218
xml.dom.minidom, 1228
xml.dom.pulldom, 1233
xml.etree.ElementTree, 1198
xml.parsers.expat, 1247
xml.parsers.expat.errors, 1254
xml.parsers.expat.model, 1253
xml.sax, 1235
xml.sax.handler, 1236
xml.sax.saxutils, 1242
xml.sax.xmlreader, 1243
xmlrpc.client, 1359
xmlrpc.server, 1366

Z

zipapp, 1742
zipfile, 529
zipimport, 1853
zlib, 513
zoneinfo, 229

No alfabético

..
    in pathnames, 654
..., **2079**
    ellipsis literal, 31, 96
    in doctests, 1555
    interpreter prompt, 1552, 1763
    placeholder, 158, 287, 293
??
    in regular expressions, 125
. (*dot*)
    in glob-style wildcards, 451
    in pathnames, 654, 655
    in printf-style formatting, 57, 72
    in regular expressions, 125
    in string formatting, 115
! (*exclamation*)
    in a command interpreter, 1444
    in curses module, 777
    in glob-style wildcards, 451, 452
    in string formatting, 115
    in struct format strings, 168
- (*minus*)
    binary operator, 35
    in doctests, 1557
    in glob-style wildcards, 451, 452
    in printf-style formatting, 58, 73
    in regular expressions, 126
    in string formatting, 117
    unary operator, 35
! (*pdb command*), 1702
? (*question mark*)
    in a command interpreter, 1444
    in argparse module, 693
    in AST grammar, 1892
    in glob-style wildcards, 451, 452
    in regular expressions, 125
    in SQL statements, 502
    in struct format strings, 170
    replacement character, 177
(*hash*)
    comment, 1844
    in doctests, 1557
    in printf-style formatting, 58, 73
    in regular expressions, 131
    in string formatting, 117
$ (*dollar*)
    environment variables expansion, 433
    in regular expressions, 125
    in template strings, 122
    interpolation in configuration files, 567
% (*percent*)
    datetime format, 224, 674, 675
    environment variables expansion (*Windows*), 433, 1961
    interpolation in configuration files, 566
    operador, 35
    printf-style formatting, 57, 72
& (*ampersand*)
    operador, 36
(?
    in regular expressions, 126
(?!
    in regular expressions, 128
(?#
    in regular expressions, 127
(?(
    in regular expressions, 128
() (*parentheses*)
    in printf-style formatting, 57, 72
    in regular expressions, 126
(?:
    in regular expressions, 127
(?<!
    in regular expressions, 128
(?<=
    in regular expressions, 128
(?=
    in regular expressions, 127
(?P<
    in regular expressions, 127
(?P=
    in regular expressions, 127
*?

in regular expressions, 125
* (*asterisk*)
in argparse module, 693
in AST grammar, 1892
in glob-style wildcards, 451, 452
in printf-style formatting, 57, 72
in regular expressions, 125
operador, 35
**
in glob-style wildcards, 451
operador, 35
+?
in regular expressions, 125
+ (*plus*)
binary operator, 35
in argparse module, 693
in doctests, 1557
in printf-style formatting, 58, 73
in regular expressions, 125
in string formatting, 117
unary operator, 35
, (*comma*)
in string formatting, 117
-
python--m-py_compileopción de línea de comando, 1935
/ (*slash*)
in pathnames, 654, 655
operador, 35
//
operador, 35
2-digit years, 670
2to3, **2079**
: (*colon*)
in SQL statements, 502
in string formatting, 115
path separator (*POSIX*), 655
; (*semicolon*), 655
< (*less*)
in string formatting, 117
in struct format strings, 168
operador, 34
<<
operador, 36
<=
operador, 34
<BLANKLINE>, 1555
<file> ... <fileN>
python--m-py_compileopción de línea de comando, 1935
!=
operador, 34
= (*equals*)
in string formatting, 117
in struct format strings, 168
==
operador, 34
> (*greater*)
in string formatting, 117
in struct format strings, 168
operador, 34
>=
operador, 34
>>
operador, 36
>>>, **2079**
interpreter prompt, 1552, 1763
@ (*at*)
in struct format strings, 168
[] (*square brackets*)
in glob-style wildcards, 451, 452
in regular expressions, 126
in string formatting, 115
\ (*backslash*)
escape sequence, 177
in pathnames (*Windows*), 654
in regular expressions, 126, 128
\\
in regular expressions, 129
\a
in regular expressions, 129
\A
in regular expressions, 128
\b
in regular expressions, 128, 129
\B
in regular expressions, 128
\d
in regular expressions, 129
\D
in regular expressions, 129
\f
in regular expressions, 129
\g
in regular expressions, 133
\n
in regular expressions, 129
\N
escape sequence, 177
in regular expressions, 129
\r
in regular expressions, 129
\s
in regular expressions, 129
\S
in regular expressions, 129
\t
in regular expressions, 129
\u
escape sequence, 177
in regular expressions, 129
\U
escape sequence, 177
in regular expressions, 129
\v
in regular expressions, 129

\w
    in regular expressions, 129
\W
    in regular expressions, 129
\x
    escape sequence, 177
    in regular expressions, 129
\Z
    in regular expressions, 129
^ (*caret*)
    in curses module, 777
    in regular expressions, 125, 126
    in string formatting, 117
    marker, 1554, 1816
    operador, 36
_ (*underscore*)
    gettext, 1392
    in string formatting, 117
__abs__() (*en el módulo operator*), 406
__add__() (*en el módulo operator*), 406
__and__() (*en el módulo operator*), 406
__args__ (*atributo de genericalias*), 92
__bases__ (*atributo de class*), 97
__breakpointhook__ (*en el módulo sys*), 1753
__bytes__() (*método de email.message.EmailMessage*), 1095
__bytes__() (*método de email.message.Message*), 1133
__call__() (*método de email.headerregistry.HeaderRegistry*), 1121
__call__() (*método de weakref.finalize*), 276
__callback__ (*atributo de weakref.ref*), 274
__cause__ (*atributo de traceback.TracebackException*), 1818
__ceil__() (*método de fractions.Fraction*), 360
__class__ (*atributo de instance*), 97
__class__ (*atributo de unittest.mock.Mock*), 1611
__code__ (*function object attribute*), 96
__concat__() (*en el módulo operator*), 408
__contains__() (*en el módulo operator*), 408
__contains__() (*método de email.message.EmailMessage*), 1096
__contains__() (*método de email.message.Message*), 1135
__contains__() (*método de mailbox.Mailbox*), 1164
__context__ (*atributo de traceback.TracebackException*), 1818
__copy__() (*copy protocol*), 286
__debug__ (*variable incorporada*), 32
__deepcopy__() (*copy protocol*), 286
__del__() (*método de io.IOBase*), 662
__delitem__() (*en el módulo operator*), 408
__delitem__() (*método de email.message.EmailMessage*), 1097
__delitem__() (*método de email.message.Message*), 1135
__delitem__() (*método de mailbox.Mailbox*), 1162
__delitem__() (*método de mailbox.MH*), 1168
__dict__ (*atributo de object*), 97
__dir__() (*método de unittest.mock.Mock*), 1607
__displayhook__ (*en el módulo sys*), 1753
__doc__ (*atributo de types.ModuleType*), 282
__enter__() (*método de contextmanager*), 88
__enter__() (*método de winreg.PyHKEY*), 1967
__eq__() (*en el módulo operator*), 406
__eq__() (*instance method*), 34
__eq__() (*método de email.charset.Charset*), 1146
__eq__() (*método de email.header.Header*), 1144
__eq__() (*método de memoryview*), 75
__excepthook__ (*en el módulo sys*), 1753
__excepthook__ (*en el módulo threading*), 822
__exit__() (*método de contextmanager*), 88
__exit__() (*método de winreg.PyHKEY*), 1967
__floor__() (*método de fractions.Fraction*), 360
__floordiv__() (*en el módulo operator*), 407
__format__, 13
__format__() (*método de datetime.date*), 202
__format__() (*método de datetime.datetime*), 211
__format__() (*método de datetime.time*), 216
__format__() (*método de ipaddress.IPv4Address*), 1374
__format__() (*método de ipaddress.IPv6Address*), 1376
__fspath__() (*método de os.PathLike*), 604
__future__, **2084**
    módulo, 1822
__ge__() (*en el módulo operator*), 406
__ge__() (*instance method*), 34
__getitem__() (*en el módulo operator*), 408
__getitem__() (*método de email.headerregistry.HeaderRegistry*), 1121
__getitem__() (*método de email.message.EmailMessage*), 1096
__getitem__() (*método de email.message.Message*), 1135
__getitem__() (*método de mailbox.Mailbox*), 1163
__getitem__() (*método de re.Match*), 137
__getnewargs__() (*método de object*), 472
__getnewargs_ex__() (*método de object*), 472
__getstate__() (*copy protocol*), 476
__getstate__() (*método de object*), 472
__gt__() (*en el módulo operator*), 406
__gt__() (*instance method*), 34
__iadd__() (*en el módulo operator*), 411
__iand__() (*en el módulo operator*), 411
__iconcat__() (*en el módulo operator*), 411
__ifloordiv__() (*en el módulo operator*), 411
__ilshift__() (*en el módulo operator*), 411
__imatmul__() (*en el módulo operator*), 411
__imod__() (*en el módulo operator*), 411
__import__()
    función incorporada, 28
__import__() (*en el módulo importlib*), 1863
__imul__() (*en el módulo operator*), 411
__index__() (*en el módulo operator*), 407

__init__() (*método de difflib.HtmlDiff*), 145
__init__() (*método de logging.Handler*), 718
__interactivehook__ (*en el módulo sys*), 1760
__inv__() (*en el módulo operator*), 407
__invert__() (*en el módulo operator*), 407
__ior__() (*en el módulo operator*), 411
__ipow__() (*en el módulo operator*), 412
__irshift__() (*en el módulo operator*), 412
__isub__() (*en el módulo operator*), 412
__iter__() (*método de container*), 41
__iter__() (*método de iterator*), 41
__iter__() (*método de mailbox.Mailbox*), 1163
__iter__() (*método de unittest.TestSuite*), 1592
__itruediv__() (*en el módulo operator*), 412
__ixor__() (*en el módulo operator*), 412
__le__() (*en el módulo operator*), 406
__le__() (*instance method*), 34
__len__() (*método de email.message.EmailMessage*), 1096
__len__() (*método de email.message.Message*), 1135
__len__() (*método de mailbox.Mailbox*), 1164
__loader__ (*atributo de types.ModuleType*), 282
__lshift__() (*en el módulo operator*), 407
__lt__() (*en el módulo operator*), 406
__lt__() (*instance method*), 34
__main__
    módulo, 1773, 1860, 1861
__matmul__() (*en el módulo operator*), 407
__missing__(), 84
__missing__() (*método de collections.defaultdict*), 248
__mod__() (*en el módulo operator*), 407
__mro__ (*atributo de class*), 97
__mul__() (*en el módulo operator*), 407
__name__ (*atributo de definition*), 97
__name__ (*atributo de types.ModuleType*), 283
__ne__() (*en el módulo operator*), 406
__ne__() (*instance method*), 34
__ne__() (*método de email.charset.Charset*), 1146
__ne__() (*método de email.header.Header*), 1144
__neg__() (*en el módulo operator*), 407
__next__() (*método de csv.csvreader*), 560
__next__() (*método de iterator*), 41
__not__() (*en el módulo operator*), 406
__optional_keys__ (*atributo de typing.TypedDict*), 1533
__or__() (*en el módulo operator*), 407
__origin__ (*atributo de genericalias*), 92
__package__ (*atributo de types.ModuleType*), 283
__parameters__ (*atributo de genericalias*), 92
__pos__() (*en el módulo operator*), 407
__pow__() (*en el módulo operator*), 407
__qualname__ (*atributo de definition*), 97
__reduce__() (*método de object*), 473
__reduce_ex__() (*método de object*), 473
__repr__() (*método de multiprocessing.managers.BaseProxy*), 861
__repr__() (*método de netrc.netrc*), 581
__required_keys__ (*atributo de typing.TypedDict*), 1533
__round__() (*método de fractions.Fraction*), 360
__rshift__() (*en el módulo operator*), 407
__setitem__() (*en el módulo operator*), 408
__setitem__() (*método de email.message.EmailMessage*), 1096
__setitem__() (*método de email.message.Message*), 1135
__setitem__() (*método de mailbox.Mailbox*), 1162
__setitem__() (*método de mailbox.Maildir*), 1165
__setstate__() (*copy protocol*), 476
__setstate__() (*método de object*), 472
__slots__, **2090**
__spec__ (*atributo de types.ModuleType*), 283
__stderr__ (*en el módulo sys*), 1766
__stdin__ (*en el módulo sys*), 1766
__stdout__ (*en el módulo sys*), 1766
__str__() (*método de datetime.date*), 201
__str__() (*método de datetime.datetime*), 211
__str__() (*método de datetime.time*), 216
__str__() (*método de email.charset.Charset*), 1146
__str__() (*método de email.header.Header*), 1144
__str__() (*método de email.headerregistry.Address*), 1122
__str__() (*método de email.headerregistry.Group*), 1122
__str__() (*método de email.message.EmailMessage*), 1095
__str__() (*método de email.message.Message*), 1132
__str__() (*método de multiprocessing.managers.BaseProxy*), 861
__sub__() (*en el módulo operator*), 407
__subclasses__() (*método de class*), 97
__subclasshook__() (*método de abc.ABCMeta*), 1811
__suppress_context__ (*atributo de traceback.TracebackException*), 1818
__total__ (*atributo de typing.TypedDict*), 1533
__truediv__() (*en el módulo operator*), 407
__truediv__() (*método de importlib.abc.Traversable*), 1871
__unraisablehook__ (*en el módulo sys*), 1753
__xor__() (*en el módulo operator*), 407
anonymous (*atributo de ctypes.Structure*), 819
_asdict() (*método de collections.somenamedtuple*), 251
_b_base_ (*atributo de ctypes._CData*), 815
_b_needsfree_ (*atributo de ctypes._CData*), 815
_callmethod() (*método de multiprocessing.managers.BaseProxy*), 860
_CData (*clase en ctypes*), 815
_clear_type_cache() (*en el módulo sys*), 1751
_current_exceptions() (*en el módulo sys*), 1751
_current_frames() (*en el módulo sys*), 1751
_debugmallocstats() (*en el módulo sys*), 1752
_enablelegacywindowsfsencoding() (*en el módulo sys*), 1766

_exit() (*en el módulo os*), 642
_field_defaults (*atributo de collections.somenamedtuple*), 251
_fields (*atributo de ast.AST*), 1892
_fields (*atributo de collections.somenamedtuple*), 251
fields (*atributo de ctypes.Structure*), 818
_flush() (*método de wsgiref.handlers.BaseHandler*), 1266
_FuncPtr (*clase en ctypes*), 809
_get_child_mock() (*método de unittest.mock.Mock*), 1607
_get_preferred_schemes() (*en el módulo sysconfig*), 1770
_getframe() (*en el módulo sys*), 1757
_getvalue() (*método de multiprocessing.managers.BaseProxy*), 861
_handle (*atributo de ctypes.PyDLL*), 808
length (*atributo de ctypes.Array*), 819
_locale
    módulo, 1400
_make() (*método de clase de collections.somenamedtuple*), 251
_makeResult() (*método de unittest.TextTestRunner*), 1597
_name (*atributo de ctypes.PyDLL*), 808
_objects (*atributo de ctypes._CData*), 815
pack (*atributo de ctypes.Structure*), 818
_parse() (*método de gettext.NullTranslations*), 1394
_Pointer (*clase en ctypes*), 819
_replace() (*método de collections.somenamedtuple*), 251
_setroot() (*método de xml.etree.ElementTree.ElementTree*), 1213
_SimpleCData (*clase en ctypes*), 816
_structure() (*en el módulo email.iterators*), 1151
_thread
    módulo, 916
type (*atributo de ctypes._Pointer*), 820
type (*atributo de ctypes.Array*), 819
_write() (*método de wsgiref.handlers.BaseHandler*), 1266
_xoptions (*en el módulo sys*), 1768
{} (*curly brackets*)
    in regular expressions, 125
    in string formatting, 115
| (*vertical bar*)
    in regular expressions, 126
    operador, 36
~ (*tilde*)
    home directory expansion, 433
    operador, 36

A

-a
    astopción de línea de comando, 1921
    pickletoolsopción de línea de comando, 1954
A (*en el módulo re*), 130
a la espera, **2080**
a2b_base64() (*en el módulo binascii*), 1186
a2b_hex() (*en el módulo binascii*), 1188
a2b_hqx() (*en el módulo binascii*), 1186
a2b_qp() (*en el módulo binascii*), 1186
a2b_uu() (*en el módulo binascii*), 1186
a85decode() (*en el módulo base64*), 1184
a85encode() (*en el módulo base64*), 1183
abc
    módulo, 1810
ABC (*clase en abc*), 1810
ABCMeta (*clase en abc*), 1810
abiflags (*en el módulo sys*), 1749
abort() (*en el módulo os*), 641
abort() (*método de asyncio.DatagramTransport*), 982
abort() (*método de asyncio.WriteTransport*), 981
abort() (*método de ftplib.FTP*), 1311
abort() (*método de threading.Barrier*), 833
above() (*método de curses.panel.Panel*), 778
ABOVE_NORMAL_PRIORITY_CLASS (*en el módulo subprocess*), 901
abs()
    función incorporada, 6
abs() (*en el módulo operator*), 406
abs() (*método de decimal.Context*), 345
AbsoluteLinkError, 541
AbsolutePathError, 541
abspath() (*en el módulo os.path*), 432
AbstractAsyncContextManager (*clase en contextlib*), 1796
AbstractBasicAuthHandler (*clase en urllib.request*), 1274
AbstractChildWatcher (*clase en asyncio*), 994
abstractclassmethod() (*en el módulo abc*), 1813
AbstractContextManager (*clase en contextlib*), 1796
AbstractDigestAuthHandler (*clase en urllib.request*), 1274
AbstractEventLoop (*clase en asyncio*), 972
AbstractEventLoopPolicy (*clase en asyncio*), 992
abstractmethod() (*en el módulo abc*), 1812
abstractproperty() (*en el módulo abc*), 1813
AbstractSet (*clase en typing*), 1536
abstractstaticmethod() (*en el módulo abc*), 1813
accept() (*método de asyncore.dispatcher*), 1992
accept() (*método de multiprocessing.connection.Listener*), 865
accept() (*método de socket.socket*), 1022
access() (*en el módulo os*), 619
accumulate() (*en el módulo itertools*), 383
aclose() (*método de contextlib.AsyncExitStack*), 1804
aclosing() (*en el módulo contextlib*), 1798
acos() (*en el módulo cmath*), 329
acos() (*en el módulo math*), 325
acosh() (*en el módulo cmath*), 329

acosh() (*en el módulo math*), 326
acquire() (*método de _thread.lock*), 917
acquire() (*método de asyncio.Condition*), 943
acquire() (*método de asyncio.Lock*), 941
acquire() (*método de asyncio.Semaphore*), 944
acquire() (*método de logging.Handler*), 718
acquire() (*método de multiprocessing.Lock*), 850
acquire() (*método de multiprocessing.RLock*), 851
acquire() (*método de threading.Condition*), 829
acquire() (*método de threading.Lock*), 826
acquire() (*método de threading.RLock*), 827
acquire() (*método de threading.Semaphore*), 830
acquire_lock() (*en el módulo imp*), 2012
action (*atributo de optparse.Option*), 2042
Action (*clase en argparse*), 699
ACTIONS (*atributo de optparse.Option*), 2054
active_children() (*en el módulo multiprocessing*), 847
active_count() (*en el módulo threading*), 821
actual() (*método de tkinter.font.Font*), 1469
Add (*clase en ast*), 1896
add() (*en el módulo audioop*), 1994
add() (*en el módulo operator*), 406
add() (*método de decimal.Context*), 345
add() (*método de frozenset*), 83
add() (*método de graphlib.TopologicalSorter*), 314
add() (*método de mailbox.Mailbox*), 1162
add() (*método de mailbox.Maildir*), 1165
add() (*método de msilib.RadioButtonGroup*), 2020
add() (*método de pstats.Stats*), 1707
add() (*método de tarfile.TarFile*), 545
add() (*método de tkinter.ttk.Notebook*), 1482
add_alias() (*en el módulo email.charset*), 1147
add_alternative() (*método de email.message.EmailMessage*), 1101
add_argument() (*método de argparse.ArgumentParser*), 690
add_argument_group() (*método de argparse.ArgumentParser*), 707
add_attachment() (*método de email.message.EmailMessage*), 1101
add_cgi_vars() (*método de wsgiref.handlers.BaseHandler*), 1266
add_charset() (*en el módulo email.charset*), 1147
add_child_handler() (*método de asyncio.AbstractChildWatcher*), 994
add_codec() (*en el módulo email.charset*), 1147
add_cookie_header() (*método de http.cookiejar.CookieJar*), 1352
add_data() (*en el módulo msilib*), 2016
add_dll_directory() (*en el módulo os*), 641
add_done_callback() (*método de asyncio.Future*), 977
add_done_callback() (*método de asyncio.Task*), 932
add_done_callback() (*método de concurrent.futures.Future*), 887
add_fallback() (*método de gettext.NullTranslations*), 1394
add_file() (*método de msilib.Directory*), 2019
add_flag() (*método de mailbox.MaildirMessage*), 1171
add_flag() (*método de mailbox.mboxMessage*), 1172
add_flag() (*método de mailbox.MMDFMessage*), 1176
add_folder() (*método de mailbox.Maildir*), 1165
add_folder() (*método de mailbox.MH*), 1167
add_get_handler() (*método de email.contentmanager.ContentManager*), 1123
add_handler() (*método de urllib.request.OpenerDirector*), 1276
add_header() (*método de email.message.EmailMessage*), 1097
add_header() (*método de email.message.Message*), 1135
add_header() (*método de urllib.request.Request*), 1276
add_header() (*método de wsgiref.headers.Headers*), 1262
add_history() (*en el módulo readline*), 163
add_label() (*método de mailbox.BabylMessage*), 1175
add_mutually_exclusive_group() (*método de argparse.ArgumentParser*), 707
add_option() (*método de optparse.OptionParser*), 2040
add_parent() (*método de urllib.request.BaseHandler*), 1277
add_password() (*método de urllib.request.HTTPPasswordMgr*), 1280
add_password() (*método de urllib.request.HTTPPasswordMgrWithPriorAuth*), 1280
add_reader() (*método de asyncio.loop*), 964
add_related() (*método de email.message.EmailMessage*), 1101
add_section() (*método de configparser.ConfigParser*), 576
add_section() (*método de configparser.RawConfigParser*), 579
add_sequence() (*método de mailbox.MHMessage*), 1174
add_set_handler() (*método de email.contentmanager.ContentManager*), 1123
add_signal_handler() (*método de asyncio.loop*), 966
add_stream() (*en el módulo msilib*), 2016
add_subparsers() (*método de argparse.ArgumentParser*), 703
add_tables() (*en el módulo msilib*), 2016
add_type() (*en el módulo mimetypes*), 1180
add_unredirected_header() (*método de urllib.request.Request*), 1276

add_writer() (*método de asyncio.loop*), 964
addAsyncCleanup() (*método de unittest.IsolatedAsyncioTestCase*), 1590
addaudithook() (*en el módulo sys*), 1749
addch() (*método de curses.window*), 763
addClassCleanup() (*método de clase de unittest.TestCase*), 1589
addCleanup() (*método de unittest.TestCase*), 1589
addcomponent() (*método de turtle.Shape*), 1437
addError() (*método de unittest.TestResult*), 1596
addExpectedFailure() (*método de unittest.TestResult*), 1596
addFailure() (*método de unittest.TestResult*), 1596
addfile() (*método de tarfile.TarFile*), 545
addFilter() (*método de logging.Handler*), 719
addFilter() (*método de logging.Logger*), 717
addHandler() (*método de logging.Logger*), 717
addinfourl (*clase en urllib.response*), 1288
addLevelName() (*en el módulo logging*), 727
addModuleCleanup() (*en el módulo unittest*), 1600
addnstr() (*método de curses.window*), 764
AddPackagePath() (*en el módulo modulefinder*), 1859
addr (*atributo de smtpd.SMTPChannel*), 2063
addr_spec (*atributo de email.headerregistry.Address*), 1122
address (*atributo de email.headerregistry.SingleAddressHeader*), 1120
address (*atributo de multiprocessing.connection.Listener*), 865
address (*atributo de multiprocessing.managers.BaseManager*), 856
Address (*clase en email.headerregistry*), 1121
address_exclude() (*método de ipaddress.IPv4Network*), 1379
address_exclude() (*método de ipaddress.IPv6Network*), 1382
address_family (*atributo de socketserver.BaseServer*), 1335
address_string() (*método de http.server.BaseHTTPRequestHandler*), 1344
addresses (*atributo de email.headerregistry.AddressHeader*), 1119
addresses (*atributo de email.headerregistry.Group*), 1122
AddressHeader (*clase en email.headerregistry*), 1119
addressof() (*en el módulo ctypes*), 812
AddressValueError, 1385
addshape() (*en el módulo turtle*), 1435
addsitedir() (*en el módulo site*), 1846
addSkip() (*método de unittest.TestResult*), 1596
addstr() (*método de curses.window*), 764
addSubTest() (*método de unittest.TestResult*), 1596
addSuccess() (*método de unittest.TestResult*), 1596
addTest() (*método de unittest.TestSuite*), 1591
addTests() (*método de unittest.TestSuite*), 1591
addTypeEqualityFunc() (*método de unittest.TestCase*), 1587
addUnexpectedSuccess() (*método de unittest.TestResult*), 1596
adjust_int_max_str_digits() (*en el módulo test.support*), 1676
adjusted() (*método de decimal.Decimal*), 337
adler32() (*en el módulo zlib*), 513
administrador asincrónico de contexto, **2080**
administrador de contextos, **2081**
ADPCM, Intel/DVI, 1994
adpcm2lin() (*en el módulo audioop*), 1994
AF_ALG (*en el módulo socket*), 1014
AF_CAN (*en el módulo socket*), 1013
AF_INET (*en el módulo socket*), 1012
AF_INET6 (*en el módulo socket*), 1012
AF_LINK (*en el módulo socket*), 1015
AF_PACKET (*en el módulo socket*), 1014
AF_QIPCRTR (*en el módulo socket*), 1015
AF_RDS (*en el módulo socket*), 1014
AF_UNIX (*en el módulo socket*), 1012
AF_VSOCK (*en el módulo socket*), 1014
aifc
    módulo, 1985
aifc() (*método de aifc.aifc*), 1986
AIFF, 1985, 2005
aiff() (*método de aifc.aifc*), 1986
AIFF-C, 1985, 2005
aiter()
    función incorporada, 6
alarm() (*en el módulo signal*), 1081
a-LAW, 1994
A-LAW, 1987, 2064
alaw2lin() (*en el módulo audioop*), 1994
alcances anidados, **2088**
ALERT_DESCRIPTION_HANDSHAKE_FAILURE (*en el módulo ssl*), 1045
ALERT_DESCRIPTION_INTERNAL_ERROR (*en el módulo ssl*), 1045
AlertDescription (*clase en ssl*), 1045
algorithms_available (*en el módulo hashlib*), 586
algorithms_guaranteed (*en el módulo hashlib*), 586
Alias
    Generic, 89
alias (*clase en ast*), 1904
alias (*pdb command*), 1702
alias de tipos, **2091**
alignment() (*en el módulo ctypes*), 812
alive (*atributo de weakref.finalize*), 276
all()
    función incorporada, 6
all_errors (*en el módulo ftplib*), 1310
all_features (*en el módulo xml.sax.handler*), 1238
all_frames (*atributo de tracemalloc.Filter*), 1726

all_properties (*en el módulo xml.sax.handler*), 1238
all_suffixes() (*en el módulo importlib.machinery*), 1874
all_tasks() (*en el módulo asyncio*), 931
allocate_lock() (*en el módulo _thread*), 917
allow_reuse_address (*atributo de socketserver.BaseServer*), 1336
allowed_domains() (*método de http.cookiejar.DefaultCookiePolicy*), 1356
alt() (*en el módulo curses.ascii*), 777
ALT_DIGITS (*en el módulo locale*), 1403
altsep (*en el módulo os*), 655
altzone (*en el módulo time*), 679
ALWAYS_EQ (*en el módulo test.support*), 1670
ALWAYS_TYPED_ACTIONS (*atributo de optparse.Option*), 2054
AMPER (*en el módulo token*), 1925
AMPEREQUAL (*en el módulo token*), 1926
anchor (*atributo de pathlib.PurePath*), 418
and
    operador, 33, 34
And (*clase en ast*), 1897
and_() (*en el módulo operator*), 406
anext()
    función incorporada, 6
AnnAssign (*clase en ast*), 1902
--annotate
    pickletoolsopción de línea de comando, 1954
Annotated (*en el módulo typing*), 1524
annotation
    type annotation; type hint, 89
annotation (*atributo de inspect.Parameter*), 1834
anotación, **2079**
anotación de función, **2084**
anotación de variable, **2092**
answer_challenge() (*en el módulo multiprocessing.connection*), 864
anticipate_failure() (*en el módulo test.support*), 1673
Any (*en el módulo typing*), 1520
ANY (*en el módulo unittest.mock*), 1635
any()
    función incorporada, 7
AnyStr (*en el módulo typing*), 1529
apagado del intérprete, **2085**
API provisional, **2089**
api_version (*en el módulo sys*), 1768
apilevel (*en el módulo sqlite3*), 495
apop() (*método de poplib.POP3*), 1315
append() (*método de array.array*), 271
append() (*método de collections.deque*), 245
append() (*método de email.header.Header*), 1143
append() (*método de imaplib.IMAP4*), 1318
append() (*método de msilib.CAB*), 2018
append() (*método de pipes.Template*), 2060
append() (*método de xml.etree.ElementTree.Element*), 1211
append() (*sequence method*), 44
append_history_file() (*en el módulo readline*), 163
appendChild() (*método de xml.dom.Node*), 1221
appendleft() (*método de collections.deque*), 245
application_uri() (*en el módulo wsgiref.util*), 1261
apply (*2to3 fixer*), 1662
apply() (*método de multiprocessing.pool.Pool*), 862
apply_async() (*método de multiprocessing.pool.Pool*), 862
apply_defaults() (*método de inspect.BoundArguments*), 1836
architecture() (*en el módulo platform*), 779
archive (*atributo de zipimport.zipimporter*), 1855
archivo binario, **2080**
archivo de texto, **2091**
aRepr (*en el módulo reprlib*), 292
arg (*clase en ast*), 1914
argparse
    módulo, 680
args (*atributo de BaseException*), 102
args (*atributo de functools.partial*), 405
args (*atributo de inspect.BoundArguments*), 1836
args (*atributo de subprocess.CompletedProcess*), 890
args (*atributo de subprocess.Popen*), 899
args (*atributo de typing.ParamSpec*), 1529
args (*pdb command*), 1701
args_from_interpreter_flags() (*en el módulo test.support*), 1671
argtypes (*atributo de ctypes._FuncPtr*), 809
ArgumentDefaultsHelpFormatter (*clase en argparse*), 685
ArgumentError, 810
argumento, **2079**
argumento nombrado, **2086**
argumento posicional, **2089**
ArgumentParser (*clase en argparse*), 682
arguments (*atributo de inspect.BoundArguments*), 1836
arguments (*clase en ast*), 1914
argv (*en el módulo sys*), 1750
arithmetic, 35
ArithmeticError, 102
array
    módulo, 59, 270
array (*clase en array*), 270
Array (*clase en ctypes*), 819
Array() (*en el módulo multiprocessing*), 853
Array() (*en el módulo multiprocessing.sharedctypes*), 853
Array() (*método de multiprocessing.managers.SyncManager*), 857
arrays, 270
arraysize (*atributo de sqlite3.Cursor*), 503
article() (*método de nntplib.NNTP*), 2027

as_bytes() (*método de email.message.EmailMessage*), 1095
as_bytes() (*método de email.message.Message*), 1132
as_completed() (*en el módulo asyncio*), 929
as_completed() (*en el módulo concurrent.futures*), 888
as_file() (*en el módulo importlib.resources*), 1872
as_integer_ratio() (*método de decimal.Decimal*), 337
as_integer_ratio() (*método de float*), 38
as_integer_ratio() (*método de fractions.Fraction*), 359
as_integer_ratio() (*método de int*), 38
as_posix() (*método de pathlib.PurePath*), 420
as_string() (*método de email.message.EmailMessage*), 1095
as_string() (*método de email.message.Message*), 1132
as_tuple() (*método de decimal.Decimal*), 337
as_uri() (*método de pathlib.PurePath*), 420
ASCII (*en el módulo re*), 130
ascii()
    función incorporada, 7
ascii() (*en el módulo curses.ascii*), 777
ascii_letters (*en el módulo string*), 113
ascii_lowercase (*en el módulo string*), 113
ascii_uppercase (*en el módulo string*), 113
asctime() (*en el módulo time*), 671
asdict() (*en el módulo dataclasses*), 1789
asin() (*en el módulo cmath*), 329
asin() (*en el módulo math*), 325
asinh() (*en el módulo cmath*), 329
asinh() (*en el módulo math*), 326
askcolor() (*en el módulo tkinter.colorchooser*), 1468
askdirectory() (*en el módulo tkinter.filedialog*), 1471
askfloat() (*en el módulo tkinter.simpledialog*), 1469
askinteger() (*en el módulo tkinter.simpledialog*), 1469
askokcancel() (*en el módulo tkinter.messagebox*), 1473
askopenfile() (*en el módulo tkinter.filedialog*), 1470
askopenfilename() (*en el módulo tkinter.filedialog*), 1470
askopenfilenames() (*en el módulo tkinter.filedialog*), 1470
askopenfiles() (*en el módulo tkinter.filedialog*), 1470
askquestion() (*en el módulo tkinter.messagebox*), 1473
askretrycancel() (*en el módulo tkinter.messagebox*), 1473
asksaveasfile() (*en el módulo tkinter.filedialog*), 1470
asksaveasfilename() (*en el módulo tkinter.filedialog*), 1471
askstring() (*en el módulo tkinter.simpledialog*), 1469
askyesno() (*en el módulo tkinter.messagebox*), 1473
askyesnocancel() (*en el módulo tkinter.messagebox*), 1473
assert
    sentencia, 103
Assert (*clase en ast*), 1903
assert_any_await() (*método de unittest.mock.AsyncMock*), 1615
assert_any_call() (*método de unittest.mock.Mock*), 1605
assert_awaited() (*método de unittest.mock.AsyncMock*), 1614
assert_awaited_once() (*método de unittest.mock.AsyncMock*), 1614
assert_awaited_once_with() (*método de unittest.mock.AsyncMock*), 1615
assert_awaited_with() (*método de unittest.mock.AsyncMock*), 1614
assert_called() (*método de unittest.mock.Mock*), 1605
assert_called_once() (*método de unittest.mock.Mock*), 1605
assert_called_once_with() (*método de unittest.mock.Mock*), 1605
assert_called_with() (*método de unittest.mock.Mock*), 1605
assert_has_awaits() (*método de unittest.mock.AsyncMock*), 1615
assert_has_calls() (*método de unittest.mock.Mock*), 1606
assert_not_awaited() (*método de unittest.mock.AsyncMock*), 1615
assert_not_called() (*método de unittest.mock.Mock*), 1606
assert_python_failure() (*en el módulo test.support.script_helper*), 1678
assert_python_ok() (*en el módulo test.support.script_helper*), 1678
assertAlmostEqual() (*método de unittest.TestCase*), 1586
assertCountEqual() (*método de unittest.TestCase*), 1587
assertDictEqual() (*método de unittest.TestCase*), 1588
assertEqual() (*método de unittest.TestCase*), 1582
assertFalse() (*método de unittest.TestCase*), 1582
assertGreater() (*método de unittest.TestCase*), 1586
assertGreaterEqual() (*método de unittest.TestCase*), 1586
assertIn() (*método de unittest.TestCase*), 1582
assertInBytecode() (*método de test.support.bytecode_helper.BytecodeTestCase*), 1679
AssertionError, 103
assertIs() (*método de unittest.TestCase*), 1582

assertIsInstance() (*método de unittest.TestCase*), 1583
assertIsNone() (*método de unittest.TestCase*), 1582
assertIsNot() (*método de unittest.TestCase*), 1582
assertIsNotNone() (*método de unittest.TestCase*), 1582
assertLess() (*método de unittest.TestCase*), 1586
assertLessEqual() (*método de unittest.TestCase*), 1586
assertListEqual() (*método de unittest.TestCase*), 1587
assertLogs() (*método de unittest.TestCase*), 1585
assertMultiLineEqual() (*método de unittest.TestCase*), 1587
assertNoLogs() (*método de unittest.TestCase*), 1585
assertNotAlmostEqual() (*método de unittest.TestCase*), 1586
assertNotEqual() (*método de unittest.TestCase*), 1582
assertNotIn() (*método de unittest.TestCase*), 1582
assertNotInBytecode() (*método de test.support.bytecode_helper.BytecodeTestCase*), 1679
assertNotIsInstance() (*método de unittest.TestCase*), 1583
assertNotRegex() (*método de unittest.TestCase*), 1586
assertRaises() (*método de unittest.TestCase*), 1583
assertRaisesRegex() (*método de unittest.TestCase*), 1584
assertRegex() (*método de unittest.TestCase*), 1586
asserts (*2to3 fixer*), 1662
assertSequenceEqual() (*método de unittest.TestCase*), 1587
assertSetEqual() (*método de unittest.TestCase*), 1587
assertTrue() (*método de unittest.TestCase*), 1582
assertTupleEqual() (*método de unittest.TestCase*), 1587
assertWarns() (*método de unittest.TestCase*), 1584
assertWarnsRegex() (*método de unittest.TestCase*), 1584
Assign (*clase en ast*), 1901
assignment
    slice, 44
    subscript, 44
ast
    módulo, 1889
AST (*clase en ast*), 1892
astimezone() (*método de datetime.datetime*), 208
astopción de línea de comando
    -a, 1921
    -h, 1921
    --help, 1921
    -i <indent>, 1921
    --include-attributes, 1921
    --indent <indent>, 1921
    -m <mode>, 1921
    --mode <mode>, 1921
    --no-type-comments, 1921
astuple() (*en el módulo dataclasses*), 1790
ASYNC (*en el módulo token*), 1926
async_chat (*clase en asynchat*), 1988
async_chat.ac_in_buffer_size (*en el módulo asynchat*), 1988
async_chat.ac_out_buffer_size (*en el módulo asynchat*), 1988
AsyncContextDecorator (*clase en contextlib*), 1801
AsyncContextManager (*clase en typing*), 1540
asynccontextmanager() (*en el módulo contextlib*), 1797
AsyncExitStack (*clase en contextlib*), 1803
AsyncFor (*clase en ast*), 1917
AsyncFunctionDef (*clase en ast*), 1917
AsyncGenerator (*clase en collections.abc*), 261
AsyncGenerator (*clase en typing*), 1539
AsyncGeneratorType (*en el módulo types*), 281
asynchat
    módulo, 1987
asyncio
    módulo, 919
asyncio.subprocess.DEVNULL (*variable incorporada*), 947
asyncio.subprocess.PIPE (*variable incorporada*), 947
asyncio.subprocess.Process (*clase incorporada*), 947
asyncio.subprocess.STDOUT (*variable incorporada*), 947
AsyncIterable (*clase en collections.abc*), 261
AsyncIterable (*clase en typing*), 1539
AsyncIterator (*clase en collections.abc*), 261
AsyncIterator (*clase en typing*), 1539
AsyncMock (*clase en unittest.mock*), 1613
asyncore
    módulo, 1990
AsyncResult (*clase en multiprocessing.pool*), 863
asyncSetUp() (*método de unittest.IsolatedAsyncioTestCase*), 1589
asyncTearDown() (*método de unittest.IsolatedAsyncioTestCase*), 1589
AsyncWith (*clase en ast*), 1917
AT (*en el módulo token*), 1926
at_eof() (*método de asyncio.StreamReader*), 937
atan() (*en el módulo cmath*), 329
atan() (*en el módulo math*), 325
atan2() (*en el módulo math*), 325
atanh() (*en el módulo cmath*), 329
atanh() (*en el módulo math*), 326
ATEQUAL (*en el módulo token*), 1926
atexit
    módulo, 1814
atexit (*atributo de weakref.finalize*), 276
atof() (*en el módulo locale*), 1405
atoi() (*en el módulo locale*), 1405

atributo, **2080**
attach() (*método de email.message.Message*), 1133
attach_loop() (*método de asyncio.AbstractChildWatcher*), 994
attach_mock() (*método de unittest.mock.Mock*), 1607
AttlistDeclHandler() (*método de xml.parsers.expat.xmlparser*), 1250
attrgetter() (*en el módulo operator*), 408
attrib (*atributo de xml.etree.ElementTree.Element*), 1211
Attribute (*clase en ast*), 1898
AttributeError, 103
attributes (*atributo de xml.dom.Node*), 1220
AttributesImpl (*clase en xml.sax.xmlreader*), 1243
AttributesNSImpl (*clase en xml.sax.xmlreader*), 1244
attroff() (*método de curses.window*), 764
attron() (*método de curses.window*), 764
attrset() (*método de curses.window*), 764
Audio Interchange File Format, 1985, 2005
AUDIO_FILE_ENCODING_ADPCM_G721 (*en el módulo sunau*), 2066
AUDIO_FILE_ENCODING_ADPCM_G722 (*en el módulo sunau*), 2066
AUDIO_FILE_ENCODING_ADPCM_G723_3 (*en el módulo sunau*), 2066
AUDIO_FILE_ENCODING_ADPCM_G723_5 (*en el módulo sunau*), 2066
AUDIO_FILE_ENCODING_ALAW_8 (*en el módulo sunau*), 2066
AUDIO_FILE_ENCODING_DOUBLE (*en el módulo sunau*), 2066
AUDIO_FILE_ENCODING_FLOAT (*en el módulo sunau*), 2066
AUDIO_FILE_ENCODING_LINEAR_8 (*en el módulo sunau*), 2066
AUDIO_FILE_ENCODING_LINEAR_16 (*en el módulo sunau*), 2066
AUDIO_FILE_ENCODING_LINEAR_24 (*en el módulo sunau*), 2066
AUDIO_FILE_ENCODING_LINEAR_32 (*en el módulo sunau*), 2066
AUDIO_FILE_ENCODING_MULAW_8 (*en el módulo sunau*), 2066
AUDIO_FILE_MAGIC (*en el módulo sunau*), 2066
AUDIODEV, 2056
audioop
    módulo, 1994
audit events, 1685
audit() (*en el módulo sys*), 1750
auditing, 1750
AugAssign (*clase en ast*), 1902
auth() (*método de ftplib.FTP_TLS*), 1313
auth() (*método de smtplib.SMTP*), 1326
authenticate() (*método de imaplib.IMAP4*), 1318
AuthenticationError, 843
authenticators() (*método de netrc.netrc*), 581
authkey (*atributo de multiprocessing.Process*), 842
auto (*clase en enum*), 295
autorange() (*método de timeit.Timer*), 1713
available_timezones() (*en el módulo zoneinfo*), 233
avg() (*en el módulo audioop*), 1995
avgpp() (*en el módulo audioop*), 1995
avoids_symlink_attacks (*atributo de shutil.rmtree*), 457
Await (*clase en ast*), 1917
AWAIT (*en el módulo token*), 1926
await_args (*atributo de unittest.mock.AsyncMock*), 1616
await_args_list (*atributo de unittest.mock.AsyncMock*), 1616
await_count (*atributo de unittest.mock.AsyncMock*), 1616
Awaitable (*clase en collections.abc*), 261
Awaitable (*clase en typing*), 1540

B

-b
    compileallopción de línea de comando, 1936
    unittestopción de línea de comando, 1574
b2a_base64() (*en el módulo binascii*), 1186
b2a_hex() (*en el módulo binascii*), 1187
b2a_hqx() (*en el módulo binascii*), 1187
b2a_qp() (*en el módulo binascii*), 1186
b2a_uu() (*en el módulo binascii*), 1186
b16decode() (*en el módulo base64*), 1183
b16encode() (*en el módulo base64*), 1183
b32decode() (*en el módulo base64*), 1183
b32encode() (*en el módulo base64*), 1183
b32hexdecode() (*en el módulo base64*), 1183
b32hexencode() (*en el módulo base64*), 1183
b64decode() (*en el módulo base64*), 1182
b64encode() (*en el módulo base64*), 1182
b85decode() (*en el módulo base64*), 1184
b85encode() (*en el módulo base64*), 1184
Babyl (*clase en mailbox*), 1168
BabylMessage (*clase en mailbox*), 1174
back() (*en el módulo turtle*), 1414
backslashreplace
    error handler's name, 177
backslashreplace_errors() (*en el módulo codecs*), 178
backup() (*método de sqlite3.Connection*), 500
backward() (*en el módulo turtle*), 1414
BadGzipFile, 517
BadStatusLine, 1303
BadZipfile, 529
BadZipFile, 529
Balloon (*clase en tkinter.tix*), 1494
Barrier (*clase en multiprocessing*), 850
Barrier (*clase en threading*), 833

Barrier() (*método de multiprocessing.managers.SyncManager*), 856
base64
    encoding, 1182
    módulo, 1182, 1186
base_exec_prefix (*en el módulo sys*), 1750
base_prefix (*en el módulo sys*), 1750
BaseCGIHandler (*clase en wsgiref.handlers*), 1265
BaseCookie (*clase en http.cookies*), 1347
BaseException, 102
BaseHandler (*clase en urllib.request*), 1273
BaseHandler (*clase en wsgiref.handlers*), 1266
BaseHeader (*clase en email.headerregistry*), 1118
BaseHTTPRequestHandler (*clase en http.server*), 1341
BaseManager (*clase en multiprocessing.managers*), 855
basename() (*en el módulo os.path*), 432
BaseProtocol (*clase en asyncio*), 984
BaseProxy (*clase en multiprocessing.managers*), 860
BaseRequestHandler (*clase en socketserver*), 1337
BaseRotatingHandler (*clase en logging.handlers*), 744
BaseSelector (*clase en selectors*), 1076
BaseServer (*clase en socketserver*), 1335
basestring (*2to3 fixer*), 1662
BaseTransport (*clase en asyncio*), 979
basicConfig() (*en el módulo logging*), 728
BasicContext (*clase en decimal*), 343
BasicInterpolation (*clase en configparser*), 566
BasicTestRunner (*clase en test.support*), 1676
baudrate() (*en el módulo curses*), 757
bbox() (*método de tkinter.ttk.Treeview*), 1486
BDADDR_ANY (*en el módulo socket*), 1015
BDADDR_LOCAL (*en el módulo socket*), 1015
bdb
    módulo, 1689, 1696
Bdb (*clase en bdb*), 1690
BdbQuit, 1689
BDFL, **2080**
beep() (*en el módulo curses*), 757
Beep() (*en el módulo winsound*), 1968
BEFORE_ASYNC_WITH (*opcode*), 1946
begin_fill() (*en el módulo turtle*), 1423
begin_poly() (*en el módulo turtle*), 1429
below() (*método de curses.panel.Panel*), 778
BELOW_NORMAL_PRIORITY_CLASS (*en el módulo subprocess*), 901
benchmarking, 673, 676
Benchmarking, 1711
--best
    gzipopción de línea de comando, 519
betavariate() (*en el módulo random*), 364
bgcolor() (*en el módulo turtle*), 1430
bgpic() (*en el módulo turtle*), 1430
bias() (*en el módulo audioop*), 1995
bidirectional() (*en el módulo unicodedata*), 159
bigaddrspacetest() (*en el módulo test.support*), 1674
BigEndianStructure (*clase en ctypes*), 818
bigmemtest() (*en el módulo test.support*), 1674
bin()
    función incorporada, 7
binary
    data, packing, 167
    literals, 35
Binary (*clase en msilib*), 2016
Binary (*clase en xmlrpc.client*), 1362
binary mode, 20
binary semaphores, 916
BINARY_ADD (*opcode*), 1944
BINARY_AND (*opcode*), 1945
BINARY_FLOOR_DIVIDE (*opcode*), 1944
BINARY_LSHIFT (*opcode*), 1945
BINARY_MATRIX_MULTIPLY (*opcode*), 1944
BINARY_MODULO (*opcode*), 1944
BINARY_MULTIPLY (*opcode*), 1944
BINARY_OR (*opcode*), 1945
BINARY_POWER (*opcode*), 1944
BINARY_RSHIFT (*opcode*), 1945
BINARY_SUBSCR (*opcode*), 1945
BINARY_SUBTRACT (*opcode*), 1945
BINARY_TRUE_DIVIDE (*opcode*), 1944
BINARY_XOR (*opcode*), 1945
BinaryIO (*clase en typing*), 1536
binascii
    módulo, 1186
bind (*widgets*), 1466
bind() (*método de asyncore.dispatcher*), 1992
bind() (*método de inspect.Signature*), 1833
bind() (*método de socket.socket*), 1022
bind_partial() (*método de inspect.Signature*), 1833
bind_port() (*en el módulo test.support.socket_helper*), 1677
bind_textdomain_codeset() (*en el módulo gettext*), 1392
bind_unix_socket() (*en el módulo test.support.socket_helper*), 1677
bindtextdomain() (*en el módulo gettext*), 1391
bindtextdomain() (*en el módulo locale*), 1407
binhex
    módulo, 1185, 1186
binhex() (*en el módulo binhex*), 1185
BinOp (*clase en ast*), 1896
bisect
    módulo, 266
bisect() (*en el módulo bisect*), 267
bisect_left() (*en el módulo bisect*), 267
bisect_right() (*en el módulo bisect*), 267
bit_count() (*método de int*), 37
bit_length() (*método de int*), 37
BitAnd (*clase en ast*), 1896
bitmap() (*método de msilib.Dialog*), 2020
BitOr (*clase en ast*), 1896

bitwise
operations, 36
BitXor (*clase en ast*), 1896
bk() (*en el módulo turtle*), 1414
bkgd() (*método de curses.window*), 764
bkgdset() (*método de curses.window*), 764
blake2b() (*en el módulo hashlib*), 589
blake2b, blake2s, 588
blake2b.MAX_DIGEST_SIZE (*en el módulo hashlib*), 590
blake2b.MAX_KEY_SIZE (*en el módulo hashlib*), 590
blake2b.PERSON_SIZE (*en el módulo hashlib*), 590
blake2b.SALT_SIZE (*en el módulo hashlib*), 590
blake2s() (*en el módulo hashlib*), 589
blake2s.MAX_DIGEST_SIZE (*en el módulo hashlib*), 590
blake2s.MAX_KEY_SIZE (*en el módulo hashlib*), 590
blake2s.PERSON_SIZE (*en el módulo hashlib*), 590
blake2s.SALT_SIZE (*en el módulo hashlib*), 590
block_size (*atributo de hmac.HMAC*), 596
blocked_domains() (*método de http.cookiejar.DefaultCookiePolicy*), 1355
BlockingIOError, 108, 659
blocksize (*atributo de http.client.HTTPConnection*), 1305
bloqueo global del intérprete, **2085**
body() (*método de nntplib.NNTP*), 2027
body() (*método de tkinter.simpledialog.Dialog*), 1470
body_encode() (*método de email.charset.Charset*), 1146
body_encoding (*atributo de email.charset.Charset*), 1146
body_line_iterator() (*en el módulo email.iterators*), 1150
BOLD (*en el módulo tkinter.font*), 1468
BOM (*en el módulo codecs*), 176
BOM_BE (*en el módulo codecs*), 176
BOM_LE (*en el módulo codecs*), 176
BOM_UTF8 (*en el módulo codecs*), 176
BOM_UTF16 (*en el módulo codecs*), 176
BOM_UTF16_BE (*en el módulo codecs*), 176
BOM_UTF16_LE (*en el módulo codecs*), 176
BOM_UTF32 (*en el módulo codecs*), 176
BOM_UTF32_BE (*en el módulo codecs*), 176
BOM_UTF32_LE (*en el módulo codecs*), 176
bool (*clase incorporada*), 7
Boolean
objeto, 35
operations, 33, 34
type, 7
values, 97
BOOLEAN_STATES (*atributo de configparser.ConfigParser*), 572
BoolOp (*clase en ast*), 1897
bootstrap() (*en el módulo ensurepip*), 1733
border() (*método de curses.window*), 764
bottom() (*método de curses.panel.Panel*), 778
bottom_panel() (*en el módulo curses.panel*), 778
BoundArguments (*clase en inspect*), 1836
BoundaryError, 1116
BoundedSemaphore (*clase en asyncio*), 945
BoundedSemaphore (*clase en multiprocessing*), 850
BoundedSemaphore (*clase en threading*), 831
BoundedSemaphore() (*método de multiprocessing.managers.SyncManager*), 856
box() (*método de curses.window*), 765
bpbynumber (*atributo de bdb.Breakpoint*), 1690
bpformat() (*método de bdb.Breakpoint*), 1690
bplist (*atributo de bdb.Breakpoint*), 1690
bpprint() (*método de bdb.Breakpoint*), 1690
Break (*clase en ast*), 1906
break (*pdb command*), 1699
break_anywhere() (*método de bdb.Bdb*), 1692
break_here() (*método de bdb.Bdb*), 1692
break_long_words (*atributo de textwrap.TextWrapper*), 158
break_on_hyphens (*atributo de textwrap.TextWrapper*), 158
Breakpoint (*clase en bdb*), 1689
breakpoint()
función incorporada, 7
breakpointhook() (*en el módulo sys*), 1751
breakpoints, 1501
broadcast_address (*atributo de ipaddress.IPv4Network*), 1378
broadcast_address (*atributo de ipaddress.IPv6Network*), 1381
broken (*atributo de threading.Barrier*), 834
BrokenBarrierError, 834
BrokenExecutor, 888
BrokenPipeError, 108
BrokenProcessPool, 888
BrokenThreadPool, 888
BROWSER, 1257, 1258
BsdDbShelf (*clase en shelve*), 484
buf (*atributo de multiprocessing.shared_memory.SharedMemory*), 878
--buffer
unittestopción de línea de comando, 1574
buffer (*2to3 fixer*), 1662
buffer (*atributo de io.TextIOBase*), 666
buffer (*atributo de unittest.TestResult*), 1595
buffer protocol
binary sequence types, 59
str (*built-in class*), 48
buffer size, I/O, 20
buffer_info() (*método de array.array*), 271
buffer_size (*atributo de xml.parsers.expat.xmlparser*), 1249
buffer_text (*atributo de xml.parsers.expat.xmlparser*), 1249
buffer_updated() (*método de asyncio.BufferedProtocol*), 985

buffer_used (*atributo de xml.parsers.expat.xmlparser*), 1249
BufferedIOBase (*clase en io*), 662
BufferedProtocol (*clase en asyncio*), 984
BufferedRandom (*clase en io*), 666
BufferedReader (*clase en io*), 665
BufferedRWPair (*clase en io*), 666
BufferedWriter (*clase en io*), 665
BufferError, 102
BufferingFormatter (*clase en logging*), 721
BufferingHandler (*clase en logging.handlers*), 752
BufferTooShort, 843
bufsize() (*método de ossaudiodev.oss_audio_device*), 2058
BUILD_CONST_KEY_MAP (*opcode*), 1949
BUILD_LIST (*opcode*), 1948
BUILD_MAP (*opcode*), 1949
build_opener() (*en el módulo urllib.request*), 1271
BUILD_SET (*opcode*), 1949
BUILD_SLICE (*opcode*), 1952
BUILD_STRING (*opcode*), 1949
BUILD_TUPLE (*opcode*), 1948
built-in
    types, 33
builtin_module_names (*en el módulo sys*), 1751
BuiltinFunctionType (*en el módulo types*), 282
BuiltinImporter (*clase en importlib.machinery*), 1874
BuiltinMethodType (*en el módulo types*), 282
builtins
    módulo, 1773
buscador, **2083**
buscador basado en ruta, **2089**
buscador de entradas de ruta, **2089**
ButtonBox (*clase en tkinter.tix*), 1494
buttonbox() (*método de tkinter.simpledialog.Dialog*), 1470
bye() (*en el módulo turtle*), 1436
byref() (*en el módulo ctypes*), 812
bytearray
    formatting, 72
    interpolation, 72
    methods, 62
    objeto, 44, 59, 61
bytearray (*clase incorporada*), 61
bytecode, **2081**
byte-code
    file, 1934, 2009
Bytecode (*clase en dis*), 1940
BYTECODE_SUFFIXES (*en el módulo importlib.machinery*), 1874
Bytecode.codeobj (*en el módulo dis*), 1941
Bytecode.first_line (*en el módulo dis*), 1941
BytecodeTestCase (*clase en test.support.bytecode_helper*), 1679
byteorder (*en el módulo sys*), 1751
bytes
    formatting, 72
    interpolation, 72
    methods, 62
    objeto, 59
    str (*built-in class*), 48
bytes (*atributo de uuid.UUID*), 1330
bytes (*clase incorporada*), 59
bytes_le (*atributo de uuid.UUID*), 1330
BytesFeedParser (*clase en email.parser*), 1103
BytesGenerator (*clase en email.generator*), 1106
BytesHeaderParser (*clase en email.parser*), 1104
BytesIO (*clase en io*), 664
BytesParser (*clase en email.parser*), 1104
ByteString (*clase en collections.abc*), 260
ByteString (*clase en typing*), 1536
byteswap() (*en el módulo audioop*), 1995
byteswap() (*método de array.array*), 271
BytesWarning, 110
bz2
    módulo, 520
BZ2Compressor (*clase en bz2*), 521
BZ2Decompressor (*clase en bz2*), 521
BZ2File (*clase en bz2*), 520

C

-c
    traceopción de línea de comando, 1717
    unittestopción de línea de comando, 1574
    zipappopción de línea de comando, 1743
C
    language, 35
    structures, 167
-C
    traceopción de línea de comando, 1717
-c <tarfile> <source1> ... <sourceN>
    tarfileopción de línea de comando, 551
-c <zipfile> <source1> ... <sourceN>
    zipfileopción de línea de comando, 538
C14NWriterTarget (*clase en xml.etree.ElementTree*), 1215
c_bool (*clase en ctypes*), 817
C_BUILTIN (*en el módulo imp*), 2013
c_byte (*clase en ctypes*), 816
c_char (*clase en ctypes*), 816
c_char_p (*clase en ctypes*), 816
c_contiguous (*atributo de memoryview*), 80
c_double (*clase en ctypes*), 816
C_EXTENSION (*en el módulo imp*), 2013
c_float (*clase en ctypes*), 816
c_int (*clase en ctypes*), 816
c_int8 (*clase en ctypes*), 816
c_int16 (*clase en ctypes*), 816
c_int32 (*clase en ctypes*), 816

c_int64 (*clase en ctypes*), 816
c_long (*clase en ctypes*), 817
c_longdouble (*clase en ctypes*), 816
c_longlong (*clase en ctypes*), 817
c_short (*clase en ctypes*), 817
c_size_t (*clase en ctypes*), 817
c_ssize_t (*clase en ctypes*), 817
c_ubyte (*clase en ctypes*), 817
c_uint (*clase en ctypes*), 817
c_uint8 (*clase en ctypes*), 817
c_uint16 (*clase en ctypes*), 817
c_uint32 (*clase en ctypes*), 817
c_uint64 (*clase en ctypes*), 817
c_ulong (*clase en ctypes*), 817
c_ulonglong (*clase en ctypes*), 817
c_ushort (*clase en ctypes*), 817
c_void_p (*clase en ctypes*), 817
c_wchar (*clase en ctypes*), 817
c_wchar_p (*clase en ctypes*), 817
CAB (*clase en msilib*), 2018
cache() (*en el módulo functools*), 396
cache_from_source() (*en el módulo imp*), 2011
cache_from_source() (*en el módulo importlib.util*), 1878
cached (*atributo de importlib.machinery.ModuleSpec*), 1878
cached_property() (*en el módulo functools*), 397
CacheFTPHandler (*clase en urllib.request*), 1275
cadena con triple comilla, **2091**
calcobjsize() (*en el módulo test.support*), 1673
calcsize() (*en el módulo struct*), 168
calcvobjsize() (*en el módulo test.support*), 1673
calendar
    módulo, 234
Calendar (*clase en calendar*), 234
calendar() (*en el módulo calendar*), 238
Call (*clase en ast*), 1897
call() (*en el módulo subprocess*), 902
call() (*en el módulo unittest.mock*), 1633
call_args (*atributo de unittest.mock.Mock*), 1609
call_args_list (*atributo de unittest.mock.Mock*), 1610
call_at() (*método de asyncio.loop*), 957
call_count (*atributo de unittest.mock.Mock*), 1608
call_exception_handler() (*método de asyncio.loop*), 968
CALL_FUNCTION (*opcode*), 1951
CALL_FUNCTION_EX (*opcode*), 1951
CALL_FUNCTION_KW (*opcode*), 1951
call_later() (*método de asyncio.loop*), 957
call_list() (*método de unittest.mock.call*), 1633
CALL_METHOD (*opcode*), 1951
call_soon() (*método de asyncio.loop*), 956
call_soon_threadsafe() (*método de asyncio.loop*), 956
call_tracing() (*en el módulo sys*), 1751
callable, **2081**
Callable (*clase en collections.abc*), 260
Callable (*en el módulo typing*), 1521
callable()
    función incorporada, 8
CallableProxyType (*en el módulo weakref*), 276
callback (*atributo de optparse.Option*), 2042
callback() (*método de contextlib.ExitStack*), 1803
callback_args (*atributo de optparse.Option*), 2042
callback_kwargs (*atributo de optparse.Option*), 2042
callbacks (*en el módulo gc*), 1826
called (*atributo de unittest.mock.Mock*), 1607
CalledProcessError, 891
CAN_BCM (*en el módulo socket*), 1013
can_change_color() (*en el módulo curses*), 757
can_fetch() (*método de urllib.robotparser.RobotFileParser*), 1298
CAN_ISOTP (*en el módulo socket*), 1014
CAN_J1939 (*en el módulo socket*), 1014
CAN_RAW_FD_FRAMES (*en el módulo socket*), 1013
CAN_RAW_JOIN_FILTERS (*en el módulo socket*), 1013
can_symlink() (*en el módulo test.support.os_helper*), 1681
can_write_eof() (*método de asyncio.StreamWriter*), 937
can_write_eof() (*método de asyncio.WriteTransport*), 981
can_xattr() (*en el módulo test.support.os_helper*), 1681
cancel() (*método de asyncio.Future*), 977
cancel() (*método de asyncio.Handle*), 970
cancel() (*método de asyncio.Task*), 931
cancel() (*método de concurrent.futures.Future*), 886
cancel() (*método de sched.scheduler*), 909
cancel() (*método de threading.Timer*), 832
cancel() (*método de tkinter.dnd.DndHandler*), 1474
cancel_command() (*método de tkinter.filedialog.FileDialog*), 1471
cancel_dump_traceback_later() (*en el módulo faulthandler*), 1695
cancel_join_thread() (*método de multiprocessing.Queue*), 846
cancelled() (*método de asyncio.Future*), 977
cancelled() (*método de asyncio.Handle*), 970
cancelled() (*método de asyncio.Task*), 932
cancelled() (*método de concurrent.futures.Future*), 886
CancelledError, 888, 952
CannotSendHeader, 1303
CannotSendRequest, 1303
canonic() (*método de bdb.Bdb*), 1691
canonical() (*método de decimal.Context*), 345
canonical() (*método de decimal.Decimal*), 337
canonicalize() (*en el módulo xml.etree.ElementTree*), 1206
capa() (*método de poplib.POP3*), 1315
capitalize() (*método de bytearray*), 68
capitalize() (*método de bytes*), 68

capitalize() (*método de str*), 49
captured_stderr() (*en el módulo test.support*), 1671
captured_stdin() (*en el módulo test.support*), 1671
captured_stdout() (*en el módulo test.support*), 1671
captureWarnings() (*en el módulo logging*), 730
capwords() (*en el módulo string*), 124
cargador, **2087**
casefold() (*método de str*), 49
cast() (*en el módulo ctypes*), 813
cast() (*en el módulo typing*), 1541
cast() (*método de memoryview*), 78
cat() (*en el módulo nis*), 2021
--catch
    unittestopción de línea de comando, 1574
catch_threading_exception() (*en el módulo test.support.threading_helper*), 1679
catch_unraisable_exception() (*en el módulo test.support*), 1674
catch_warnings (*clase en warnings*), 1785
category() (*en el módulo unicodedata*), 159
cbreak() (*en el módulo curses*), 757
ccc() (*método de ftplib.FTP_TLS*), 1313
C-contiguous, 2082
cdf() (*método de statistics.NormalDist*), 377
CDLL (*clase en ctypes*), 807
ceil() (*en el módulo math*), 321
ceil() (*in module math*), 36
CellType (*en el módulo types*), 282
center() (*método de bytearray*), 65
center() (*método de bytes*), 65
center() (*método de str*), 49
CERT_NONE (*en el módulo ssl*), 1040
CERT_OPTIONAL (*en el módulo ssl*), 1040
CERT_REQUIRED (*en el módulo ssl*), 1040
cert_store_stats() (*método de ssl.SSLContext*), 1051
cert_time_to_seconds() (*en el módulo ssl*), 1038
CertificateError, 1037
certificates, 1058
CFUNCTYPE() (*en el módulo ctypes*), 810
cget() (*método de tkinter.font.Font*), 1469
cgi
    módulo, 1997
CGI
    debugging, 2003
    exceptions, 2004
    protocol, 1997
    security, 2002
    tracebacks, 2004
cgi_directories (*atributo de http.server.CGIHTTPRequestHandler*), 1346
CGIHandler (*clase en wsgiref.handlers*), 1265
CGIHTTPRequestHandler (*clase en http.server*), 1346
cgitb
    módulo, 2004
CGIXMLRPCRequestHandler (*clase en xmlrpc.server*), 1367
chain() (*en el módulo itertools*), 384
chaining
    comparisons, 34
ChainMap (*clase en collections*), 239
ChainMap (*clase en typing*), 1535
change_cwd() (*en el módulo test.support.os_helper*), 1681
CHANNEL_BINDING_TYPES (*en el módulo ssl*), 1045
channel_class (*atributo de smtpd.SMTPServer*), 2062
channels() (*método de ossaudiodev.oss_audio_device*), 2057
CHAR_MAX (*en el módulo locale*), 1406
character, 158
CharacterDataHandler() (*método de xml.parsers.expat.xmlparser*), 1251
characters() (*método de xml.sax.handler.ContentHandler*), 1240
characters_written (*atributo de BlockingIOError*), 108
Charset (*clase en email.charset*), 1145
charset() (*método de gettext.NullTranslations*), 1395
chdir() (*en el módulo os*), 620
check (*atributo de lzma.LZMADecompressor*), 527
check() (*en el módulo tabnanny*), 1932
check() (*método de imaplib.IMAP4*), 1318
check__all__() (*en el módulo test.support*), 1675
check_call() (*en el módulo subprocess*), 902
check_disallow_instantiation() (*en el módulo test.support*), 1676
check_free_after_iterating() (*en el módulo test.support*), 1675
check_hostname (*atributo de ssl.SSLContext*), 1056
check_impl_detail() (*en el módulo test.support*), 1671
check_no_resource_warning() (*en el módulo test.support.warnings_helper*), 1683
check_output() (*en el módulo subprocess*), 903
check_output() (*método de doctest.OutputChecker*), 1567
check_returncode() (*método de subprocess.CompletedProcess*), 890
check_syntax_error() (*en el módulo test.support*), 1674
check_syntax_warning() (*en el módulo test.support.warnings_helper*), 1683
check_unused_args() (*método de string.Formatter*), 115
check_warnings() (*en el módulo test.support.warnings_helper*), 1683
checkbox() (*método de msilib.Dialog*), 2020
checkcache() (*en el módulo linecache*), 454

CHECKED_HASH (*atributo de py_compile.PycInvalidationMode*), 1935
checkfuncname() (*en el módulo bdb*), 1694
CheckList (*clase en tkinter.tix*), 1495
checksizeof() (*en el módulo test.support*), 1673
checksum
    Cyclic Redundancy Check, 514
chflags() (*en el módulo os*), 621
chgat() (*método de curses.window*), 765
childNodes (*atributo de xml.dom.Node*), 1221
ChildProcessError, 108
children (*atributo de pyclbr.Class*), 1934
children (*atributo de pyclbr.Function*), 1933
children (*atributo de tkinter.Tk*), 1457
chmod() (*en el módulo os*), 621
chmod() (*método de pathlib.Path*), 424
choice() (*en el módulo random*), 362
choice() (*en el módulo secrets*), 597
choices (*atributo de optparse.Option*), 2042
choices() (*en el módulo random*), 362
Chooser (*clase en tkinter.colorchooser*), 1468
chown() (*en el módulo os*), 622
chown() (*en el módulo shutil*), 458
chr()
    función incorporada, 8
chroot() (*en el módulo os*), 622
chunk
    módulo, 2005
Chunk (*clase en chunk*), 2005
cipher
    DES, 2006
cipher() (*método de ssl.SSLSocket*), 1049
circle() (*en el módulo turtle*), 1416
CIRCUMFLEX (*en el módulo token*), 1925
CIRCUMFLEXEQUAL (*en el módulo token*), 1926
Clamped (*clase en decimal*), 349
clase, **2081**
clase base abstracta, **2079**
clase de nuevo estilo, **2088**
Class (*clase en symtable*), 1923
ClassDef (*clase en ast*), 1916
classmethod()
    función incorporada, 8
ClassMethodDescriptorType (*en el módulo types*), 282
ClassVar (*en el módulo typing*), 1523
CLD_CONTINUED (*en el módulo os*), 650
CLD_DUMPED (*en el módulo os*), 650
CLD_EXITED (*en el módulo os*), 650
CLD_KILLED (*en el módulo os*), 650
CLD_STOPPED (*en el módulo os*), 650
CLD_TRAPPED (*en el módulo os*), 650
clean() (*método de mailbox.Maildir*), 1165
cleandoc() (*en el módulo inspect*), 1832
CleanImport (*clase en test.support.import_helper*), 1683
clear (*pdb command*), 1700
Clear Breakpoint, 1501
clear() (*en el módulo turtle*), 1424
clear() (*método de asyncio.Event*), 942
clear() (*método de collections.deque*), 245
clear() (*método de curses.window*), 765
clear() (*método de dict*), 85
clear() (*método de email.message.EmailMessage*), 1102
clear() (*método de frozenset*), 83
clear() (*método de http.cookiejar.CookieJar*), 1352
clear() (*método de mailbox.Mailbox*), 1164
clear() (*método de threading.Event*), 832
clear() (*método de xml.etree.ElementTree.Element*), 1211
clear() (*sequence method*), 44
clear_all_breaks() (*método de bdb.Bdb*), 1693
clear_all_file_breaks() (*método de bdb.Bdb*), 1693
clear_bpbynumber() (*método de bdb.Bdb*), 1693
clear_break() (*método de bdb.Bdb*), 1693
clear_cache() (*en el módulo filecmp*), 444
clear_cache() (*método de clase de zoneinfo.ZoneInfo*), 231
clear_content() (*método de email.message.EmailMessage*), 1102
clear_flags() (*método de decimal.Context*), 344
clear_frames() (*en el módulo traceback*), 1817
clear_history() (*en el módulo readline*), 163
clear_session_cookies() (*método de http.cookiejar.CookieJar*), 1353
clear_traces() (*en el módulo tracemalloc*), 1724
clear_traps() (*método de decimal.Context*), 344
clearcache() (*en el módulo linecache*), 454
ClearData() (*método de msilib.Record*), 2018
clearok() (*método de curses.window*), 765
clearscreen() (*en el módulo turtle*), 1431
clearstamp() (*en el módulo turtle*), 1417
clearstamps() (*en el módulo turtle*), 1417
Client() (*en el módulo multiprocessing.connection*), 864
client_address (*atributo de http.server.BaseHTTPRequestHandler*), 1341
CLOCK_BOOTTIME (*en el módulo time*), 678
clock_getres() (*en el módulo time*), 671
clock_gettime() (*en el módulo time*), 671
clock_gettime_ns() (*en el módulo time*), 671
CLOCK_HIGHRES (*en el módulo time*), 678
CLOCK_MONOTONIC (*en el módulo time*), 678
CLOCK_MONOTONIC_RAW (*en el módulo time*), 678
CLOCK_PROCESS_CPUTIME_ID (*en el módulo time*), 678
CLOCK_PROF (*en el módulo time*), 678
CLOCK_REALTIME (*en el módulo time*), 679
clock_settime() (*en el módulo time*), 672
clock_settime_ns() (*en el módulo time*), 672
CLOCK_TAI (*en el módulo time*), 678
CLOCK_THREAD_CPUTIME_ID (*en el módulo time*), 678
CLOCK_UPTIME (*en el módulo time*), 678

CLOCK_UPTIME_RAW (*en el módulo time*), 679
clone() (*en el módulo turtle*), 1429
clone() (*método de email.generator.BytesGenerator*), 1107
clone() (*método de email.generator.Generator*), 1108
clone() (*método de email.policy.Policy*), 1111
clone() (*método de pipes.Template*), 2060
cloneNode() (*método de xml.dom.Node*), 1222
close() (*en el módulo fileinput*), 438
close() (*en el módulo os*), 609
close() (*en el módulo socket*), 1017
close() (*método de aifc.aifc*), 1986
close() (*método de asyncio.AbstractChildWatcher*), 994
close() (*método de asyncio.BaseTransport*), 980
close() (*método de asyncio.loop*), 955
close() (*método de asyncio.Server*), 971
close() (*método de asyncio.StreamWriter*), 937
close() (*método de asyncio.SubprocessTransport*), 983
close() (*método de asyncore.dispatcher*), 1992
close() (*método de chunk.Chunk*), 2006
close() (*método de contextlib.ExitStack*), 1803
close() (*método de dbm.dumb.dumbdbm*), 490
close() (*método de dbm.gnu.gdbm*), 489
close() (*método de dbm.ndbm.ndbm*), 489
close() (*método de email.parser.BytesFeedParser*), 1103
close() (*método de ftplib.FTP*), 1313
close() (*método de html.parser.HTMLParser*), 1193
close() (*método de http.client.HTTPConnection*), 1305
close() (*método de imaplib.IMAP4*), 1319
close() (*método de io.IOBase*), 661
close() (*método de logging.FileHandler*), 743
close() (*método de logging.Handler*), 719
close() (*método de logging.handlers.MemoryHandler*), 752
close() (*método de logging.handlers.NTEventLogHandler*), 751
close() (*método de logging.handlers.SocketHandler*), 747
close() (*método de logging.handlers.SysLogHandler*), 749
close() (*método de mailbox.Mailbox*), 1164
close() (*método de mailbox.Maildir*), 1166
close() (*método de mailbox.MH*), 1168
close() (*método de mmap.mmap*), 1089
Close() (*método de msilib.Database*), 2016
Close() (*método de msilib.View*), 2017
close() (*método de multiprocessing.connection.Connection*), 848
close() (*método de multiprocessing.connection.Listener*), 865
close() (*método de multiprocessing.pool.Pool*), 863
close() (*método de multiprocessing.Process*), 843
close() (*método de multiprocessing.Queue*), 845
close() (*método de multiprocessing.shared_memory.SharedMemory*), 878
close() (*método de multiprocessing.SimpleQueue*), 846
close() (*método de ossaudiodev.oss_audio_device*), 2056
close() (*método de ossaudiodev.oss_mixer_device*), 2059
close() (*método de os.scandir*), 628
close() (*método de select.devpoll*), 1070
close() (*método de select.epoll*), 1071
close() (*método de select.kqueue*), 1073
close() (*método de selectors.BaseSelector*), 1076
close() (*método de shelve.Shelf*), 483
close() (*método de socket.socket*), 1022
close() (*método de sqlite3.Connection*), 496
close() (*método de sqlite3.Cursor*), 503
close() (*método de sunau.AU_read*), 2067
close() (*método de sunau.AU_write*), 2068
close() (*método de tarfile.TarFile*), 545
close() (*método de telnetlib.Telnet*), 2070
close() (*método de urllib.request.BaseHandler*), 1277
close() (*método de wave.Wave_read*), 1388
close() (*método de wave.Wave_write*), 1389
Close() (*método de winreg.PyHKEY*), 1967
close() (*método de xml.etree.ElementTree.TreeBuilder*), 1214
close() (*método de xml.etree.ElementTree.XMLParser*), 1216
close() (*método de xml.etree.ElementTree.XMLPullParser*), 1217
close() (*método de xml.sax.xmlreader.IncrementalParser*), 1245
close() (*método de zipfile.ZipFile*), 531
close_connection (*atributo de http.server.BaseHTTPRequestHandler*), 1341
close_when_done() (*método de asynchat.async_chat*), 1988
closed (*atributo de http.client.HTTPResponse*), 1306
closed (*atributo de io.IOBase*), 661
closed (*atributo de mmap.mmap*), 1089
closed (*atributo de ossaudiodev.oss_audio_device*), 2058
closed (*atributo de select.devpoll*), 1070
closed (*atributo de select.epoll*), 1071
closed (*atributo de select.kqueue*), 1073
CloseKey() (*en el módulo winreg*), 1959
closelog() (*en el módulo syslog*), 1984
closerange() (*en el módulo os*), 609
closing() (*en el módulo contextlib*), 1798
clrtobot() (*método de curses.window*), 765
clrtoeol() (*método de curses.window*), 765
cmath
    módulo, 328
cmd
    módulo, 1443, 1696
cmd (*atributo de subprocess.CalledProcessError*), 891

cmd (*atributo de subprocess.TimeoutExpired*), 891
Cmd (*clase en cmd*), 1443
cmdloop() (*método de cmd.Cmd*), 1443
cmdqueue (*atributo de cmd.Cmd*), 1445
cmp() (*en el módulo filecmp*), 444
cmp_op (*en el módulo dis*), 1953
cmp_to_key() (*en el módulo functools*), 397
cmpfiles() (*en el módulo filecmp*), 444
CMSG_LEN() (*en el módulo socket*), 1020
CMSG_SPACE() (*en el módulo socket*), 1020
CO_ASYNC_GENERATOR (*en el módulo inspect*), 1843
CO_COROUTINE (*en el módulo inspect*), 1843
CO_GENERATOR (*en el módulo inspect*), 1843
CO_ITERABLE_COROUTINE (*en el módulo inspect*), 1843
CO_NESTED (*en el módulo inspect*), 1843
CO_NEWLOCALS (*en el módulo inspect*), 1843
CO_NOFREE (*en el módulo inspect*), 1843
CO_OPTIMIZED (*en el módulo inspect*), 1843
CO_VARARGS (*en el módulo inspect*), 1843
CO_VARKEYWORDS (*en el módulo inspect*), 1843
code
    módulo, 1849
code (*atributo de SystemExit*), 106
code (*atributo de urllib.error.HTTPError*), 1297
code (*atributo de urllib.response.addinfourl*), 1288
code (*atributo de xml.etree.ElementTree.ParseError*), 1217
code (*atributo de xml.parsers.expat.ExpatError*), 1252
code object, 95, 485
code_info() (*en el módulo dis*), 1941
CodecInfo (*clase en codecs*), 174
codecs
    módulo, 174
Codecs, 174
    decode, 174
    encode, 174
coded_value (*atributo de http.cookies.Morsel*), 1348
codeop
    módulo, 1851
codepoint2name (*en el módulo html.entities*), 1196
codes (*en el módulo xml.parsers.expat.errors*), 1254
CODESET (*en el módulo locale*), 1402
CodeType (*clase en types*), 281
codificación de la configuración regional, **2086**
codificación de texto, **2091**
codificación del sistema de archivos y manejador de errores, **2083**
coerción, **2081**
col_offset (*atributo de ast.AST*), 1892
collapse_addresses() (*en el módulo ipaddress*), 1385
collapse_rfc2231_value() (*en el módulo email.utils*), 1150
collect() (*en el módulo gc*), 1824
collect_incoming_data() (*método de asynchat.async_chat*), 1988
Collection (*clase en collections.abc*), 260
Collection (*clase en typing*), 1536
collections
    módulo, 239
collections.abc
    módulo, 257
colno (*atributo de json.JSONDecodeError*), 1158
colno (*atributo de re.error*), 135
COLON (*en el módulo token*), 1924
COLONEQUAL (*en el módulo token*), 1926
color() (*en el módulo turtle*), 1423
color_content() (*en el módulo curses*), 757
color_pair() (*en el módulo curses*), 757
colormode() (*en el módulo turtle*), 1435
colorsys
    módulo, 1389
COLS, 763
column() (*método de tkinter.ttk.Treeview*), 1487
COLUMNS, 763
columns (*atributo de os.terminal_size*), 618
comb() (*en el módulo math*), 321
combinations() (*en el módulo itertools*), 384
combinations_with_replacement() (*en el módulo itertools*), 385
combine() (*método de clase de datetime.datetime*), 205
combining() (*en el módulo unicodedata*), 159
ComboBox (*clase en tkinter.tix*), 1494
Combobox (*clase en tkinter.ttk*), 1479
COMMA (*en el módulo token*), 1925
command (*atributo de http.server.BaseHTTPRequestHandler*), 1341
CommandCompiler (*clase en codeop*), 1852
commands (*pdb command*), 1700
comment (*atributo de http.cookiejar.Cookie*), 1357
comment (*atributo de zipfile.ZipFile*), 534
comment (*atributo de zipfile.ZipInfo*), 537
COMMENT (*en el módulo token*), 1927
Comment() (*en el módulo xml.etree.ElementTree*), 1206
comment() (*método de xml.etree.ElementTree.TreeBuilder*), 1215
comment() (*método de xml.sax.handler.LexicalHandler*), 1241
comment_url (*atributo de http.cookiejar.Cookie*), 1357
commenters (*atributo de shlex.shlex*), 1450
CommentHandler() (*método de xml.parsers.expat.xmlparser*), 1251
commit() (*método de msilib.CAB*), 2018
Commit() (*método de msilib.Database*), 2016
commit() (*método de sqlite3.Connection*), 496
common (*atributo de filecmp.dircmp*), 445
Common Gateway Interface, 1997
common_dirs (*atributo de filecmp.dircmp*), 445
common_files (*atributo de filecmp.dircmp*), 445
common_funny (*atributo de filecmp.dircmp*), 445
common_types (*en el módulo mimetypes*), 1180
commonpath() (*en el módulo os.path*), 432

commonprefix() (*en el módulo os.path*), 432
communicate() (*método de asyncio.subprocess.Process*), 947
communicate() (*método de subprocess.Popen*), 898
--compact
    json.toolopción de línea de comando, 1161
Compare (*clase en ast*), 1897
compare() (*método de decimal.Context*), 345
compare() (*método de decimal.Decimal*), 337
compare() (*método de difflib.Differ*), 152
compare_digest() (*en el módulo hmac*), 597
compare_digest() (*en el módulo secrets*), 598
compare_networks() (*método de ipaddress.IPv4Network*), 1380
compare_networks() (*método de ipaddress.IPv6Network*), 1382
COMPARE_OP (*opcode*), 1949
compare_signal() (*método de decimal.Context*), 345
compare_signal() (*método de decimal.Decimal*), 337
compare_to() (*método de tracemalloc.Snapshot*), 1726
compare_total() (*método de decimal.Context*), 345
compare_total() (*método de decimal.Decimal*), 337
compare_total_mag() (*método de decimal.Context*), 345
compare_total_mag() (*método de decimal.Decimal*), 338
comparing
    objects, 34
comparison
    operator, 34
COMPARISON_FLAGS (*en el módulo doctest*), 1556
comparisons
    chaining, 34
Compat32 (*clase en email.policy*), 1115
compat32 (*en el módulo email.policy*), 1116
compile
    función incorporada, 96, 281
Compile (*clase en codeop*), 1852
compile()
    función incorporada, 9
compile() (*en el módulo py_compile*), 1934
compile() (*en el módulo re*), 131
compile_command() (*en el módulo code*), 1850
compile_command() (*en el módulo codeop*), 1851
compile_dir() (*en el módulo compileall*), 1937
compile_file() (*en el módulo compileall*), 1938
compile_path() (*en el módulo compileall*), 1939
compileall
    módulo, 1936
compileallopción de línea de comando
    -b, 1936
    -d destdir, 1936
    directory ..., 1936
    -e dir, 1937
    -f, 1936
    file ..., 1936
    --hardlink-dupes, 1937
    -i list, 1936
    --invalidation-mode [timestamp|checked-hash|unchecked-hash], 1937
    -j N, 1937
    -l, 1936
    -o level, 1937
    -p prepend_prefix, 1936
    -q, 1936
    -r, 1936
    -s strip_prefix, 1936
    -x regex, 1936
complete() (*método de rlcompleter.Completer*), 166
complete_statement() (*en el módulo sqlite3*), 494
completedefault() (*método de cmd.Cmd*), 1444
CompletedProcess (*clase en subprocess*), 890
complex
    función incorporada, 35
Complex (*clase en numbers*), 317
complex (*clase incorporada*), 9
complex number
    literals, 35
    objeto, 35
comprehension (*clase en ast*), 1900
comprensión de conjuntos, **2091**
comprensión de diccionarios, **2082**
comprensión de listas, **2087**
--compress
    zipappopción de línea de comando, 1743
compress() (*en el módulo bz2*), 522
compress() (*en el módulo gzip*), 518
compress() (*en el módulo itertools*), 385
compress() (*en el módulo lzma*), 527
compress() (*en el módulo zlib*), 513
compress() (*método de bz2.BZ2Compressor*), 521
compress() (*método de lzma.LZMACompressor*), 526
compress() (*método de zlib.Compress*), 515
compress_size (*atributo de zipfile.ZipInfo*), 537
compress_type (*atributo de zipfile.ZipInfo*), 537
compressed (*atributo de ipaddress.IPv4Address*), 1374
compressed (*atributo de ipaddress.IPv4Network*), 1379
compressed (*atributo de ipaddress.IPv6Address*), 1375
compressed (*atributo de ipaddress.IPv6Network*), 1381
compression() (*método de ssl.SSLSocket*), 1049
CompressionError, 541
compressobj() (*en el módulo zlib*), 514
COMSPEC, 649, 894
concat() (*en el módulo operator*), 408

Concatenate (*en el módulo typing*), 1522
concatenation
    operation, 42
concurrent.futures
    módulo, 882
cond (*atributo de bdb.Breakpoint*), 1690
Condition (*clase en asyncio*), 943
Condition (*clase en multiprocessing*), 850
Condition (*clase en threading*), 829
condition (*pdb command*), 1700
condition() (*método de msilib.Control*), 2020
Condition() (*método de multiprocessing.managers.SyncManager*), 857
config() (*método de tkinter.font.Font*), 1469
configparser
    módulo, 562
ConfigParser (*clase en configparser*), 575
configuration
    file, 562
    file, debugger, 1699
    file, path, 1844
configuration information, 1769
configure() (*método de tkinter.ttk.Style*), 1490
configure_mock() (*método de unittest.mock.Mock*), 1607
confstr() (*en el módulo os*), 654
confstr_names (*en el módulo os*), 654
conjugate() (*complex number method*), 35
conjugate() (*método de decimal.Decimal*), 338
conjugate() (*método de numbers.Complex*), 317
conn (*atributo de smtpd.SMTPChannel*), 2063
connect() (*en el módulo sqlite3*), 493
connect() (*método de asyncore.dispatcher*), 1992
connect() (*método de ftplib.FTP*), 1310
connect() (*método de http.client.HTTPConnection*), 1305
connect() (*método de multiprocessing.managers.BaseManager*), 855
connect() (*método de smtplib.SMTP*), 1325
connect() (*método de socket.socket*), 1023
connect_accepted_socket() (*método de asyncio.loop*), 962
connect_ex() (*método de socket.socket*), 1023
connect_read_pipe() (*método de asyncio.loop*), 966
connect_write_pipe() (*método de asyncio.loop*), 966
connection (*atributo de sqlite3.Cursor*), 503
Connection (*clase en multiprocessing.connection*), 848
Connection (*clase en sqlite3*), 496
connection_lost() (*método de asyncio.BaseProtocol*), 984
connection_made() (*método de asyncio.BaseProtocol*), 984
ConnectionAbortedError, 108
ConnectionError, 108
ConnectionRefusedError, 108
ConnectionResetError, 108
ConnectRegistry() (*en el módulo winreg*), 1959
const (*atributo de optparse.Option*), 2042
Constant (*clase en ast*), 1893
constructor() (*en el módulo copyreg*), 481
consumed (*atributo de asyncio.LimitOverrunError*), 953
contador de referencias, **2090**
container
    iteration over, 41
Container (*clase en collections.abc*), 260
Container (*clase en typing*), 1536
contains() (*en el módulo operator*), 408
CONTAINS_OP (*opcode*), 1949
content type
    MIME, 1179
content_disposition (*atributo de email.headerregistry.ContentDispositionHeader*), 1120
content_manager (*atributo de email.policy.EmailPolicy*), 1113
content_type (*atributo de email.headerregistry.ContentTypeHeader*), 1120
ContentDispositionHeader (*clase en email.headerregistry*), 1120
ContentHandler (*clase en xml.sax.handler*), 1236
ContentManager (*clase en email.contentmanager*), 1123
contents (*atributo de ctypes._Pointer*), 820
contents() (*en el módulo importlib.resources*), 1873
contents() (*método de importlib.abc.ResourceReader*), 1868
ContentTooShortError, 1298
ContentTransferEncoding (*clase en email.headerregistry*), 1120
ContentTypeHeader (*clase en email.headerregistry*), 1120
context (*atributo de ssl.SSLSocket*), 1050
Context (*clase en contextvars*), 914
Context (*clase en decimal*), 343
context management protocol, 88
context manager, 88
context_diff() (*en el módulo difflib*), 146
ContextDecorator (*clase en contextlib*), 1800
contextlib
    módulo, 1796
ContextManager (*clase en typing*), 1540
contextmanager() (*en el módulo contextlib*), 1796
ContextVar (*clase en contextvars*), 913
contextvars
    módulo, 912
contiguo, **2082**
contiguous (*atributo de memoryview*), 81
Continue (*clase en ast*), 1906
continue (*pdb command*), 1701
Control (*clase en msilib*), 2020
Control (*clase en tkinter.tix*), 1494

control() (*método de msilib.Dialog*), 2020
control() (*método de select.kqueue*), 1073
controlnames (*en el módulo curses.ascii*), 777
controls() (*método de ossaudiodev.oss_mixer_device*), 2059
ConversionError, 2075
conversions
    numeric, 36
convert_arg_line_to_args() (*método de argparse.ArgumentParser*), 709
convert_field() (*método de string.Formatter*), 115
Cookie (*clase en http.cookiejar*), 1351
CookieError, 1347
cookiejar (*atributo de urllib.request.HTTPCookieProcessor*), 1279
CookieJar (*clase en http.cookiejar*), 1350
CookiePolicy (*clase en http.cookiejar*), 1351
Coordinated Universal Time, 670
copy
    módulo, 286, 481
    protocol, 473
Copy, 1501
copy() (*en el módulo copy*), 286
copy() (*en el módulo multiprocessing.sharedctypes*), 854
copy() (*en el módulo shutil*), 456
copy() (*método de collections.deque*), 245
copy() (*método de contextvars.Context*), 915
copy() (*método de decimal.Context*), 344
copy() (*método de dict*), 85
copy() (*método de frozenset*), 82
copy() (*método de hashlib.hash*), 587
copy() (*método de hmac.HMAC*), 596
copy() (*método de http.cookies.Morsel*), 1349
copy() (*método de imaplib.IMAP4*), 1319
copy() (*método de pipes.Template*), 2061
copy() (*método de tkinter.font.Font*), 1469
copy() (*método de types.MappingProxyType*), 284
copy() (*método de zlib.Compress*), 515
copy() (*método de zlib.Decompress*), 516
copy() (*sequence method*), 44
copy2() (*en el módulo shutil*), 456
copy_abs() (*método de decimal.Context*), 345
copy_abs() (*método de decimal.Decimal*), 338
copy_context() (*en el módulo contextvars*), 914
copy_decimal() (*método de decimal.Context*), 344
COPY_DICT_WITHOUT_KEYS (*opcode*), 1947
copy_file_range() (*en el módulo os*), 609
copy_location() (*en el módulo ast*), 1919
copy_negate() (*método de decimal.Context*), 345
copy_negate() (*método de decimal.Decimal*), 338
copy_sign() (*método de decimal.Context*), 345
copy_sign() (*método de decimal.Decimal*), 338
copyfile() (*en el módulo shutil*), 454
copyfileobj() (*en el módulo shutil*), 454
copying files, 454
copymode() (*en el módulo shutil*), 455
copyreg
    módulo, 481
copyright (*en el módulo sys*), 1751
copyright (*variable incorporada*), 32
copysign() (*en el módulo math*), 321
copystat() (*en el módulo shutil*), 455
copytree() (*en el módulo shutil*), 456
Coroutine (*clase en collections.abc*), 261
Coroutine (*clase en typing*), 1539
coroutine() (*en el módulo asyncio*), 934
coroutine() (*en el módulo types*), 285
CoroutineType (*en el módulo types*), 281
correlation() (*en el módulo statistics*), 376
corrutina, **2082**
cos() (*en el módulo cmath*), 329
cos() (*en el módulo math*), 325
cosh() (*en el módulo cmath*), 329
cosh() (*en el módulo math*), 326
--count
    traceopción de línea de comando, 1717
count (*atributo de tracemalloc.Statistic*), 1727
count (*atributo de tracemalloc.StatisticDiff*), 1728
count() (*en el módulo itertools*), 385
count() (*método de array.array*), 271
count() (*método de bytearray*), 62
count() (*método de bytes*), 62
count() (*método de collections.deque*), 245
count() (*método de multiprocessing.shared_memory.ShareableList*), 880
count() (*método de str*), 49
count() (*sequence method*), 42
count_diff (*atributo de tracemalloc.StatisticDiff*), 1728
Counter (*clase en collections*), 242
Counter (*clase en typing*), 1535
countOf() (*en el módulo operator*), 408
countTestCases() (*método de unittest.TestCase*), 1588
countTestCases() (*método de unittest.TestSuite*), 1592
covariance() (*en el módulo statistics*), 375
CoverageResults (*clase en trace*), 1718
--coverdir=<dir>
    traceopción de línea de comando, 1717
cProfile
    módulo, 1705
CPU time, 673, 676
cpu_count() (*en el módulo multiprocessing*), 847
cpu_count() (*en el módulo os*), 654
CPython, **2082**
cpython_only() (*en el módulo test.support*), 1673
crawl_delay() (*método de urllib.robotparser.RobotFileParser*), 1298
CRC (*atributo de zipfile.ZipInfo*), 537
crc32() (*en el módulo binascii*), 1187
crc32() (*en el módulo zlib*), 514
crc_hqx() (*en el módulo binascii*), 1187

--create <tarfile> <source1> ... <sourceN>
    tarfileopción de línea de comando, 551
--create <zipfile> <source1> ... <sourceN>
    zipfileopción de línea de comando, 538
create() (*en el módulo venv*), 1738
create() (*método de imaplib.IMAP4*), 1319
create() (*método de venv.EnvBuilder*), 1737
create_aggregate() (*método de sqlite3.Connection*), 497
create_archive() (*en el módulo zipapp*), 1744
create_autospec() (*en el módulo unittest.mock*), 1634
CREATE_BREAKAWAY_FROM_JOB (*en el módulo subprocess*), 902
create_collation() (*método de sqlite3.Connection*), 497
create_configuration() (*método de venv.EnvBuilder*), 1738
create_connection() (*en el módulo socket*), 1016
create_connection() (*método de asyncio.loop*), 958
create_datagram_endpoint() (*método de asyncio.loop*), 960
create_decimal() (*método de decimal.Context*), 344
create_decimal_from_float() (*método de decimal.Context*), 344
create_default_context() (*en el módulo ssl*), 1035
CREATE_DEFAULT_ERROR_MODE (*en el módulo subprocess*), 902
create_empty_file() (*en el módulo test.support.os_helper*), 1681
create_function() (*método de sqlite3.Connection*), 496
create_future() (*método de asyncio.loop*), 958
create_module() (*método de importlib.abc.Loader*), 1866
create_module() (*método de importlib.machinery.ExtensionFileLoader*), 1877
create_module() (*método de zipimport.zipimporter*), 1854
CREATE_NEW_CONSOLE (*en el módulo subprocess*), 901
CREATE_NEW_PROCESS_GROUP (*en el módulo subprocess*), 901
CREATE_NO_WINDOW (*en el módulo subprocess*), 902
create_server() (*en el módulo socket*), 1016
create_server() (*método de asyncio.loop*), 961
create_socket() (*método de asyncore.dispatcher*), 1992
create_stats() (*método de profile.Profile*), 1706
create_string_buffer() (*en el módulo ctypes*), 813
create_subprocess_exec() (*en el módulo asyncio*), 946
create_subprocess_shell() (*en el módulo asyncio*), 946
create_system (*atributo de zipfile.ZipInfo*), 537
create_task() (*en el módulo asyncio*), 924
create_task() (*método de asyncio.loop*), 958
create_unicode_buffer() (*en el módulo ctypes*), 813
create_unix_connection() (*método de asyncio.loop*), 961
create_unix_server() (*método de asyncio.loop*), 962
create_version (*atributo de zipfile.ZipInfo*), 537
createAttribute() (*método de xml.dom.Document*), 1223
createAttributeNS() (*método de xml.dom.Document*), 1223
createComment() (*método de xml.dom.Document*), 1223
createDocument() (*método de xml.dom.DOMImplementation*), 1220
createDocumentType() (*método de xml.dom.DOMImplementation*), 1220
createElement() (*método de xml.dom.Document*), 1223
createElementNS() (*método de xml.dom.Document*), 1223
createfilehandler() (*método de tkinter.Widget.tk*), 1467
CreateKey() (*en el módulo winreg*), 1959
CreateKeyEx() (*en el módulo winreg*), 1959
createLock() (*método de logging.Handler*), 718
createLock() (*método de logging.NullHandler*), 743
createProcessingInstruction() (*método de xml.dom.Document*), 1223
CreateRecord() (*en el módulo msilib*), 2016
createSocket() (*método de logging.handlers.SocketHandler*), 748
createTextNode() (*método de xml.dom.Document*), 1223
credits (*variable incorporada*), 32
critical() (*en el módulo logging*), 727
critical() (*método de logging.Logger*), 717
CRNCYSTR (*en el módulo locale*), 1403
cross() (*en el módulo audioop*), 1995
crypt
    módulo, 1972, 2006
crypt() (*en el módulo crypt*), 2007
crypt(3), 2006, 2007
cryptography, 585
cssclass_month (*atributo de calendar.HTMLCalendar*), 236
cssclass_month_head (*atributo de calendar.HTMLCalendar*), 236
cssclass_noday (*atributo de calendar.HTMLCalendar*), 236

cssclass_year (*atributo de calendar.HTMLCalendar*), 236
cssclass_year_head (*atributo de calendar.HTMLCalendar*), 237
cssclasses (*atributo de calendar.HTMLCalendar*), 236
cssclasses_weekday_head (*atributo de calendar.HTMLCalendar*), 236
csv, 555
    módulo, 555
cte (*atributo de email.headerregistry.ContentTransferEncoding*), 1120
cte_type (*atributo de email.policy.Policy*), 1111
ctermid() (*en el módulo os*), 603
ctime() (*en el módulo time*), 672
ctime() (*método de datetime.date*), 201
ctime() (*método de datetime.datetime*), 211
ctrl() (*en el módulo curses.ascii*), 777
CTRL_BREAK_EVENT (*en el módulo signal*), 1080
CTRL_C_EVENT (*en el módulo signal*), 1080
ctypes
    módulo, 788
curdir (*en el módulo os*), 654
currency() (*en el módulo locale*), 1405
current() (*método de tkinter.ttk.Combobox*), 1479
current_process() (*en el módulo multiprocessing*), 847
current_task() (*en el módulo asyncio*), 931
current_thread() (*en el módulo threading*), 822
CurrentByteIndex (*atributo de xml.parsers.expat.xmlparser*), 1250
CurrentColumnNumber (*atributo de xml.parsers.expat.xmlparser*), 1250
currentframe() (*en el módulo inspect*), 1841
CurrentLineNumber (*atributo de xml.parsers.expat.xmlparser*), 1250
curs_set() (*en el módulo curses*), 757
curses
    módulo, 756
curses.ascii
    módulo, 775
curses.panel
    módulo, 778
curses.textpad
    módulo, 774
Cursor (*clase en sqlite3*), 501
cursor() (*método de sqlite3.Connection*), 496
cursyncup() (*método de curses.window*), 765
Cut, 1501
cwd() (*método de clase de pathlib.Path*), 424
cwd() (*método de ftplib.FTP*), 1312
cycle() (*en el módulo itertools*), 386
CycleError, 315
Cyclic Redundancy Check, 514

D

-d
    gzipopción de línea de comando, 519
-d destdir
    compileallopción de línea de comando, 1936
D_FMT (*en el módulo locale*), 1402
D_T_FMT (*en el módulo locale*), 1402
daemon (*atributo de multiprocessing.Process*), 842
daemon (*atributo de threading.Thread*), 826
data
    packing binary, 167
    tabular, 555
data (*atributo de collections.UserDict*), 255
data (*atributo de collections.UserList*), 256
data (*atributo de collections.UserString*), 256
data (*atributo de select.kevent*), 1074
data (*atributo de selectors.SelectorKey*), 1076
data (*atributo de urllib.request.Request*), 1275
data (*atributo de xml.dom.Comment*), 1225
data (*atributo de xml.dom.ProcessingInstruction*), 1226
data (*atributo de xml.dom.Text*), 1225
data (*atributo de xmlrpc.client.Binary*), 1362
data() (*método de xml.etree.ElementTree.TreeBuilder*), 1215
data_filter() (*en el módulo tarfile*), 549
data_open() (*método de urllib.request.DataHandler*), 1282
data_received() (*método de asyncio.Protocol*), 985
database
    Unicode, 158
DatabaseError, 504
databases, 490
dataclass() (*en el módulo dataclasses*), 1786
dataclasses
    módulo, 1785
DataError, 504
datagram_received() (*método de asyncio.DatagramProtocol*), 986
DatagramHandler (*clase en logging.handlers*), 748
DatagramProtocol (*clase en asyncio*), 984
DatagramRequestHandler (*clase en socketserver*), 1337
DatagramTransport (*clase en asyncio*), 980
DataHandler (*clase en urllib.request*), 1274
date (*clase en datetime*), 199
date() (*método de datetime.datetime*), 207
date() (*método de nntplib.NNTP*), 2027
date_time (*atributo de zipfile.ZipInfo*), 536
date_time_string() (*método de http.server.BaseHTTPRequestHandler*), 1344
DateHeader (*clase en email.headerregistry*), 1119
datetime
    módulo, 193
datetime (*atributo de email.headerregistry.DateHeader*), 1119
datetime (*clase en datetime*), 203
DateTime (*clase en xmlrpc.client*), 1362
day (*atributo de datetime.date*), 200
day (*atributo de datetime.datetime*), 206

day_abbr (*en el módulo calendar*), 238
day_name (*en el módulo calendar*), 238
daylight (*en el módulo time*), 679
Daylight Saving Time, 670
DbfilenameShelf (*clase en shelve*), 484
dbm
    módulo, 486
dbm.dumb
    módulo, 490
dbm.gnu
    módulo, 483, 488
dbm.ndbm
    módulo, 483, 489
dcgettext() (*en el módulo locale*), 1407
debug (*atributo de imaplib.IMAP4*), 1322
debug (*atributo de shlex.shlex*), 1451
debug (*atributo de zipfile.ZipFile*), 534
DEBUG (*en el módulo re*), 130
debug (*pdb command*), 1702
debug() (*en el módulo doctest*), 1569
debug() (*en el módulo logging*), 726
debug() (*método de logging.Logger*), 716
debug() (*método de pipes.Template*), 2060
debug() (*método de unittest.TestCase*), 1581
debug() (*método de unittest.TestSuite*), 1591
DEBUG_BYTECODE_SUFFIXES (*en el módulo importlib.machinery*), 1874
DEBUG_COLLECTABLE (*en el módulo gc*), 1827
DEBUG_LEAK (*en el módulo gc*), 1827
DEBUG_SAVEALL (*en el módulo gc*), 1827
debug_src() (*en el módulo doctest*), 1569
DEBUG_STATS (*en el módulo gc*), 1827
DEBUG_UNCOLLECTABLE (*en el módulo gc*), 1827
debugger, 823, 1500, 1757, 1764
    configuration file, 1699
debugging, 1696
    CGI, 2003
DebuggingServer (*clase en smtpd*), 2062
debuglevel (*atributo de http.client.HTTPResponse*), 1306
DebugRunner (*clase en doctest*), 1569
decimal
    módulo, 331
Decimal (*clase en decimal*), 335
decimal() (*en el módulo unicodedata*), 159
DecimalException (*clase en decimal*), 349
decode
    Codecs, 174
decode (*atributo de codecs.CodecInfo*), 174
decode() (*en el módulo base64*), 1184
decode() (*en el módulo codecs*), 174
decode() (*en el módulo quopri*), 1188
decode() (*en el módulo uu*), 2071
decode() (*método de bytearray*), 63
decode() (*método de bytes*), 63
decode() (*método de codecs.Codec*), 179
decode() (*método de codecs.IncrementalDecoder*), 180
decode() (*método de json.JSONDecoder*), 1156
decode() (*método de xmlrpc.client.Binary*), 1362
decode() (*método de xmlrpc.client.DateTime*), 1362
decode_header() (*en el módulo email.header*), 1144
decode_header() (*en el módulo nntplib*), 2028
decode_params() (*en el módulo email.utils*), 1150
decode_rfc2231() (*en el módulo email.utils*), 1150
decode_source() (*en el módulo importlib.util*), 1879
decodebytes() (*en el módulo base64*), 1184
DecodedGenerator (*clase en email.generator*), 1108
decodestring() (*en el módulo quopri*), 1188
decomposition() (*en el módulo unicodedata*), 159
--decompress
    gzipopción de línea de comando, 519
decompress() (*en el módulo bz2*), 522
decompress() (*en el módulo gzip*), 519
decompress() (*en el módulo lzma*), 527
decompress() (*en el módulo zlib*), 514
decompress() (*método de bz2.BZ2Decompressor*), 522
decompress() (*método de lzma.LZMADecompressor*), 526
decompress() (*método de zlib.Decompress*), 516
decompressobj() (*en el módulo zlib*), 515
decorador, **2082**
DEDENT (*en el módulo token*), 1924
dedent() (*en el módulo textwrap*), 155
deepcopy() (*en el módulo copy*), 286
def_prog_mode() (*en el módulo curses*), 757
def_shell_mode() (*en el módulo curses*), 757
default (*atributo de inspect.Parameter*), 1834
default (*atributo de optparse.Option*), 2042
default (*en el módulo email.policy*), 1114
DEFAULT (*en el módulo unittest.mock*), 1633
default() (*método de cmd.Cmd*), 1444
default() (*método de json.JSONEncoder*), 1157
DEFAULT_BUFFER_SIZE (*en el módulo io*), 658
default_bufsize (*en el módulo xml.dom.pulldom*), 1234
default_exception_handler() (*método de asyncio.loop*), 968
default_factory (*atributo de collections.defaultdict*), 248
DEFAULT_FORMAT (*en el módulo tarfile*), 542
DEFAULT_IGNORES (*en el módulo filecmp*), 446
default_open() (*método de urllib.request.BaseHandler*), 1277
DEFAULT_PROTOCOL (*en el módulo pickle*), 467
default_timer() (*en el módulo timeit*), 1712
DefaultContext (*clase en decimal*), 343
DefaultCookiePolicy (*clase en http.cookiejar*), 1351
defaultdict (*clase en collections*), 248
DefaultDict (*clase en typing*), 1535
DefaultEventLoopPolicy (*clase en asyncio*), 993

DefaultHandler() (*método de xml.parsers.expat.xmlparser*), 1251
DefaultHandlerExpand() (*método de xml.parsers.expat.xmlparser*), 1251
defaults() (*método de configparser.ConfigParser*), 575
DefaultSelector (*clase en selectors*), 1077
defaultTestLoader (*en el módulo unittest*), 1597
defaultTestResult() (*método de unittest.TestCase*), 1588
defects (*atributo de email.headerregistry.BaseHeader*), 1118
defects (*atributo de email.message.EmailMessage*), 1102
defects (*atributo de email.message.Message*), 1140
defpath (*en el módulo os*), 655
DefragResult (*clase en urllib.parse*), 1294
DefragResultBytes (*clase en urllib.parse*), 1295
degrees() (*en el módulo math*), 326
degrees() (*en el módulo turtle*), 1420
del
    sentencia, 44, 83
Del (*clase en ast*), 1895
del_param() (*método de email.message.EmailMessage*), 1098
del_param() (*método de email.message.Message*), 1137
delattr()
    función incorporada, 10
delay() (*en el módulo turtle*), 1432
delay_output() (*en el módulo curses*), 757
delayload (*atributo de http.cookiejar.FileCookieJar*), 1353
delch() (*método de curses.window*), 765
dele() (*método de poplib.POP3*), 1315
Delete (*clase en ast*), 1903
delete() (*método de ftplib.FTP*), 1312
delete() (*método de imaplib.IMAP4*), 1319
delete() (*método de tkinter.ttk.Treeview*), 1487
DELETE_ATTR (*opcode*), 1948
DELETE_DEREF (*opcode*), 1951
DELETE_FAST (*opcode*), 1950
DELETE_GLOBAL (*opcode*), 1948
DELETE_NAME (*opcode*), 1948
DELETE_SUBSCR (*opcode*), 1946
deleteacl() (*método de imaplib.IMAP4*), 1319
deletefilehandler() (*método de tkinter.Widget.tk*), 1467
DeleteKey() (*en el módulo winreg*), 1960
DeleteKeyEx() (*en el módulo winreg*), 1960
deleteln() (*método de curses.window*), 765
deleteMe() (*método de bdb.Breakpoint*), 1689
DeleteValue() (*en el módulo winreg*), 1960
delimiter (*atributo de csv.Dialect*), 559
delitem() (*en el módulo operator*), 408
deliver_challenge() (*en el módulo multiprocessing.connection*), 864
delocalize() (*en el módulo locale*), 1405
demo_app() (*en el módulo wsgiref.simple_server*), 1263
denominator (*atributo de fractions.Fraction*), 359
denominator (*atributo de numbers.Rational*), 318
DeprecationWarning, 109
deque (*clase en collections*), 245
Deque (*clase en typing*), 1535
dequeue() (*método de logging.handlers.QueueListener*), 755
DER_cert_to_PEM_cert() (*en el módulo ssl*), 1038
derwin() (*método de curses.window*), 765
DES
    cipher, 2006
description (*atributo de inspect.Parameter.kind*), 1835
description (*atributo de sqlite3.Cursor*), 503
description() (*método de nntplib.NNTP*), 2026
descriptions() (*método de nntplib.NNTP*), 2025
descriptor, **2082**
despacho único, **2091**
dest (*atributo de optparse.Option*), 2042
detach() (*método de io.BufferedIOBase*), 663
detach() (*método de io.TextIOBase*), 666
detach() (*método de socket.socket*), 1023
detach() (*método de tkinter.ttk.Treeview*), 1487
detach() (*método de weakref.finalize*), 276
Detach() (*método de winreg.PyHKEY*), 1967
DETACHED_PROCESS (*en el módulo subprocess*), 902
--details
    inspectopción de línea de comando, 1844
detect_api_mismatch() (*en el módulo test.support*), 1675
detect_encoding() (*en el módulo tokenize*), 1929
deterministic profiling, 1703
device_encoding() (*en el módulo os*), 610
devnull (*en el módulo os*), 655
DEVNULL (*en el módulo subprocess*), 890
devpoll() (*en el módulo select*), 1068
DevpollSelector (*clase en selectors*), 1077
dgettext() (*en el módulo gettext*), 1392
dgettext() (*en el módulo locale*), 1407
dialect (*atributo de csv.csvreader*), 560
dialect (*atributo de csv.csvwriter*), 561
Dialect (*clase en csv*), 558
Dialog (*clase en msilib*), 2020
Dialog (*clase en tkinter.commondialog*), 1472
Dialog (*clase en tkinter.simpledialog*), 1469
diccionario, **2082**
dict (*2to3 fixer*), 1662
Dict (*clase en ast*), 1895
Dict (*clase en typing*), 1534
dict (*clase incorporada*), 83
dict() (*método de multiprocessing.managers.SyncManager*), 857
DICT_MERGE (*opcode*), 1949
DICT_UPDATE (*opcode*), 1949

DictComp (*clase en ast*), 1899
dictConfig() (*en el módulo logging.config*), 731
dictionary
    objeto, 83
    type, operations on, 83
DictReader (*clase en csv*), 557
DictWriter (*clase en csv*), 557
diff_bytes() (*en el módulo difflib*), 148
diff_files (*atributo de filecmp.dircmp*), 445
Differ (*clase en difflib*), 145
difference() (*método de frozenset*), 82
difference_update() (*método de frozenset*), 83
difflib
    módulo, 144
digest() (*en el módulo hmac*), 596
digest() (*método de hashlib.hash*), 587
digest() (*método de hashlib.shake*), 587
digest() (*método de hmac.HMAC*), 596
digest_size (*atributo de hmac.HMAC*), 596
digit() (*en el módulo unicodedata*), 159
digits (*en el módulo string*), 113
dir()
    función incorporada, 10
dir() (*método de ftplib.FTP*), 1312
dircmp (*clase en filecmp*), 445
directory
    changing, 620
    creating, 624
    deleting, 457, 626
    site-packages, 1844
    traversal, 636, 637
    walking, 636, 637
directory ...
    compileallopción de línea de comando, 1936
Directory (*clase en msilib*), 2019
Directory (*clase en tkinter.filedialog*), 1471
DirEntry (*clase en os*), 628
DirList (*clase en tkinter.tix*), 1495
dirname() (*en el módulo os.path*), 432
dirs_double_event() (*método de tkinter.filedialog.FileDialog*), 1471
dirs_select_event() (*método de tkinter.filedialog.FileDialog*), 1471
DirSelectBox (*clase en tkinter.tix*), 1495
DirSelectDialog (*clase en tkinter.tix*), 1495
DirsOnSysPath (*clase en test.support.import_helper*), 1683
DirTree (*clase en tkinter.tix*), 1495
dis
    módulo, 1940
dis() (*en el módulo dis*), 1941
dis() (*en el módulo pickletools*), 1954
dis() (*método de dis.Bytecode*), 1941
disable (*pdb command*), 1700
disable() (*en el módulo faulthandler*), 1695
disable() (*en el módulo gc*), 1824
disable() (*en el módulo logging*), 727
disable() (*método de bdb.Breakpoint*), 1690
disable() (*método de profile.Profile*), 1706
disable_faulthandler() (*en el módulo test.support*), 1672
disable_gc() (*en el módulo test.support*), 1672
disable_interspersed_args() (*método de optparse.OptionParser*), 2046
DisableReflectionKey() (*en el módulo winreg*), 1964
disassemble() (*en el módulo dis*), 1942
discard (*atributo de http.cookiejar.Cookie*), 1357
discard() (*método de frozenset*), 83
discard() (*método de mailbox.Mailbox*), 1162
discard() (*método de mailbox.MH*), 1168
discard_buffers() (*método de asynchat.async_chat*), 1988
disco() (*en el módulo dis*), 1942
discover() (*método de unittest.TestLoader*), 1593
disk_usage() (*en el módulo shutil*), 458
dispatch_call() (*método de bdb.Bdb*), 1691
dispatch_exception() (*método de bdb.Bdb*), 1691
dispatch_line() (*método de bdb.Bdb*), 1691
dispatch_return() (*método de bdb.Bdb*), 1691
dispatch_table (*atributo de pickle.Pickler*), 469
dispatcher (*clase en asyncore*), 1991
dispatcher_with_send (*clase en asyncore*), 1992
DISPLAY, 1457
display (*pdb command*), 1701
display_name (*atributo de email.headerregistry.Address*), 1122
display_name (*atributo de email.headerregistry.Group*), 1122
displayhook() (*en el módulo sys*), 1752
dist() (*en el módulo math*), 325
distance() (*en el módulo turtle*), 1419
distb() (*en el módulo dis*), 1942
distutils
    módulo, 1731
Div (*clase en ast*), 1896
divide() (*método de decimal.Context*), 345
divide_int() (*método de decimal.Context*), 345
división entera, **2083**
DivisionByZero (*clase en decimal*), 349
divmod()
    función incorporada, 11
divmod() (*método de decimal.Context*), 345
DllCanUnloadNow() (*en el módulo ctypes*), 813
DllGetClassObject() (*en el módulo ctypes*), 813
dllhandle (*en el módulo sys*), 1752
dnd_start() (*en el módulo tkinter.dnd*), 1474
DndHandler (*clase en tkinter.dnd*), 1474
dngettext() (*en el módulo gettext*), 1392
dnpgettext() (*en el módulo gettext*), 1392
do_clear() (*método de bdb.Bdb*), 1692
do_command() (*método de curses.textpad.Textbox*), 774
do_GET() (*método de*

http.server.SimpleHTTPRequestHandler), 1345
do_handshake() (*método de ssl.SSLSocket*), 1047
do_HEAD() (*método de http.server.SimpleHTTPRequestHandler*), 1344
do_POST() (*método de http.server.CGIHTTPRequestHandler*), 1346
doc (*atributo de json.JSONDecodeError*), 1158
doc_header (*atributo de cmd.Cmd*), 1445
DocCGIXMLRPCRequestHandler (*clase en xmlrpc.server*), 1371
DocFileSuite() (*en el módulo doctest*), 1560
doClassCleanups() (*método de clase de unittest.TestCase*), 1589
doCleanups() (*método de unittest.TestCase*), 1589
docmd() (*método de smtplib.SMTP*), 1325
docstring, **2083**
docstring (*atributo de doctest.DocTest*), 1564
doctest
    módulo, 1548
DocTest (*clase en doctest*), 1563
DocTestFailure, 1570
DocTestFinder (*clase en doctest*), 1564
DocTestParser (*clase en doctest*), 1565
DocTestRunner (*clase en doctest*), 1566
DocTestSuite() (*en el módulo doctest*), 1561
doctype() (*método de xml.etree.ElementTree.TreeBuilder*), 1215
documentation
    generation, 1544
    online, 1544
documentElement (*atributo de xml.dom.Document*), 1223
DocXMLRPCRequestHandler (*clase en xmlrpc.server*), 1371
DocXMLRPCServer (*clase en xmlrpc.server*), 1371
domain (*atributo de email.headerregistry.Address*), 1122
domain (*atributo de tracemalloc.DomainFilter*), 1725
domain (*atributo de tracemalloc.Filter*), 1725
domain (*atributo de tracemalloc.Trace*), 1728
domain_initial_dot (*atributo de http.cookiejar.Cookie*), 1358
domain_return_ok() (*método de http.cookiejar.CookiePolicy*), 1354
domain_specified (*atributo de http.cookiejar.Cookie*), 1358
DomainFilter (*clase en tracemalloc*), 1725
DomainLiberal (*atributo de http.cookiejar.DefaultCookiePolicy*), 1357
DomainRFC2965Match (*atributo de http.cookiejar.DefaultCookiePolicy*), 1357
DomainStrict (*atributo de http.cookiejar.DefaultCookiePolicy*), 1357
DomainStrictNoDots (*atributo de http.cookiejar.DefaultCookiePolicy*), 1356
DomainStrictNonDomain (*atributo de http.cookiejar.DefaultCookiePolicy*), 1356
DOMEventStream (*clase en xml.dom.pulldom*), 1234
DOMException, 1226
doModuleCleanups() (*en el módulo unittest*), 1600
DomstringSizeErr, 1226
done() (*en el módulo turtle*), 1434
done() (*método de asyncio.Future*), 976
done() (*método de asyncio.Task*), 932
done() (*método de concurrent.futures.Future*), 886
done() (*método de graphlib.TopologicalSorter*), 314
done() (*método de xdrlib.Unpacker*), 2074
DONT_ACCEPT_BLANKLINE (*en el módulo doctest*), 1555
DONT_ACCEPT_TRUE_FOR_1 (*en el módulo doctest*), 1555
dont_write_bytecode (*en el módulo sys*), 1753
doRollover() (*método de logging.handlers.RotatingFileHandler*), 746
doRollover() (*método de logging.handlers.TimedRotatingFileHandler*), 747
DOT (*en el módulo token*), 1925
dot() (*en el módulo turtle*), 1416
DOTALL (*en el módulo re*), 131
doublequote (*atributo de csv.Dialect*), 559
DOUBLESLASH (*en el módulo token*), 1926
DOUBLESLASHEQUAL (*en el módulo token*), 1926
DOUBLESTAR (*en el módulo token*), 1926
DOUBLESTAREQUAL (*en el módulo token*), 1926
doupdate() (*en el módulo curses*), 758
down (*pdb command*), 1699
down() (*en el módulo turtle*), 1420
dpgettext() (*en el módulo gettext*), 1392
drain() (*método de asyncio.StreamWriter*), 938
drive (*atributo de pathlib.PurePath*), 418
drop_whitespace (*atributo de textwrap.TextWrapper*), 157
dropwhile() (*en el módulo itertools*), 386
dst() (*método de datetime.datetime*), 208
dst() (*método de datetime.time*), 216
dst() (*método de datetime.timezone*), 224
dst() (*método de datetime.tzinfo*), 217
DTDHandler (*clase en xml.sax.handler*), 1236
dump() (*en el módulo ast*), 1920
dump() (*en el módulo json*), 1154
dump() (*en el módulo marshal*), 485
dump() (*en el módulo pickle*), 468
dump() (*en el módulo plistlib*), 582
dump() (*en el módulo xml.etree.ElementTree*), 1206
dump() (*método de pickle.Pickler*), 469
dump() (*método de tracemalloc.Snapshot*), 1726
dump_stats() (*método de profile.Profile*), 1706
dump_stats() (*método de pstats.Stats*), 1707
dump_traceback() (*en el módulo faulthandler*), 1695
dump_traceback_later() (*en el módulo faulthandler*), 1695
dumps() (*en el módulo json*), 1154

dumps() (*en el módulo marshal*), 486
dumps() (*en el módulo pickle*), 468
dumps() (*en el módulo plistlib*), 582
dumps() (*en el módulo xmlrpc.client*), 1365
dup() (*en el módulo os*), 610
dup() (*método de socket.socket*), 1023
dup2() (*en el módulo os*), 610
DUP_TOP (*opcode*), 1944
DUP_TOP_TWO (*opcode*), 1944
DuplicateOptionError, 579
DuplicateSectionError, 579
dwFlags (*atributo de subprocess.STARTUPINFO*), 900
DynamicClassAttribute() (*en el módulo types*), 285

E

-e
    tokenizeopción de línea de comando, 1929
e (*en el módulo cmath*), 330
e (*en el módulo math*), 327
-e <tarfile> [<output_dir>]
    tarfileopción de línea de comando, 551
-e <zipfile> <output_dir>
    zipfileopción de línea de comando, 538
-e dir
    compileallopción de línea de comando, 1937
E2BIG (*en el módulo errno*), 783
EACCES (*en el módulo errno*), 783
EADDRINUSE (*en el módulo errno*), 787
EADDRNOTAVAIL (*en el módulo errno*), 787
EADV (*en el módulo errno*), 785
EAFNOSUPPORT (*en el módulo errno*), 787
EAFP, **2083**
EAGAIN (*en el módulo errno*), 783
EALREADY (*en el módulo errno*), 787
east_asian_width() (*en el módulo unicodedata*), 159
EBADE (*en el módulo errno*), 785
EBADF (*en el módulo errno*), 783
EBADFD (*en el módulo errno*), 786
EBADMSG (*en el módulo errno*), 786
EBADR (*en el módulo errno*), 785
EBADRQC (*en el módulo errno*), 785
EBADSLT (*en el módulo errno*), 785
EBFONT (*en el módulo errno*), 785
EBUSY (*en el módulo errno*), 783
ECHILD (*en el módulo errno*), 783
echo() (*en el módulo curses*), 758
echochar() (*método de curses.window*), 765
ECHRNG (*en el módulo errno*), 784
ECOMM (*en el módulo errno*), 785
ECONNABORTED (*en el módulo errno*), 787
ECONNREFUSED (*en el módulo errno*), 787
ECONNRESET (*en el módulo errno*), 787
EDEADLK (*en el módulo errno*), 784
EDEADLOCK (*en el módulo errno*), 785
EDESTADDRREQ (*en el módulo errno*), 786
edit() (*método de curses.textpad.Textbox*), 774
EDOM (*en el módulo errno*), 784
EDOTDOT (*en el módulo errno*), 786
EDQUOT (*en el módulo errno*), 788
EEXIST (*en el módulo errno*), 783
EFAULT (*en el módulo errno*), 783
EFBIG (*en el módulo errno*), 784
EFD_CLOEXEC (*en el módulo os*), 639
EFD_NONBLOCK (*en el módulo os*), 639
EFD_SEMAPHORE (*en el módulo os*), 639
effective() (*en el módulo bdb*), 1694
ehlo() (*método de smtplib.SMTP*), 1326
ehlo_or_helo_if_needed() (*método de smtplib.SMTP*), 1326
EHOSTDOWN (*en el módulo errno*), 787
EHOSTUNREACH (*en el módulo errno*), 787
EIDRM (*en el módulo errno*), 784
EILSEQ (*en el módulo errno*), 786
EINPROGRESS (*en el módulo errno*), 787
EINTR (*en el módulo errno*), 783
EINVAL (*en el módulo errno*), 783
EIO (*en el módulo errno*), 783
EISCONN (*en el módulo errno*), 787
EISDIR (*en el módulo errno*), 783
EISNAM (*en el módulo errno*), 788
EL2HLT (*en el módulo errno*), 785
EL2NSYNC (*en el módulo errno*), 784
EL3HLT (*en el módulo errno*), 784
EL3RST (*en el módulo errno*), 784
Element (*clase en xml.etree.ElementTree*), 1210
element_create() (*método de tkinter.ttk.Style*), 1491
element_names() (*método de tkinter.ttk.Style*), 1492
element_options() (*método de tkinter.ttk.Style*), 1492
ElementDeclHandler() (*método de xml.parsers.expat.xmlparser*), 1250
elements() (*método de collections.Counter*), 242
ElementTree (*clase en xml.etree.ElementTree*), 1213
ELIBACC (*en el módulo errno*), 786
ELIBBAD (*en el módulo errno*), 786
ELIBEXEC (*en el módulo errno*), 786
ELIBMAX (*en el módulo errno*), 786
ELIBSCN (*en el módulo errno*), 786
Ellinghouse, Lance, 2071
ELLIPSIS (*en el módulo doctest*), 1555
ELLIPSIS (*en el módulo token*), 1926
Ellipsis (*variable incorporada*), 31
EllipsisType (*en el módulo types*), 283
ELNRNG (*en el módulo errno*), 785
ELOOP (*en el módulo errno*), 784
email
    módulo, 1093
email.charset
    módulo, 1145

email.contentmanager
    módulo, 1123
email.encoders
    módulo, 1147
email.errors
    módulo, 1116
email.generator
    módulo, 1106
email.header
    módulo, 1142
email.headerregistry
    módulo, 1117
email.iterators
    módulo, 1150
email.message
    módulo, 1094
EmailMessage (*clase en email.message*), 1095
email.mime
    módulo, 1140
email.parser
    módulo, 1102
email.policy
    módulo, 1109
EmailPolicy (*clase en email.policy*), 1113
email.utils
    módulo, 1148
EMFILE (*en el módulo errno*), 783
emit() (*método de logging.FileHandler*), 743
emit() (*método de logging.Handler*), 719
emit() (*método de logging.handlers.BufferingHandler*), 752
emit() (*método de logging.handlers.DatagramHandler*), 748
emit() (*método de logging.handlers.HTTPHandler*), 753
emit() (*método de logging.handlers.NTEventLogHandler*), 751
emit() (*método de logging.handlers.QueueHandler*), 753
emit() (*método de logging.handlers.RotatingFileHandler*), 746
emit() (*método de logging.handlers.SMTPHandler*), 752
emit() (*método de logging.handlers.SocketHandler*), 747
emit() (*método de logging.handlers.SysLogHandler*), 749
emit() (*método de logging.handlers.TimedRotatingFileHandler*), 747
emit() (*método de logging.handlers.WatchedFileHandler*), 744
emit() (*método de logging.NullHandler*), 743
emit() (*método de logging.StreamHandler*), 742
EMLINK (*en el módulo errno*), 784
Empty, 910
empty (*atributo de inspect.Parameter*), 1834
empty (*atributo de inspect.Signature*), 1833
empty() (*método de asyncio.Queue*), 950
empty() (*método de multiprocessing.Queue*), 845
empty() (*método de multiprocessing.SimpleQueue*), 846
empty() (*método de queue.Queue*), 910
empty() (*método de queue.SimpleQueue*), 912
empty() (*método de sched.scheduler*), 909
EMPTY_NAMESPACE (*en el módulo xml.dom*), 1219
emptyline() (*método de cmd.Cmd*), 1444
EMSGSIZE (*en el módulo errno*), 786
EMULTIHOP (*en el módulo errno*), 786
enable (*pdb command*), 1700
enable() (*en el módulo cgitb*), 2004
enable() (*en el módulo faulthandler*), 1695
enable() (*en el módulo gc*), 1824
enable() (*método de bdb.Breakpoint*), 1689
enable() (*método de imaplib.IMAP4*), 1319
enable() (*método de profile.Profile*), 1706
enable_callback_tracebacks() (*en el módulo sqlite3*), 494
enable_interspersed_args() (*método de optparse.OptionParser*), 2046
enable_load_extension() (*método de sqlite3.Connection*), 499
enable_traversal() (*método de tkinter.ttk.Notebook*), 1482
ENABLE_USER_SITE (*en el módulo site*), 1846
enabled (*atributo de bdb.Breakpoint*), 1690
EnableReflectionKey() (*en el módulo winreg*), 1964
ENAMETOOLONG (*en el módulo errno*), 784
ENAVAIL (*en el módulo errno*), 788
enclose() (*método de curses.window*), 765
encode
    Codecs, 174
encode (*atributo de codecs.CodecInfo*), 174
encode() (*en el módulo base64*), 1184
encode() (*en el módulo codecs*), 174
encode() (*en el módulo quopri*), 1188
encode() (*en el módulo uu*), 2071
encode() (*método de codecs.Codec*), 179
encode() (*método de codecs.IncrementalEncoder*), 180
encode() (*método de email.header.Header*), 1144
encode() (*método de json.JSONEncoder*), 1158
encode() (*método de str*), 49
encode() (*método de xmlrpc.client.Binary*), 1362
encode() (*método de xmlrpc.client.DateTime*), 1362
encode_7or8bit() (*en el módulo email.encoders*), 1148
encode_base64() (*en el módulo email.encoders*), 1148
encode_noop() (*en el módulo email.encoders*), 1148
encode_quopri() (*en el módulo email.encoders*), 1147
encode_rfc2231() (*en el módulo email.utils*), 1150
encodebytes() (*en el módulo base64*), 1184
EncodedFile() (*en el módulo codecs*), 176

encodePriority() (*método de logging.handlers.SysLogHandler*), 750
encodestring() (*en el módulo quopri*), 1189
encoding
    base64, 1182
    quoted-printable, 1188
encoding (*atributo de curses.window*), 765
encoding (*atributo de io.TextIOBase*), 666
encoding (*atributo de UnicodeError*), 107
ENCODING (*en el módulo tarfile*), 541
ENCODING (*en el módulo token*), 1927
encodings_map (*atributo de mimetypes.MimeTypes*), 1181
encodings_map (*en el módulo mimetypes*), 1180
encodings.idna
    módulo, 190
encodings.mbcs
    módulo, 190
encodings.utf_8_sig
    módulo, 191
EncodingWarning, 109
end (*atributo de UnicodeError*), 107
end() (*método de re.Match*), 138
end() (*método de xml.etree.ElementTree.TreeBuilder*), 1215
END_ASYNC_FOR (*opcode*), 1946
end_col_offset (*atributo de ast.AST*), 1892
end_fill() (*en el módulo turtle*), 1423
end_headers() (*método de http.server.BaseHTTPRequestHandler*), 1343
end_lineno (*atributo de ast.AST*), 1892
end_lineno (*atributo de SyntaxError*), 106
end_lineno (*atributo de traceback.TracebackException*), 1818
end_ns() (*método de xml.etree.ElementTree.TreeBuilder*), 1215
end_offset (*atributo de SyntaxError*), 106
end_offset (*atributo de traceback.TracebackException*), 1819
end_poly() (*en el módulo turtle*), 1429
endCDATA() (*método de xml.sax.handler.LexicalHandler*), 1242
EndCdataSectionHandler() (*método de xml.parsers.expat.xmlparser*), 1251
EndDoctypeDeclHandler() (*método de xml.parsers.expat.xmlparser*), 1250
endDocument() (*método de xml.sax.handler.ContentHandler*), 1239
endDTD() (*método de xml.sax.handler.LexicalHandler*), 1242
endElement() (*método de xml.sax.handler.ContentHandler*), 1239
EndElementHandler() (*método de xml.parsers.expat.xmlparser*), 1251
endElementNS() (*método de xml.sax.handler.ContentHandler*), 1240
endheaders() (*método de http.client.HTTPConnection*), 1305
ENDMARKER (*en el módulo token*), 1924
EndNamespaceDeclHandler() (*método de xml.parsers.expat.xmlparser*), 1251
endpos (*atributo de re.Match*), 138
endPrefixMapping() (*método de xml.sax.handler.ContentHandler*), 1239
endswith() (*método de bytearray*), 63
endswith() (*método de bytes*), 63
endswith() (*método de str*), 50
endwin() (*en el módulo curses*), 758
ENETDOWN (*en el módulo errno*), 787
ENETRESET (*en el módulo errno*), 787
ENETUNREACH (*en el módulo errno*), 787
ENFILE (*en el módulo errno*), 783
ENOANO (*en el módulo errno*), 785
ENOBUFS (*en el módulo errno*), 787
ENOCSI (*en el módulo errno*), 785
ENODATA (*en el módulo errno*), 785
ENODEV (*en el módulo errno*), 783
ENOENT (*en el módulo errno*), 783
ENOEXEC (*en el módulo errno*), 783
ENOLCK (*en el módulo errno*), 784
ENOLINK (*en el módulo errno*), 785
ENOMEM (*en el módulo errno*), 783
ENOMSG (*en el módulo errno*), 784
ENONET (*en el módulo errno*), 785
ENOPKG (*en el módulo errno*), 785
ENOPROTOOPT (*en el módulo errno*), 786
ENOSPC (*en el módulo errno*), 784
ENOSR (*en el módulo errno*), 785
ENOSTR (*en el módulo errno*), 785
ENOSYS (*en el módulo errno*), 784
ENOTBLK (*en el módulo errno*), 783
ENOTCONN (*en el módulo errno*), 787
ENOTDIR (*en el módulo errno*), 783
ENOTEMPTY (*en el módulo errno*), 784
ENOTNAM (*en el módulo errno*), 788
ENOTSOCK (*en el módulo errno*), 786
ENOTTY (*en el módulo errno*), 784
ENOTUNIQ (*en el módulo errno*), 786
enqueue() (*método de logging.handlers.QueueHandler*), 754
enqueue_sentinel() (*método de logging.handlers.QueueListener*), 755
ensure_directories() (*método de venv.EnvBuilder*), 1737
ensure_future() (*en el módulo asyncio*), 975
ensurepip
    módulo, 1732
enter() (*método de sched.scheduler*), 909
enter_async_context() (*método de contextlib.AsyncExitStack*), 1804
enter_context() (*método de contextlib.ExitStack*), 1803
enterabs() (*método de sched.scheduler*), 908
entities (*atributo de xml.dom.DocumentType*), 1222
EntityDeclHandler() (*método de xml.parsers.expat.xmlparser*), 1251

entitydefs (*en el módulo html.entities*), 1196
EntityResolver (*clase en xml.sax.handler*), 1237
entorno virtual, **2092**
entrada de ruta, **2089**
enum
    módulo, 294
Enum (*clase en enum*), 294
enum_certificates() (*en el módulo ssl*), 1039
enum_crls() (*en el módulo ssl*), 1039
enumerate()
    función incorporada, 11
enumerate() (*en el módulo threading*), 822
EnumKey() (*en el módulo winreg*), 1961
EnumValue() (*en el módulo winreg*), 1961
EnvBuilder (*clase en venv*), 1736
environ (*en el módulo os*), 603
environ (*en el módulo posix*), 1972
environb (*en el módulo os*), 603
environment variables
    deleting, 609
    setting, 606
EnvironmentError, 107
Environments
    virtual, 1733
EnvironmentVarGuard (*clase en test.support.os_helper*), 1680
ENXIO (*en el módulo errno*), 783
eof (*atributo de bz2.BZ2Decompressor*), 522
eof (*atributo de lzma.LZMADecompressor*), 527
eof (*atributo de shlex.shlex*), 1451
eof (*atributo de ssl.MemoryBIO*), 1065
eof (*atributo de zlib.Decompress*), 516
eof_received() (*método de asyncio.BufferedProtocol*), 986
eof_received() (*método de asyncio.Protocol*), 985
EOFError, 103
EOPNOTSUPP (*en el módulo errno*), 787
EOVERFLOW (*en el módulo errno*), 786
EPERM (*en el módulo errno*), 782
EPFNOSUPPORT (*en el módulo errno*), 787
epilogue (*atributo de email.message.EmailMessage*), 1102
epilogue (*atributo de email.message.Message*), 1139
EPIPE (*en el módulo errno*), 784
epoch, 670
epoll() (*en el módulo select*), 1068
EpollSelector (*clase en selectors*), 1077
EPROTO (*en el módulo errno*), 786
EPROTONOSUPPORT (*en el módulo errno*), 786
EPROTOTYPE (*en el módulo errno*), 786
Eq (*clase en ast*), 1897
eq() (*en el módulo operator*), 406
EQEQUAL (*en el módulo token*), 1925
EQUAL (*en el módulo token*), 1925
ERA (*en el módulo locale*), 1403
ERA_D_FMT (*en el módulo locale*), 1403
ERA_D_T_FMT (*en el módulo locale*), 1403
ERA_T_FMT (*en el módulo locale*), 1403
ERANGE (*en el módulo errno*), 784
erase() (*método de curses.window*), 766
erasechar() (*en el módulo curses*), 758
EREMCHG (*en el módulo errno*), 786
EREMOTE (*en el módulo errno*), 785
EREMOTEIO (*en el módulo errno*), 788
ERESTART (*en el módulo errno*), 786
erf() (*en el módulo math*), 326
erfc() (*en el módulo math*), 326
EROFS (*en el módulo errno*), 784
ERR (*en el módulo curses*), 770
errcheck (*atributo de ctypes._FuncPtr*), 809
errcode (*atributo de xmlrpc.client.ProtocolError*), 1364
errmsg (*atributo de xmlrpc.client.ProtocolError*), 1364
errno
    módulo, 104, 782
errno (*atributo de OSError*), 104
error, 134, 168, 486, 488490, 513, 601, 712, 757, 916, 1011, 1068, 1247, 1979, 1994, 2021
Error, 286, 459, 504, 559, 579, 1177, 1185, 1188, 1257, 1387, 1400, 2066, 2072, 2075
error handler's name
    backslashreplace, 177
    ignore, 177
    namereplace, 177
    replace, 177
    strict, 177
    surrogateescape, 177
    surrogatepass, 177
    xmlcharrefreplace, 177
error() (*en el módulo logging*), 727
error() (*método de argparse.ArgumentParser*), 710
error() (*método de logging.Logger*), 717
error() (*método de urllib.request.OpenerDirector*), 1277
error() (*método de xml.sax.handler.ErrorHandler*), 1241
error_body (*atributo de wsgiref.handlers.BaseHandler*), 1268
error_content_type (*atributo de http.server.BaseHTTPRequestHandler*), 1342
error_headers (*atributo de wsgiref.handlers.BaseHandler*), 1268
error_leader() (*método de shlex.shlex*), 1450
error_message_format (*atributo de http.server.BaseHTTPRequestHandler*), 1342
error_output() (*método de wsgiref.handlers.BaseHandler*), 1267
error_perm, 1310
error_proto, 1310, 1314
error_received() (*método de asyncio.DatagramProtocol*), 986
error_reply, 1310
error_status (*atributo de wsgiref.handlers.BaseHandler*), 1268
error_temp, 1310

ErrorByteIndex (*atributo de xml.parsers.expat.xmlparser*), 1249
ErrorCode (*atributo de xml.parsers.expat.xmlparser*), 1250
errorcode (*en el módulo errno*), 782
ErrorColumnNumber (*atributo de xml.parsers.expat.xmlparser*), 1250
ErrorHandler (*clase en xml.sax.handler*), 1237
errorlevel (*atributo de tarfile.TarFile*), 544
ErrorLineNumber (*atributo de xml.parsers.expat.xmlparser*), 1250
Errors
    logging, 714
errors (*atributo de io.TextIOBase*), 666
errors (*atributo de unittest.TestLoader*), 1592
errors (*atributo de unittest.TestResult*), 1594
ErrorString() (*en el módulo xml.parsers.expat*), 1247
ERRORTOKEN (*en el módulo token*), 1927
escape (*atributo de shlex.shlex*), 1451
escape() (*en el módulo glob*), 451
escape() (*en el módulo html*), 1191
escape() (*en el módulo re*), 134
escape() (*en el módulo xml.sax.saxutils*), 1242
escapechar (*atributo de csv.Dialect*), 559
escapedquotes (*atributo de shlex.shlex*), 1451
ESHUTDOWN (*en el módulo errno*), 787
ESOCKTNOSUPPORT (*en el módulo errno*), 787
espacio de nombres, **2088**
especificador de módulo, **2087**
ESPIPE (*en el módulo errno*), 784
ESRCH (*en el módulo errno*), 783
ESRMNT (*en el módulo errno*), 785
ESTALE (*en el módulo errno*), 787
ESTRPIPE (*en el módulo errno*), 786
ETIME (*en el módulo errno*), 785
ETIMEDOUT (*en el módulo errno*), 787
Etiny() (*método de decimal.Context*), 345
ETOOMANYREFS (*en el módulo errno*), 787
Etop() (*método de decimal.Context*), 345
ETXTBSY (*en el módulo errno*), 784
EUCLEAN (*en el módulo errno*), 788
EUNATCH (*en el módulo errno*), 785
EUSERS (*en el módulo errno*), 786
eval
    función incorporada, 96, 289
eval()
    función incorporada, 11
Event (*clase en asyncio*), 942
Event (*clase en multiprocessing*), 850
Event (*clase en threading*), 831
event scheduling, 908
event() (*método de msilib.Control*), 2020
Event() (*método de multiprocessing.managers.SyncManager*), 857
eventfd() (*en el módulo os*), 638
eventfd_read() (*en el módulo os*), 639
eventfd_write() (*en el módulo os*), 639
events (*atributo de selectors.SelectorKey*), 1075
events (*widgets*), 1466
EWOULDBLOCK (*en el módulo errno*), 784
EX_CANTCREAT (*en el módulo os*), 643
EX_CONFIG (*en el módulo os*), 643
EX_DATAERR (*en el módulo os*), 642
EX_IOERR (*en el módulo os*), 643
EX_NOHOST (*en el módulo os*), 643
EX_NOINPUT (*en el módulo os*), 642
EX_NOPERM (*en el módulo os*), 643
EX_NOTFOUND (*en el módulo os*), 643
EX_NOUSER (*en el módulo os*), 642
EX_OK (*en el módulo os*), 642
EX_OSERR (*en el módulo os*), 643
EX_OSFILE (*en el módulo os*), 643
EX_PROTOCOL (*en el módulo os*), 643
EX_SOFTWARE (*en el módulo os*), 643
EX_TEMPFAIL (*en el módulo os*), 643
EX_UNAVAILABLE (*en el módulo os*), 643
EX_USAGE (*en el módulo os*), 642
--exact
    tokenizeopción de línea de comando, 1929
example (*atributo de doctest.DocTestFailure*), 1570
example (*atributo de doctest.UnexpectedException*), 1570
Example (*clase en doctest*), 1564
examples (*atributo de doctest.DocTest*), 1563
exc_info (*atributo de doctest.UnexpectedException*), 1570
exc_info() (*en el módulo sys*), 1753
exc_msg (*atributo de doctest.Example*), 1564
exc_type (*atributo de traceback.TracebackException*), 1818
excel (*clase en csv*), 558
excel_tab (*clase en csv*), 558
except
    sentencia, 101
except (*2to3 fixer*), 1662
ExceptHandler (*clase en ast*), 1907
excepthook() (*en el módulo sys*), 1753
excepthook() (*en el módulo threading*), 822
excepthook() (*in module sys*), 2004
Exception, 102
EXCEPTION (*en el módulo tkinter*), 1467
exception() (*en el módulo logging*), 727
exception() (*método de asyncio.Future*), 977
exception() (*método de asyncio.Task*), 932
exception() (*método de concurrent.futures.Future*), 887
exception() (*método de logging.Logger*), 717
exceptions
    in CGI scripts, 2004
EXDEV (*en el módulo errno*), 783
exec
    función incorporada, 12, 96
exec (*2to3 fixer*), 1662
exec()

función incorporada, 12
exec_module() (*método de importlib.abc.InspectLoader*), 1869
exec_module() (*método de importlib.abc.Loader*), 1867
exec_module() (*método de importlib.abc.SourceLoader*), 1871
exec_module() (*método de importlib.machinery.ExtensionFileLoader*), 1877
exec_module() (*método de zipimport.zipimporter*), 1854
exec_prefix (*en el módulo sys*), 1754
execfile (*2to3 fixer*), 1662
execl() (*en el módulo os*), 641
execle() (*en el módulo os*), 641
execlp() (*en el módulo os*), 641
execlpe() (*en el módulo os*), 641
executable (*en el módulo sys*), 1754
Executable Zip Files, 1742
Execute() (*método de msilib.View*), 2017
execute() (*método de sqlite3.Connection*), 496
execute() (*método de sqlite3.Cursor*), 502
executemany() (*método de sqlite3.Connection*), 496
executemany() (*método de sqlite3.Cursor*), 502
executescript() (*método de sqlite3.Connection*), 496
executescript() (*método de sqlite3.Cursor*), 502
ExecutionLoader (*clase en importlib.abc*), 1869
Executor (*clase en concurrent.futures*), 882
execv() (*en el módulo os*), 641
execve() (*en el módulo os*), 641
execvp() (*en el módulo os*), 641
execvpe() (*en el módulo os*), 641
ExFileSelectBox (*clase en tkinter.tix*), 1495
EXFULL (*en el módulo errno*), 785
exists() (*en el módulo os.path*), 432
exists() (*método de pathlib.Path*), 424
exists() (*método de tkinter.ttk.Treeview*), 1487
exists() (*método de zipfile.Path*), 535
exit (*variable incorporada*), 32
exit() (*en el módulo _thread*), 917
exit() (*en el módulo sys*), 1754
exit() (*método de argparse.ArgumentParser*), 710
exitcode (*atributo de multiprocessing.Process*), 842
exitfunc (*2to3 fixer*), 1662
exitonclick() (*en el módulo turtle*), 1436
ExitStack (*clase en contextlib*), 1802
exp() (*en el módulo cmath*), 329
exp() (*en el módulo math*), 324
exp() (*método de decimal.Context*), 345
exp() (*método de decimal.Decimal*), 338
expand() (*método de re.Match*), 136
expand_tabs (*atributo de textwrap.TextWrapper*), 157
ExpandEnvironmentStrings() (*en el módulo winreg*), 1961
expandNode() (*método de xml.dom.pulldom.DOMEventStream*), 1234
expandtabs() (*método de bytearray*), 68
expandtabs() (*método de bytes*), 68
expandtabs() (*método de str*), 50
expanduser() (*en el módulo os.path*), 433
expanduser() (*método de pathlib.Path*), 425
expandvars() (*en el módulo os.path*), 433
Expat, 1247
ExpatError, 1247
expect() (*método de telnetlib.Telnet*), 2070
expected (*atributo de asyncio.IncompleteReadError*), 953
expectedFailure() (*en el módulo unittest*), 1579
expectedFailures (*atributo de unittest.TestResult*), 1595
expires (*atributo de http.cookiejar.Cookie*), 1357
exploded (*atributo de ipaddress.IPv4Address*), 1374
exploded (*atributo de ipaddress.IPv4Network*), 1379
exploded (*atributo de ipaddress.IPv6Address*), 1375
exploded (*atributo de ipaddress.IPv6Network*), 1381
expm1() (*en el módulo math*), 324
expovariate() (*en el módulo random*), 364
Expr (*clase en ast*), 1896
expresión, **2083**
expresión generadora, **2084**
expunge() (*método de imaplib.IMAP4*), 1319
extend() (*método de array.array*), 271
extend() (*método de collections.deque*), 245
extend() (*método de xml.etree.ElementTree.Element*), 1211
extend() (*sequence method*), 44
extend_path() (*en el módulo pkgutil*), 1855
EXTENDED_ARG (*opcode*), 1952
ExtendedContext (*clase en decimal*), 343
ExtendedInterpolation (*clase en configparser*), 567
extendleft() (*método de collections.deque*), 245
EXTENSION_SUFFIXES (*en el módulo importlib.machinery*), 1874
ExtensionFileLoader (*clase en importlib.machinery*), 1877
extensions_map (*atributo de http.server.SimpleHTTPRequestHandler*), 1344
External Data Representation, 467, 2072
external_attr (*atributo de zipfile.ZipInfo*), 537
ExternalClashError, 1177
ExternalEntityParserCreate() (*método de xml.parsers.expat.xmlparser*), 1248
ExternalEntityRefHandler() (*método de xml.parsers.expat.xmlparser*), 1252
extra (*atributo de zipfile.ZipInfo*), 537
--extract <tarfile> [<output_dir>]
   tarfileopción de línea de comando, 551
--extract <zipfile> <output_dir>

zipfileopción de línea de comando, 538
extract() (*método de clase de traceback.StackSummary*), 1819
extract() (*método de tarfile.TarFile*), 544
extract() (*método de zipfile.ZipFile*), 532
extract_cookies() (*método de http.cookiejar.CookieJar*), 1352
extract_stack() (*en el módulo traceback*), 1817
extract_tb() (*en el módulo traceback*), 1817
extract_version (*atributo de zipfile.ZipInfo*), 537
extractall() (*método de tarfile.TarFile*), 543
extractall() (*método de zipfile.ZipFile*), 532
ExtractError, 541
extractfile() (*método de tarfile.TarFile*), 544
extraction_filter (*atributo de tarfile.TarFile*), 544
extsep (*en el módulo os*), 655

F

-f
compileallopción de línea de comando, 1936
traceopción de línea de comando, 1717
unittestopción de línea de comando, 1574
f-string, **2083**
f_contiguous (*atributo de memoryview*), 81
F_LOCK (*en el módulo os*), 611
F_OK (*en el módulo os*), 620
F_TEST (*en el módulo os*), 611
F_TLOCK (*en el módulo os*), 611
F_ULOCK (*en el módulo os*), 611
fabs() (*en el módulo math*), 321
factorial() (*en el módulo math*), 321
factory() (*método de clase de importlib.util.LazyLoader*), 1881
fail() (*método de unittest.TestCase*), 1588
FAIL_FAST (*en el módulo doctest*), 1556
--failfast
unittestopción de línea de comando, 1574
failfast (*atributo de unittest.TestResult*), 1595
failureException (*atributo de unittest.TestCase*), 1588
failures (*atributo de unittest.TestResult*), 1595
FakePath (*clase en test.support.os_helper*), 1680
false, 33
False, 33, 97
False (*Built-in object*), 33
False (*variable incorporada*), 31
families() (*en el módulo tkinter.font*), 1469
family (*atributo de socket.socket*), 1029
FancyURLopener (*clase en urllib.request*), 1286
--fast
gzipopción de línea de comando, 519
fast (*atributo de pickle.Pickler*), 469
FastChildWatcher (*clase en asyncio*), 995
fatalError() (*método de xml.sax.handler.ErrorHandler*), 1241
Fault (*clase en xmlrpc.client*), 1363
faultCode (*atributo de xmlrpc.client.Fault*), 1363
faulthandler
módulo, 1694
faultString (*atributo de xmlrpc.client.Fault*), 1363
fchdir() (*en el módulo os*), 622
fchmod() (*en el módulo os*), 610
fchown() (*en el módulo os*), 610
FCICreate() (*en el módulo msilib*), 2015
fcntl
módulo, 1977
fcntl() (*en el módulo fcntl*), 1977
fd (*atributo de selectors.SelectorKey*), 1075
fd() (*en el módulo turtle*), 1414
fd_count() (*en el módulo test.support.os_helper*), 1681
fdatasync() (*en el módulo os*), 610
fdopen() (*en el módulo os*), 609
Feature (*clase en msilib*), 2019
feature_external_ges (*en el módulo xml.sax.handler*), 1237
feature_external_pes (*en el módulo xml.sax.handler*), 1238
feature_namespace_prefixes (*en el módulo xml.sax.handler*), 1237
feature_namespaces (*en el módulo xml.sax.handler*), 1237
feature_string_interning (*en el módulo xml.sax.handler*), 1237
feature_validation (*en el módulo xml.sax.handler*), 1237
feed() (*método de email.parser.BytesFeedParser*), 1103
feed() (*método de html.parser.HTMLParser*), 1193
feed() (*método de xml.etree.ElementTree.XMLParser*), 1216
feed() (*método de xml.etree.ElementTree.XMLPullParser*), 1217
feed() (*método de xml.sax.xmlreader.IncrementalParser*), 1245
FeedParser (*clase en email.parser*), 1103
fetch() (*método de imaplib.IMAP4*), 1319
Fetch() (*método de msilib.View*), 2017
fetchall() (*método de sqlite3.Cursor*), 503
fetchmany() (*método de sqlite3.Cursor*), 503
fetchone() (*método de sqlite3.Cursor*), 502
fflags (*atributo de select.kevent*), 1074
Field (*clase en dataclasses*), 1789
field() (*en el módulo dataclasses*), 1788
field_size_limit() (*en el módulo csv*), 557
fieldnames (*atributo de csv.DictReader*), 560
fields (*atributo de uuid.UUID*), 1330
fields() (*en el módulo dataclasses*), 1789
file

byte-code, 1934, 2009
configuration, 562
copying, 454
debugger configuration, 1699
gzipopción de línea de comando, 519
.ini, 562
large files, 1971
mime.types, 1180
modes, 18
path configuration, 1844
.pdbrc, 1699
plist, 581
temporary, 446
file ...
compileallopción de línea de comando, 1936
file (*atributo de bdb.Breakpoint*), 1690
file (*atributo de pyclbr.Class*), 1933
file (*atributo de pyclbr.Function*), 1933
file control
UNIX, 1977
file name
temporary, 446
file object
io module, 657
open() built-in function, 18
--file=<file>
traceopción de línea de comando, 1717
FILE_ATTRIBUTE_ARCHIVE (*en el módulo stat*), 443
FILE_ATTRIBUTE_COMPRESSED (*en el módulo stat*), 443
FILE_ATTRIBUTE_DEVICE (*en el módulo stat*), 443
FILE_ATTRIBUTE_DIRECTORY (*en el módulo stat*), 443
FILE_ATTRIBUTE_ENCRYPTED (*en el módulo stat*), 443
FILE_ATTRIBUTE_HIDDEN (*en el módulo stat*), 443
FILE_ATTRIBUTE_INTEGRITY_STREAM (*en el módulo stat*), 443
FILE_ATTRIBUTE_NO_SCRUB_DATA (*en el módulo stat*), 443
FILE_ATTRIBUTE_NORMAL (*en el módulo stat*), 443
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED (*en el módulo stat*), 443
FILE_ATTRIBUTE_OFFLINE (*en el módulo stat*), 443
FILE_ATTRIBUTE_READONLY (*en el módulo stat*), 443
FILE_ATTRIBUTE_REPARSE_POINT (*en el módulo stat*), 443
FILE_ATTRIBUTE_SPARSE_FILE (*en el módulo stat*), 443
FILE_ATTRIBUTE_SYSTEM (*en el módulo stat*), 443
FILE_ATTRIBUTE_TEMPORARY (*en el módulo stat*), 443
FILE_ATTRIBUTE_VIRTUAL (*en el módulo stat*), 443
file_dispatcher (*clase en asyncore*), 1992
file_open() (*método de urllib.request.FileHandler*), 1281
file_size (*atributo de zipfile.ZipInfo*), 537
file_wrapper (*clase en asyncore*), 1993
filecmp
módulo, 444
fileConfig() (*en el módulo logging.config*), 731
FileCookieJar (*clase en http.cookiejar*), 1351
FileDialog (*clase en tkinter.filedialog*), 1471
FileEntry (*clase en tkinter.tix*), 1495
FileExistsError, 108
FileFinder (*clase en importlib.machinery*), 1875
FileHandler (*clase en logging*), 743
FileHandler (*clase en urllib.request*), 1274
fileinput
módulo, 437
FileInput (*clase en fileinput*), 438
FileIO (*clase en io*), 664
filelineno() (*en el módulo fileinput*), 438
FileLoader (*clase en importlib.abc*), 1870
filemode() (*en el módulo stat*), 440
filename (*atributo de doctest.DocTest*), 1563
filename (*atributo de http.cookiejar.FileCookieJar*), 1353
filename (*atributo de OSError*), 105
filename (*atributo de SyntaxError*), 106
filename (*atributo de traceback.TracebackException*), 1818
filename (*atributo de tracemalloc.Frame*), 1726
filename (*atributo de zipfile.ZipFile*), 534
filename (*atributo de zipfile.ZipInfo*), 536
filename() (*en el módulo fileinput*), 437
filename2 (*atributo de OSError*), 105
filename_only (*en el módulo tabnanny*), 1932
filename_pattern (*atributo de tracemalloc.Filter*), 1726
filenames
pathname expansion, 451
wildcard expansion, 452
fileno() (*en el módulo fileinput*), 438
fileno() (*método de http.client.HTTPResponse*), 1306
fileno() (*método de io.IOBase*), 661
fileno() (*método de multiprocessing.connection.Connection*), 848
fileno() (*método de ossaudiodev.oss_audio_device*), 2056
fileno() (*método de ossaudiodev.oss_mixer_device*), 2059
fileno() (*método de select.devpoll*), 1070
fileno() (*método de select.epoll*), 1071
fileno() (*método de select.kqueue*), 1073
fileno() (*método de selectors.DevpollSelector*), 1077
fileno() (*método de selectors.EpollSelector*), 1077
fileno() (*método de selectors.KqueueSelector*), 1077
fileno() (*método de socketserver.BaseServer*), 1335
fileno() (*método de socket.socket*), 1023

fileno() (*método de telnetlib.Telnet*), 2070
FileNotFoundError, 108
fileobj (*atributo de selectors.SelectorKey*), 1075
files() (*en el módulo importlib.resources*), 1872
files_double_event() (*método de tkinter.filedialog.FileDialog*), 1471
files_select_event() (*método de tkinter.filedialog.FileDialog*), 1471
FileSelectBox (*clase en tkinter.tix*), 1495
FileType (*clase en argparse*), 706
FileWrapper (*clase en wsgiref.util*), 1261
fill() (*en el módulo textwrap*), 155
fill() (*método de textwrap.TextWrapper*), 158
fillcolor() (*en el módulo turtle*), 1422
filling() (*en el módulo turtle*), 1423
filter (*2to3 fixer*), 1662
filter (*atributo de select.kevent*), 1073
Filter (*clase en logging*), 722
Filter (*clase en tracemalloc*), 1725
--filter <filtername>
    tarfileopción de línea de comando, 551
filter()
    función incorporada, 12
filter() (*en el módulo curses*), 758
filter() (*en el módulo fnmatch*), 453
filter() (*método de logging.Filter*), 722
filter() (*método de logging.Handler*), 719
filter() (*método de logging.Logger*), 717
filter_command() (*método de tkinter.filedialog.FileDialog*), 1471
FILTER_DIR (*en el módulo unittest.mock*), 1635
filter_traces() (*método de tracemalloc.Snapshot*), 1726
FilterError, 541
filterfalse() (*en el módulo itertools*), 386
filterwarnings() (*en el módulo warnings*), 1784
Final (*en el módulo typing*), 1524
final() (*en el módulo typing*), 1541
finalize (*clase en weakref*), 275
find() (*en el módulo gettext*), 1393
find() (*método de bytearray*), 63
find() (*método de bytes*), 63
find() (*método de doctest.DocTestFinder*), 1565
find() (*método de mmap.mmap*), 1089
find() (*método de str*), 50
find() (*método de xml.etree.ElementTree.Element*), 1211
find() (*método de xml.etree.ElementTree.ElementTree*), 1213
find_class() (*método de pickle.Unpickler*), 470
find_class() (*pickle protocol*), 480
find_library() (*en el módulo ctypes.util*), 813
find_loader() (*en el módulo importlib*), 1863
find_loader() (*en el módulo pkgutil*), 1856
find_loader() (*método de importlib.abc.PathEntryFinder*), 1866
find_loader() (*método de importlib.machinery.FileFinder*), 1876
find_loader() (*método de zipimport.zipimporter*), 1854
find_longest_match() (*método de difflib.SequenceMatcher*), 149
find_module() (*en el módulo imp*), 2010
find_module() (*método de clase de importlib.machinery.PathFinder*), 1875
find_module() (*método de imp.NullImporter*), 2013
find_module() (*método de importlib.abc.Finder*), 1865
find_module() (*método de importlib.abc.MetaPathFinder*), 1865
find_module() (*método de importlib.abc.PathEntryFinder*), 1866
find_module() (*método de zipimport.zipimporter*), 1854
find_msvcrt() (*en el módulo ctypes.util*), 813
find_spec() (*en el módulo importlib.util*), 1879
find_spec() (*método de clase de importlib.machinery.PathFinder*), 1875
find_spec() (*método de importlib.abc.MetaPathFinder*), 1865
find_spec() (*método de importlib.abc.PathEntryFinder*), 1866
find_spec() (*método de importlib.machinery.FileFinder*), 1876
find_spec() (*método de zipimport.zipimporter*), 1854
find_unused_port() (*en el módulo test.support.socket_helper*), 1677
find_user_password() (*método de urllib.request.HTTPPasswordMgr*), 1280
find_user_password() (*método de urllib.request.HTTPPasswordMgrWithPriorAuth*), 1280
findall() (*en el módulo re*), 132
findall() (*método de re.Pattern*), 136
findall() (*método de xml.etree.ElementTree.Element*), 1211
findall() (*método de xml.etree.ElementTree.ElementTree*), 1213
findCaller() (*método de logging.Logger*), 717
Finder (*clase en importlib.abc*), 1865
findfactor() (*en el módulo audioop*), 1995
findfile() (*en el módulo test.support*), 1670
findfit() (*en el módulo audioop*), 1995
finditer() (*en el módulo re*), 133
finditer() (*método de re.Pattern*), 136
findlabels() (*en el módulo dis*), 1943
findlinestarts() (*en el módulo dis*), 1942
findmatch() (*en el módulo mailcap*), 2014
findmax() (*en el módulo audioop*), 1995
findtext() (*método de xml.etree.ElementTree.Element*), 1211
findtext() (*método de xml.etree.ElementTree.ElementTree*), 1213
finish() (*método de socketser-*

ver.BaseRequestHandler), 1337
finish() (método de tkinter.dnd.DndHandler), 1474
finish_request() (método de socketserver.BaseServer), 1336
firstChild (atributo de xml.dom.Node), 1221
firstkey() (método de dbm.gnu.gdbm), 488
firstweekday() (en el módulo calendar), 237
fix_missing_locations() (en el módulo ast), 1919
fix_sentence_endings (atributo de textwrap.TextWrapper), 157
Flag (clase en enum), 295
flag_bits (atributo de zipfile.ZipInfo), 537
flags (atributo de re.Pattern), 136
flags (atributo de select.kevent), 1073
flags (en el módulo sys), 1754
flash() (en el módulo curses), 758
flatten() (método de email.generator.BytesGenerator), 1107
flatten() (método de email.generator.Generator), 1108
flattening
    objects, 465
float
    función incorporada, 35
float (clase incorporada), 13
float_info (en el módulo sys), 1755
float_repr_style (en el módulo sys), 1756
floating point
    literals, 35
    objeto, 35
FloatingPointError, 103
FloatOperation (clase en decimal), 350
flock() (en el módulo fcntl), 1978
floor() (en el módulo math), 321
floor() (in module math), 36
FloorDiv (clase en ast), 1896
floordiv() (en el módulo operator), 407
flush() (método de bz2.BZ2Compressor), 521
flush() (método de io.BufferedWriter), 666
flush() (método de io.IOBase), 661
flush() (método de logging.Handler), 719
flush() (método de logging.handlers.BufferingHandler), 752
flush() (método de logging.handlers.MemoryHandler), 752
flush() (método de logging.StreamHandler), 742
flush() (método de lzma.LZMACompressor), 526
flush() (método de mailbox.Mailbox), 1164
flush() (método de mailbox.Maildir), 1166
flush() (método de mailbox.MH), 1168
flush() (método de mmap.mmap), 1089
flush() (método de zlib.Compress), 515
flush() (método de zlib.Decompress), 516
flush_headers() (método de http.server.BaseHTTPRequestHandler), 1343
flushinp() (en el módulo curses), 758
FlushKey() (en el módulo winreg), 1961
fma() (método de decimal.Context), 345
fma() (método de decimal.Decimal), 339
fmean() (en el módulo statistics), 370
fmod() (en el módulo math), 321
FMT_BINARY (en el módulo plistlib), 583
FMT_XML (en el módulo plistlib), 582
fnmatch
    módulo, 452
fnmatch() (en el módulo fnmatch), 453
fnmatchcase() (en el módulo fnmatch), 453
focus() (método de tkinter.ttk.Treeview), 1487
fold (atributo de datetime.datetime), 206
fold (atributo de datetime.time), 214
fold() (método de email.headerregistry.BaseHeader), 1118
fold() (método de email.policy.Compat32), 1115
fold() (método de email.policy.EmailPolicy), 1114
fold() (método de email.policy.Policy), 1112
fold_binary() (método de email.policy.Compat32), 1115
fold_binary() (método de email.policy.EmailPolicy), 1114
fold_binary() (método de email.policy.Policy), 1113
Font (clase en tkinter.font), 1468
For (clase en ast), 1905
FOR_ITER (opcode), 1950
forget() (en el módulo test.support.import_helper), 1682
forget() (método de tkinter.ttk.Notebook), 1482
fork() (en el módulo os), 643
fork() (en el módulo pty), 1975
ForkingMixIn (clase en socketserver), 1334
ForkingTCPServer (clase en socketserver), 1334
ForkingUDPServer (clase en socketserver), 1334
forkpty() (en el módulo os), 644
Form (clase en tkinter.tix), 1496
format (atributo de memoryview), 80
format (atributo de multiprocessing.shared_memory.ShareableList), 881
format (atributo de struct.Struct), 173
format()
    función incorporada, 13
format() (en el módulo locale), 1404
format() (método de logging.BufferingFormatter), 721
format() (método de logging.Formatter), 720
format() (método de logging.Handler), 719
format() (método de pprint.PrettyPrinter), 289
format() (método de str), 50
format() (método de string.Formatter), 114
format() (método de traceback.StackSummary), 1819
format() (método de traceback.TracebackException), 1819
format() (método de tracemalloc.Traceback), 1729
format_datetime() (en el módulo email.utils), 1150
format_exc() (en el módulo traceback), 1817

format_exception() (*en el módulo traceback*), 1817
format_exception_only() (*en el módulo traceback*), 1817
format_exception_only() (*método de traceback.TracebackException*), 1819
format_field() (*método de string.Formatter*), 115
format_help() (*método de argparse.ArgumentParser*), 709
format_list() (*en el módulo traceback*), 1817
format_map() (*método de str*), 51
format_stack() (*en el módulo traceback*), 1817
format_stack_entry() (*método de bdb.Bdb*), 1693
format_string() (*en el módulo locale*), 1404
format_tb() (*en el módulo traceback*), 1817
format_usage() (*método de argparse.ArgumentParser*), 709
FORMAT_VALUE (*opcode*), 1952
formataddr() (*en el módulo email.utils*), 1149
formatargspec() (*en el módulo inspect*), 1838
formatargvalues() (*en el módulo inspect*), 1838
formatdate() (*en el módulo email.utils*), 1149
FormatError, 1177
FormatError() (*en el módulo ctypes*), 813
formatException() (*método de logging.Formatter*), 721
formatFooter() (*método de logging.BufferingFormatter*), 721
formatHeader() (*método de logging.BufferingFormatter*), 721
formatmonth() (*método de calendar.HTMLCalendar*), 236
formatmonth() (*método de calendar.TextCalendar*), 235
formatStack() (*método de logging.Formatter*), 721
FormattedValue (*clase en ast*), 1893
Formatter (*clase en logging*), 720
Formatter (*clase en string*), 114
formatTime() (*método de logging.Formatter*), 720
formatting
    bytearray (%), 72
    bytes (%), 72
formatting, string (%), 57
formatwarning() (*en el módulo warnings*), 1784
formatyear() (*método de calendar.HTMLCalendar*), 236
formatyear() (*método de calendar.TextCalendar*), 235
formatyearpage() (*método de calendar.HTMLCalendar*), 236
Fortran contiguous, 2082
forward() (*en el módulo turtle*), 1414
ForwardRef (*clase en typing*), 1543
found_terminator() (*método de asynchat.async_chat*), 1988
fpathconf() (*en el módulo os*), 610
fqdn (*atributo de smtpd.SMTPChannel*), 2064
Fraction (*clase en fractions*), 358
fractions
    módulo, 358
frame (*atributo de tkinter.scrolledtext.ScrolledText*), 1473
Frame (*clase en tracemalloc*), 1726
FrameSummary (*clase en traceback*), 1820
FrameType (*en el módulo types*), 283
freedesktop_os_release() (*en el módulo platform*), 782
freeze() (*en el módulo gc*), 1826
freeze_support() (*en el módulo multiprocessing*), 847
frexp() (*en el módulo math*), 321
FRIDAY (*en el módulo calendar*), 238
from_address() (*método de ctypes._CData*), 815
from_buffer() (*método de ctypes._CData*), 815
from_buffer_copy() (*método de ctypes._CData*), 815
from_bytes() (*método de clase de int*), 38
from_callable() (*método de clase de inspect.Signature*), 1834
from_decimal() (*método de clase de fractions.Fraction*), 359
from_exception() (*método de clase de traceback.TracebackException*), 1819
from_file() (*método de clase de zipfile.ZipInfo*), 536
from_file() (*método de clase de zoneinfo.ZoneInfo*), 231
from_float() (*método de clase de decimal.Decimal*), 338
from_float() (*método de clase de fractions.Fraction*), 359
from_iterable() (*método de clase de itertools.chain*), 384
from_list() (*método de clase de traceback.StackSummary*), 1819
from_param() (*método de ctypes._CData*), 815
from_samples() (*método de clase de statistics.NormalDist*), 377
from_traceback() (*método de clase de dis.Bytecode*), 1940
frombuf() (*método de clase de tarfile.TarInfo*), 546
frombytes() (*método de array.array*), 271
fromfd() (*en el módulo socket*), 1017
fromfd() (*método de select.epoll*), 1071
fromfd() (*método de select.kqueue*), 1073
fromfile() (*método de array.array*), 271
fromhex() (*método de clase de bytearray*), 61
fromhex() (*método de clase de bytes*), 60
fromhex() (*método de clase de float*), 39
fromisocalendar() (*método de clase de datetime.date*), 199
fromisocalendar() (*método de clase de datetime.datetime*), 205
fromisoformat() (*método de clase de datetime.date*), 199

fromisoformat() (*método de clase de datetime.datetime*), 205
fromisoformat() (*método de clase de datetime.time*), 214
fromkeys() (*método de clase de dict*), 85
fromkeys() (*método de collections.Counter*), 243
fromlist() (*método de array.array*), 271
fromordinal() (*método de clase de datetime.date*), 199
fromordinal() (*método de clase de datetime.datetime*), 205
fromshare() (*en el módulo socket*), 1017
fromstring() (*en el módulo xml.etree.ElementTree*), 1207
fromstringlist() (*en el módulo xml.etree.ElementTree*), 1207
fromtarfile() (*método de clase de tarfile.TarInfo*), 546
fromtimestamp() (*método de clase de datetime.date*), 199
fromtimestamp() (*método de clase de datetime.datetime*), 204
fromunicode() (*método de array.array*), 271
fromutc() (*método de datetime.timezone*), 224
fromutc() (*método de datetime.tzinfo*), 218
FrozenImporter (*clase en importlib.machinery*), 1874
FrozenInstanceError, 1791
FrozenSet (*clase en typing*), 1534
frozenset (*clase incorporada*), 81
fs_is_case_insensitive() (*en el módulo test.support.os_helper*), 1681
FS_NONASCII (*en el módulo test.support.os_helper*), 1680
fsdecode() (*en el módulo os*), 604
fsencode() (*en el módulo os*), 604
fspath() (*en el módulo os*), 604
fstat() (*en el módulo os*), 611
fstatvfs() (*en el módulo os*), 611
fsum() (*en el módulo math*), 321
fsync() (*en el módulo os*), 611
FTP, 1287
    ftplib (*standard module*), 1308
    protocol, 1287, 1308
FTP (*clase en ftplib*), 1308
ftp_open() (*método de urllib.request.FTPHandler*), 1282
FTP_TLS (*clase en ftplib*), 1309
FTPHandler (*clase en urllib.request*), 1275
ftplib
    módulo, 1308
ftruncate() (*en el módulo os*), 611
Full, 910
full() (*método de asyncio.Queue*), 950
full() (*método de multiprocessing.Queue*), 845
full() (*método de queue.Queue*), 910
full_url (*atributo de urllib.request.Request*), 1275
fullmatch() (*en el módulo re*), 132
fullmatch() (*método de re.Pattern*), 135
fully_trusted_filter() (*en el módulo tarfile*), 548
func (*atributo de functools.partial*), 405
funcattrs (*2to3 fixer*), 1663
función, **2084**
función clave, **2086**
función corrutina, **2082**
función genérica, **2084**
función incorporada
    __import__(), 28
    abs(), 6
    aiter(), 6
    all(), 6
    anext(), 6
    any(), 7
    ascii(), 7
    bin(), 7
    breakpoint(), 7
    callable(), 8
    chr(), 8
    classmethod(), 8
    compile, 96, 281
    compile(), 9
    complex, 35
    delattr(), 10
    dir(), 10
    divmod(), 11
    enumerate(), 11
    eval, 96, 289
    eval(), 11
    exec, 12, 96
    exec(), 12
    filter(), 12
    float, 35
    format(), 13
    getattr(), 14
    globals(), 14
    hasattr(), 14
    hash, 44
    hash(), 14
    help(), 14
    hex(), 14
    id(), 15
    input(), 15
    int, 35
    isinstance(), 16
    issubclass(), 16
    iter(), 16
    len, 42, 83
    len(), 16
    locals(), 17
    map(), 17
    max, 42
    max(), 17
    min, 42
    min(), 17
    multiprocessing.Manager(), 855

    next(), 17
    oct(), 18
    open(), 18
    ord(), 21
    pow(), 21
    print(), 21
    repr(), 23
    reversed(), 23
    round(), 23
    setattr(), 23
    slice, 1952
    sorted(), 24
    staticmethod(), 24
    sum(), 25
    type, 96
    vars(), 26
    xml.etree.ElementInclude.default_loader(), 1210
    xml.etree.ElementInclude.include(), 1210
    zip(), 27
funcname (*atributo de bdb.Breakpoint*), 1690
Function (*clase en symtable*), 1922
FunctionDef (*clase en ast*), 1913
FunctionTestCase (*clase en unittest*), 1590
FunctionType (*en el módulo types*), 281
functools
    módulo, 396
funny_files (*atributo de filecmp.dircmp*), 445
future (*2to3 fixer*), 1663
Future (*clase en asyncio*), 976
Future (*clase en concurrent.futures*), 886
FutureWarning, 109
fwalk() (*en el módulo os*), 637

G

-g
    traceopción de línea de comando, 1717
G.722, 1987
gaierror, 1011
gamma() (*en el módulo math*), 326
gammavariate() (*en el módulo random*), 364
gancho a entrada de ruta, **2089**
garbage (*en el módulo gc*), 1826
gather() (*en el módulo asyncio*), 925
gather() (*método de curses.textpad.Textbox*), 775
gauss() (*en el módulo random*), 364
gc
    módulo, 1824
gc_collect() (*en el módulo test.support*), 1672
gcd() (*en el módulo math*), 322
ge() (*en el módulo operator*), 406
GEN_START (*opcode*), 1952
gen_uuid() (*en el módulo msilib*), 2016
generador, **2084**
generador asincrónico, **2080**
generate_tokens() (*en el módulo tokenize*), 1928
generator, 2084
Generator (*clase en collections.abc*), 260
Generator (*clase en email.generator*), 1107
Generator (*clase en typing*), 1538
generator expression, 2084
GeneratorExit, 103
GeneratorExp (*clase en ast*), 1899
GeneratorType (*en el módulo types*), 281
Generic
    Alias, 89
Generic (*clase en typing*), 1526
generic_visit() (*método de ast.NodeVisitor*), 1919
GenericAlias
    objeto, 89
GenericAlias (*clase en types*), 283
genops() (*en el módulo pickletools*), 1954
geometric_mean() (*en el módulo statistics*), 370
get() (*en el módulo webbrowser*), 1258
get() (*método de asyncio.Queue*), 950
get() (*método de configparser.ConfigParser*), 577
get() (*método de contextvars.Context*), 915
get() (*método de contextvars.ContextVar*), 913
get() (*método de dict*), 85
get() (*método de email.message.EmailMessage*), 1097
get() (*método de email.message.Message*), 1135
get() (*método de mailbox.Mailbox*), 1163
get() (*método de multiprocessing.pool.AsyncResult*), 863
get() (*método de multiprocessing.Queue*), 845
get() (*método de multiprocessing.SimpleQueue*), 846
get() (*método de ossaudiodev.oss_mixer_device*), 2059
get() (*método de queue.Queue*), 911
get() (*método de queue.SimpleQueue*), 912
get() (*método de tkinter.ttk.Combobox*), 1479
get() (*método de tkinter.ttk.Spinbox*), 1480
get() (*método de types.MappingProxyType*), 284
get() (*método de xml.etree.ElementTree.Element*), 1211
GET_AITER (*opcode*), 1946
get_all() (*método de email.message.EmailMessage*), 1097
get_all() (*método de email.message.Message*), 1135
get_all() (*método de wsgiref.headers.Headers*), 1262
get_all_breaks() (*método de bdb.Bdb*), 1693
get_all_start_methods() (*en el módulo multiprocessing*), 847
GET_ANEXT (*opcode*), 1946
get_annotations() (*en el módulo inspect*), 1839
get_app() (*método de wsgiref.simple_server.WSGIServer*), 1264
get_archive_formats() (*en el módulo shutil*), 460
get_args() (*en el módulo typing*), 1542
get_asyncgen_hooks() (*en el módulo sys*), 1758
get_attribute() (*en el módulo test.support*), 1674
GET_AWAITABLE (*opcode*), 1946
get_begidx() (*en el módulo readline*), 164

get_blocking() (*en el módulo os*), 611
get_body() (*método de email.message.EmailMessage*), 1100
get_body_encoding() (*método de email.charset.Charset*), 1146
get_boundary() (*método de email.message.EmailMessage*), 1099
get_boundary() (*método de email.message.Message*), 1138
get_bpbynumber() (*método de bdb.Bdb*), 1693
get_break() (*método de bdb.Bdb*), 1693
get_breaks() (*método de bdb.Bdb*), 1693
get_buffer() (*método de asyncio.BufferedProtocol*), 985
get_buffer() (*método de xdrlib.Packer*), 2072
get_buffer() (*método de xdrlib.Unpacker*), 2073
get_bytes() (*método de mailbox.Mailbox*), 1163
get_ca_certs() (*método de ssl.SSLContext*), 1052
get_cache_token() (*en el módulo abc*), 1814
get_channel_binding() (*método de ssl.SSLSocket*), 1049
get_charset() (*método de email.message.Message*), 1134
get_charsets() (*método de email.message.EmailMessage*), 1099
get_charsets() (*método de email.message.Message*), 1138
get_child_watcher() (*en el módulo asyncio*), 993
get_child_watcher() (*método de asyncio.AbstractEventLoopPolicy*), 992
get_children() (*método de symtable.SymbolTable*), 1922
get_children() (*método de tkinter.ttk.Treeview*), 1486
get_ciphers() (*método de ssl.SSLContext*), 1052
get_clock_info() (*en el módulo time*), 672
get_close_matches() (*en el módulo difflib*), 146
get_code() (*método de importlib.abc.InspectLoader*), 1869
get_code() (*método de importlib.abc.SourceLoader*), 1871
get_code() (*método de importlib.machinery.ExtensionFileLoader*), 1877
get_code() (*método de importlib.machinery.SourcelessFileLoader*), 1877
get_code() (*método de zipimport.zipimporter*), 1854
get_completer() (*en el módulo readline*), 164
get_completer_delims() (*en el módulo readline*), 164
get_completion_type() (*en el módulo readline*), 164
get_config_h_filename() (*en el módulo sysconfig*), 1772
get_config_var() (*en el módulo sysconfig*), 1769
get_config_vars() (*en el módulo sysconfig*), 1769
get_content() (*en el módulo email.contentmanager*), 1124
get_content() (*método de email.contentmanager.ContentManager*), 1123
get_content() (*método de email.message.EmailMessage*), 1101
get_content_charset() (*método de email.message.EmailMessage*), 1099
get_content_charset() (*método de email.message.Message*), 1138
get_content_disposition() (*método de email.message.EmailMessage*), 1099
get_content_disposition() (*método de email.message.Message*), 1138
get_content_maintype() (*método de email.message.EmailMessage*), 1098
get_content_maintype() (*método de email.message.Message*), 1136
get_content_subtype() (*método de email.message.EmailMessage*), 1098
get_content_subtype() (*método de email.message.Message*), 1136
get_content_type() (*método de email.message.EmailMessage*), 1097
get_content_type() (*método de email.message.Message*), 1136
get_context() (*en el módulo multiprocessing*), 847
get_coro() (*método de asyncio.Task*), 933
get_coroutine_origin_tracking_depth() (*en el módulo sys*), 1758
get_count() (*en el módulo gc*), 1825
get_current_history_length() (*en el módulo readline*), 163
get_data() (*en el módulo pkgutil*), 1857
get_data() (*método de importlib.abc.FileLoader*), 1870
get_data() (*método de importlib.abc.ResourceLoader*), 1868
get_data() (*método de zipimport.zipimporter*), 1854
get_date() (*método de mailbox.MaildirMessage*), 1171
get_debug() (*en el módulo gc*), 1824
get_debug() (*método de asyncio.loop*), 969
get_default() (*método de argparse.ArgumentParser*), 708
get_default_domain() (*en el módulo nis*), 2021
get_default_scheme() (*en el módulo sysconfig*), 1770
get_default_type() (*método de email.message.EmailMessage*), 1098
get_default_type() (*método de email.message.Message*), 1136
get_default_verify_paths() (*en el módulo ssl*), 1039
get_dialect() (*en el módulo csv*), 557
get_disassembly_as_string() (*método de test.support.bytecode_helper.BytecodeTestCase*), 1679

get_docstring() (*en el módulo ast*), 1919
get_doctest() (*método de doctest.DocTestParser*), 1565
get_endidx() (*en el módulo readline*), 164
get_environ() (*método de wsgiref.simple_server.WSGIRequestHandler*), 1264
get_errno() (*en el módulo ctypes*), 813
get_escdelay() (*en el módulo curses*), 761
get_event_loop() (*en el módulo asyncio*), 953
get_event_loop() (*método de asyncio.AbstractEventLoopPolicy*), 992
get_event_loop_policy() (*en el módulo asyncio*), 992
get_examples() (*método de doctest.DocTestParser*), 1565
get_exception_handler() (*método de asyncio.loop*), 968
get_exec_path() (*en el módulo os*), 605
get_extra_info() (*método de asyncio.BaseTransport*), 980
get_extra_info() (*método de asyncio.StreamWriter*), 938
get_field() (*método de string.Formatter*), 114
get_file() (*método de mailbox.Babyl*), 1169
get_file() (*método de mailbox.Mailbox*), 1163
get_file() (*método de mailbox.Maildir*), 1166
get_file() (*método de mailbox.mbox*), 1166
get_file() (*método de mailbox.MH*), 1168
get_file() (*método de mailbox.MMDF*), 1169
get_file_breaks() (*método de bdb.Bdb*), 1693
get_filename() (*método de email.message.EmailMessage*), 1098
get_filename() (*método de email.message.Message*), 1138
get_filename() (*método de importlib.abc.ExecutionLoader*), 1869
get_filename() (*método de importlib.abc.FileLoader*), 1870
get_filename() (*método de importlib.machinery.ExtensionFileLoader*), 1877
get_filename() (*método de zipimport.zipimporter*), 1854
get_filter() (*método de tkinter.filedialog.FileDialog*), 1471
get_flags() (*método de mailbox.MaildirMessage*), 1171
get_flags() (*método de mailbox.mboxMessage*), 1172
get_flags() (*método de mailbox.MMDFMessage*), 1176
get_folder() (*método de mailbox.Maildir*), 1165
get_folder() (*método de mailbox.MH*), 1167
get_frees() (*método de symtable.Function*), 1923
get_freeze_count() (*en el módulo gc*), 1826
get_from() (*método de mailbox.mboxMessage*), 1172
get_from() (*método de mailbox.MMDFMessage*), 1176
get_full_url() (*método de urllib.request.Request*), 1276
get_globals() (*método de symtable.Function*), 1923
get_grouped_opcodes() (*método de difflib.SequenceMatcher*), 150
get_handle_inheritable() (*en el módulo os*), 619
get_header() (*método de urllib.request.Request*), 1276
get_history_item() (*en el módulo readline*), 163
get_history_length() (*en el módulo readline*), 163
get_id() (*método de symtable.SymbolTable*), 1922
get_ident() (*en el módulo _thread*), 917
get_ident() (*en el módulo threading*), 822
get_identifiers() (*método de symtable.SymbolTable*), 1922
get_importer() (*en el módulo pkgutil*), 1856
get_info() (*método de mailbox.MaildirMessage*), 1171
get_inheritable() (*en el módulo os*), 619
get_inheritable() (*método de socket.socket*), 1023
get_instructions() (*en el módulo dis*), 1942
get_int_max_str_digits() (*en el módulo sys*), 1757
get_interpreter() (*en el módulo zipapp*), 1744
GET_ITER (*opcode*), 1944
get_key() (*método de selectors.BaseSelector*), 1077
get_labels() (*método de mailbox.Babyl*), 1168
get_labels() (*método de mailbox.BabylMessage*), 1174
get_last_error() (*en el módulo ctypes*), 814
GET_LEN (*opcode*), 1947
get_line_buffer() (*en el módulo readline*), 162
get_lineno() (*método de symtable.SymbolTable*), 1922
get_loader() (*en el módulo pkgutil*), 1856
get_locals() (*método de symtable.Function*), 1922
get_logger() (*en el módulo multiprocessing*), 867
get_loop() (*método de asyncio.Future*), 977
get_loop() (*método de asyncio.Server*), 971
get_magic() (*en el módulo imp*), 2009
get_makefile_filename() (*en el módulo sysconfig*), 1772
get_map() (*método de selectors.BaseSelector*), 1077
get_matching_blocks() (*método de difflib.SequenceMatcher*), 150
get_message() (*método de mailbox.Mailbox*), 1163
get_method() (*método de urllib.request.Request*), 1275
get_methods() (*método de symtable.Class*), 1923
get_mixed_type_key() (*en el módulo ipaddress*), 1385
get_name() (*método de asyncio.Task*), 933
get_name() (*método de symtable.Symbol*), 1923
get_name() (*método de symtable.SymbolTable*), 1922

get_namespace() (*método de symtable.Symbol*), 1924
get_namespaces() (*método de symtable.Symbol*), 1924
get_native_id() (*en el módulo _thread*), 917
get_native_id() (*en el módulo threading*), 822
get_nonlocals() (*método de symtable.Function*), 1923
get_nonstandard_attr() (*método de http.cookiejar.Cookie*), 1358
get_nowait() (*método de asyncio.Queue*), 950
get_nowait() (*método de multiprocessing.Queue*), 845
get_nowait() (*método de queue.Queue*), 911
get_nowait() (*método de queue.SimpleQueue*), 912
get_object_traceback() (*en el módulo tracemalloc*), 1724
get_objects() (*en el módulo gc*), 1824
get_opcodes() (*método de difflib.SequenceMatcher*), 150
get_option() (*método de optparse.OptionParser*), 2047
get_option_group() (*método de optparse.OptionParser*), 2037
get_origin() (*en el módulo typing*), 1542
get_original_stdout() (*en el módulo test.support*), 1671
get_osfhandle() (*en el módulo msvcrt*), 1958
get_output_charset() (*método de email.charset.Charset*), 1146
get_param() (*método de email.message.Message*), 1137
get_parameters() (*método de symtable.Function*), 1922
get_params() (*método de email.message.Message*), 1136
get_path() (*en el módulo sysconfig*), 1771
get_path_names() (*en el módulo sysconfig*), 1771
get_paths() (*en el módulo sysconfig*), 1771
get_payload() (*método de email.message.Message*), 1133
get_pid() (*método de asyncio.SubprocessTransport*), 983
get_pipe_transport() (*método de asyncio.SubprocessTransport*), 983
get_platform() (*en el módulo sysconfig*), 1771
get_poly() (*en el módulo turtle*), 1429
get_position() (*método de xdrlib.Unpacker*), 2073
get_preferred_scheme() (*en el módulo sysconfig*), 1770
get_protocol() (*método de asyncio.BaseTransport*), 981
get_python_version() (*en el módulo sysconfig*), 1771
get_ready() (*método de graphlib.TopologicalSorter*), 314
get_recsrc() (*método de ossaudiodev.oss_mixer_device*), 2059
get_referents() (*en el módulo gc*), 1825
get_referrers() (*en el módulo gc*), 1825
get_request() (*método de socketserver.BaseServer*), 1336
get_returncode() (*método de asyncio.SubprocessTransport*), 983
get_running_loop() (*en el módulo asyncio*), 953
get_scheme() (*método de wsgiref.handlers.BaseHandler*), 1267
get_scheme_names() (*en el módulo sysconfig*), 1770
get_selection() (*método de tkinter.filedialog.FileDialog*), 1471
get_sequences() (*método de mailbox.MH*), 1167
get_sequences() (*método de mailbox.MHMessage*), 1173
get_server() (*método de multiprocessing.managers.BaseManager*), 855
get_server_certificate() (*en el módulo ssl*), 1038
get_shapepoly() (*en el módulo turtle*), 1427
get_socket() (*método de telnetlib.Telnet*), 2070
get_source() (*método de importlib.abc.InspectLoader*), 1869
get_source() (*método de importlib.abc.SourceLoader*), 1871
get_source() (*método de importlib.machinery.ExtensionFileLoader*), 1877
get_source() (*método de importlib.machinery.SourcelessFileLoader*), 1877
get_source() (*método de zipimport.zipimporter*), 1854
get_source_segment() (*en el módulo ast*), 1919
get_stack() (*método de asyncio.Task*), 933
get_stack() (*método de bdb.Bdb*), 1693
get_start_method() (*en el módulo multiprocessing*), 847
get_starttag_text() (*método de html.parser.HTMLParser*), 1193
get_stats() (*en el módulo gc*), 1824
get_stats_profile() (*método de pstats.Stats*), 1709
get_stderr() (*método de wsgiref.handlers.BaseHandler*), 1266
get_stderr() (*método de wsgiref.simple_server.WSGIRequestHandler*), 1264
get_stdin() (*método de wsgiref.handlers.BaseHandler*), 1266
get_string() (*método de mailbox.Mailbox*), 1163
get_subdir() (*método de mailbox.MaildirMessage*), 1170
get_suffixes() (*en el módulo imp*), 2009
get_symbols() (*método de symtable.SymbolTable*), 1922
get_tabsize() (*en el módulo curses*), 761

get_tag() (*en el módulo imp*), 2012
get_task_factory() (*método de asyncio.loop*), 958
get_terminal_size() (*en el módulo os*), 618
get_terminal_size() (*en el módulo shutil*), 463
get_terminator() (*método de asynchat.async_chat*), 1988
get_threshold() (*en el módulo gc*), 1825
get_token() (*método de shlex.shlex*), 1450
get_traceback_limit() (*en el módulo tracemalloc*), 1724
get_traced_memory() (*en el módulo tracemalloc*), 1724
get_tracemalloc_memory() (*en el módulo tracemalloc*), 1724
get_type() (*método de symtable.SymbolTable*), 1922
get_type_hints() (*en el módulo typing*), 1542
get_unixfrom() (*método de email.message.EmailMessage*), 1096
get_unixfrom() (*método de email.message.Message*), 1133
get_unpack_formats() (*en el módulo shutil*), 462
get_usage() (*método de optparse.OptionParser*), 2048
get_value() (*método de string.Formatter*), 114
get_version() (*método de optparse.OptionParser*), 2038
get_visible() (*método de mailbox.BabylMessage*), 1175
get_wch() (*método de curses.window*), 766
get_write_buffer_limits() (*método de asyncio.WriteTransport*), 981
get_write_buffer_size() (*método de asyncio.WriteTransport*), 981
GET_YIELD_FROM_ITER (*opcode*), 1944
getacl() (*método de imaplib.IMAP4*), 1319
getaddresses() (*en el módulo email.utils*), 1149
getaddrinfo() (*en el módulo socket*), 1017
getaddrinfo() (*método de asyncio.loop*), 965
getallocatedblocks() (*en el módulo sys*), 1756
getandroidapilevel() (*en el módulo sys*), 1756
getannotation() (*método de imaplib.IMAP4*), 1319
getargspec() (*en el módulo inspect*), 1837
getargvalues() (*en el módulo inspect*), 1837
getatime() (*en el módulo os.path*), 433
getattr()
    función incorporada, 14
getattr_static() (*en el módulo inspect*), 1841
getAttribute() (*método de xml.dom.Element*), 1224
getAttributeNode() (*método de xml.dom.Element*), 1224
getAttributeNodeNS() (*método de xml.dom.Element*), 1224
getAttributeNS() (*método de xml.dom.Element*), 1224
GetBase() (*método de xml.parsers.expat.xmlparser*), 1248
getbegyx() (*método de curses.window*), 766
getbkgd() (*método de curses.window*), 766
getblocking() (*método de socket.socket*), 1024
getboolean() (*método de configparser.ConfigParser*), 577
getbuffer() (*método de io.BytesIO*), 664
getByteStream() (*método de xml.sax.xmlreader.InputSource*), 1246
getcallargs() (*en el módulo inspect*), 1838
getcanvas() (*en el módulo turtle*), 1435
getcapabilities() (*método de nntplib.NNTP*), 2024
getcaps() (*en el módulo mailcap*), 2015
getch() (*en el módulo msvcrt*), 1958
getch() (*método de curses.window*), 766
getCharacterStream() (*método de xml.sax.xmlreader.InputSource*), 1246
getche() (*en el módulo msvcrt*), 1958
getChild() (*método de logging.Logger*), 715
getclasstree() (*en el módulo inspect*), 1837
getclosurevars() (*en el módulo inspect*), 1839
getcode() (*método de http.client.HTTPResponse*), 1306
getcode() (*método de urllib.response.addinfourl*), 1288
GetColumnInfo() (*método de msilib.View*), 2017
getColumnNumber() (*método de xml.sax.xmlreader.Locator*), 1245
getcomments() (*en el módulo inspect*), 1831
getcompname() (*método de aifc.aifc*), 1986
getcompname() (*método de sunau.AU_read*), 2067
getcompname() (*método de wave.Wave_read*), 1388
getcomptype() (*método de aifc.aifc*), 1986
getcomptype() (*método de sunau.AU_read*), 2067
getcomptype() (*método de wave.Wave_read*), 1388
getContentHandler() (*método de xml.sax.xmlreader.XMLReader*), 1244
getcontext() (*en el módulo decimal*), 342
getcoroutinelocals() (*en el módulo inspect*), 1843
getcoroutinestate() (*en el módulo inspect*), 1842
getctime() (*en el módulo os.path*), 433
getcwd() (*en el módulo os*), 622
getcwdb() (*en el módulo os*), 622
getcwdu (*2to3 fixer*), 1663
getdecoder() (*en el módulo codecs*), 175
getdefaultencoding() (*en el módulo sys*), 1756
getdefaultlocale() (*en el módulo locale*), 1403
getdefaulttimeout() (*en el módulo socket*), 1021
getdlopenflags() (*en el módulo sys*), 1756
getdoc() (*en el módulo inspect*), 1831
getDOMImplementation() (*en el módulo xml.dom*), 1219
getDTDHandler() (*método de xml.sax.xmlreader.XMLReader*), 1244

getEffectiveLevel() (*método de logging.Logger*), 715
getegid() (*en el módulo os*), 605
getElementsByTagName() (*método de xml.dom.Document*), 1223
getElementsByTagName() (*método de xml.dom.Element*), 1224
getElementsByTagNameNS() (*método de xml.dom.Document*), 1223
getElementsByTagNameNS() (*método de xml.dom.Element*), 1224
getencoder() (*en el módulo codecs*), 175
getEncoding() (*método de xml.sax.xmlreader.InputSource*), 1246
getEntityResolver() (*método de xml.sax.xmlreader.XMLReader*), 1244
getenv() (*en el módulo os*), 604
getenvb() (*en el módulo os*), 604
getErrorHandler() (*método de xml.sax.xmlreader.XMLReader*), 1244
geteuid() (*en el módulo os*), 605
getEvent() (*método de xml.dom.pulldom.DOMEventStream*), 1234
getEventCategory() (*método de logging.handlers.NTEventLogHandler*), 751
getEventType() (*método de logging.handlers.NTEventLogHandler*), 751
getException() (*método de xml.sax.SAXException*), 1236
getFeature() (*método de xml.sax.xmlreader.XMLReader*), 1244
GetFieldCount() (*método de msilib.Record*), 2018
getfile() (*en el módulo inspect*), 1831
getFilesToDelete() (*método de logging.handlers.TimedRotatingFileHandler*), 747
getfilesystemencodeerrors() (*en el módulo sys*), 1756
getfilesystemencoding() (*en el módulo sys*), 1756
getfirst() (*método de cgi.FieldStorage*), 2000
getfloat() (*método de configparser.ConfigParser*), 577
getfmts() (*método de ossaudiodev.oss_audio_device*), 2057
getfqdn() (*en el módulo socket*), 1018
getframeinfo() (*en el módulo inspect*), 1840
getframerate() (*método de aifc.aifc*), 1986
getframerate() (*método de sunau.AU_read*), 2067
getframerate() (*método de wave.Wave_read*), 1388
getfullargspec() (*en el módulo inspect*), 1837
getgeneratorlocals() (*en el módulo inspect*), 1842
getgeneratorstate() (*en el módulo inspect*), 1842
getgid() (*en el módulo os*), 605
getgrall() (*en el módulo grp*), 1973
getgrgid() (*en el módulo grp*), 1973
getgrnam() (*en el módulo grp*), 1973
getgrouplist() (*en el módulo os*), 605
getgroups() (*en el módulo os*), 605
getheader() (*método de http.client.HTTPResponse*), 1306
getheaders() (*método de http.client.HTTPResponse*), 1306
gethostbyaddr() (*en el módulo socket*), 1018
gethostbyaddr() (*in module socket*), 608
gethostbyname() (*en el módulo socket*), 1018
gethostbyname_ex() (*en el módulo socket*), 1018
gethostname() (*en el módulo socket*), 1018
gethostname() (*in module socket*), 608
getincrementaldecoder() (*en el módulo codecs*), 175
getincrementalencoder() (*en el módulo codecs*), 175
getinfo() (*método de zipfile.ZipFile*), 531
getinnerframes() (*en el módulo inspect*), 1841
GetInputContext() (*método de xml.parsers.expat.xmlparser*), 1248
getint() (*método de configparser.ConfigParser*), 577
GetInteger() (*método de msilib.Record*), 2018
getitem() (*en el módulo operator*), 408
getitimer() (*en el módulo signal*), 1083
getkey() (*método de curses.window*), 766
GetLastError() (*en el módulo ctypes*), 813
getLength() (*método de xml.sax.xmlreader.Attributes*), 1246
getLevelName() (*en el módulo logging*), 727
getline() (*en el módulo linecache*), 453
getLineNumber() (*método de xml.sax.xmlreader.Locator*), 1245
getlist() (*método de cgi.FieldStorage*), 2001
getloadavg() (*en el módulo os*), 654
getlocale() (*en el módulo locale*), 1403
getLogger() (*en el módulo logging*), 725
getLoggerClass() (*en el módulo logging*), 725
getlogin() (*en el módulo os*), 605
getLogRecordFactory() (*en el módulo logging*), 725
getmark() (*método de aifc.aifc*), 1986
getmark() (*método de sunau.AU_read*), 2067
getmark() (*método de wave.Wave_read*), 1388
getmarkers() (*método de aifc.aifc*), 1986
getmarkers() (*método de sunau.AU_read*), 2067
getmarkers() (*método de wave.Wave_read*), 1388
getmaxyx() (*método de curses.window*), 766
getmember() (*método de tarfile.TarFile*), 543
getmembers() (*en el módulo inspect*), 1829
getmembers() (*método de tarfile.TarFile*), 543
getMessage() (*método de logging.LogRecord*), 723
getMessage() (*método de xml.sax.SAXException*), 1236
getMessageID() (*método de logging.handlers.NTEventLogHandler*), 751
getmodule() (*en el módulo inspect*), 1832

getmodulename() (*en el módulo inspect*), 1829
getmouse() (*en el módulo curses*), 758
getmro() (*en el módulo inspect*), 1838
getmtime() (*en el módulo os.path*), 433
getname() (*método de chunk.Chunk*), 2006
getName() (*método de threading.Thread*), 825
getNameByQName() (*método de xml.sax.xmlreader.AttributesNS*), 1247
getnameinfo() (*en el módulo socket*), 1019
getnameinfo() (*método de asyncio.loop*), 965
getnames() (*método de tarfile.TarFile*), 543
getNames() (*método de xml.sax.xmlreader.Attributes*), 1246
getnchannels() (*método de aifc.aifc*), 1986
getnchannels() (*método de sunau.AU_read*), 2067
getnchannels() (*método de wave.Wave_read*), 1388
getnframes() (*método de aifc.aifc*), 1986
getnframes() (*método de sunau.AU_read*), 2067
getnframes() (*método de wave.Wave_read*), 1388
getnode, 1331
getnode() (*en el módulo uuid*), 1331
getopt
    módulo, 711
getopt() (*en el módulo getopt*), 712
GetoptError, 712
getouterframes() (*en el módulo inspect*), 1840
getoutput() (*en el módulo subprocess*), 907
getpagesize() (*en el módulo resource*), 1983
getparams() (*método de aifc.aifc*), 1986
getparams() (*método de sunau.AU_read*), 2067
getparams() (*método de wave.Wave_read*), 1388
getparyx() (*método de curses.window*), 766
getpass
    módulo, 755
getpass() (*en el módulo getpass*), 755
GetPassWarning, 756
getpeercert() (*método de ssl.SSLSocket*), 1048
getpeername() (*método de socket.socket*), 1023
getpen() (*en el módulo turtle*), 1429
getpgid() (*en el módulo os*), 605
getpgrp() (*en el módulo os*), 605
getpid() (*en el módulo os*), 606
getpos() (*método de html.parser.HTMLParser*), 1193
getppid() (*en el módulo os*), 606
getpreferredencoding() (*en el módulo locale*), 1404
getpriority() (*en el módulo os*), 606
getprofile() (*en el módulo sys*), 1757
getprofile() (*en el módulo threading*), 823
GetProperty() (*método de msilib.SummaryInformation*), 2017
getProperty() (*método de xml.sax.xmlreader.XMLReader*), 1245
GetPropertyCount() (*método de msilib.SummaryInformation*), 2017
getprotobyname() (*en el módulo socket*), 1019
getproxies() (*en el módulo urllib.request*), 1272
getPublicId() (*método de xml.sax.xmlreader.InputSource*), 1246
getPublicId() (*método de xml.sax.xmlreader.Locator*), 1245
getpwall() (*en el módulo pwd*), 1973
getpwnam() (*en el módulo pwd*), 1973
getpwuid() (*en el módulo pwd*), 1972
getQNameByName() (*método de xml.sax.xmlreader.AttributesNS*), 1247
getQNames() (*método de xml.sax.xmlreader.AttributesNS*), 1247
getquota() (*método de imaplib.IMAP4*), 1319
getquotaroot() (*método de imaplib.IMAP4*), 1319
getrandbits() (*en el módulo random*), 362
getrandom() (*en el módulo os*), 655
getreader() (*en el módulo codecs*), 175
getrecursionlimit() (*en el módulo sys*), 1757
getrefcount() (*en el módulo sys*), 1757
getresgid() (*en el módulo os*), 606
getresponse() (*método de http.client.HTTPConnection*), 1304
getresuid() (*en el módulo os*), 606
getrlimit() (*en el módulo resource*), 1979
getroot() (*método de xml.etree.ElementTree.ElementTree*), 1213
getrusage() (*en el módulo resource*), 1982
getsample() (*en el módulo audioop*), 1995
getsampwidth() (*método de aifc.aifc*), 1986
getsampwidth() (*método de sunau.AU_read*), 2067
getsampwidth() (*método de wave.Wave_read*), 1388
getscreen() (*en el módulo turtle*), 1429
getservbyname() (*en el módulo socket*), 1019
getservbyport() (*en el módulo socket*), 1019
GetSetDescriptorType (*en el módulo types*), 283
getshapes() (*en el módulo turtle*), 1435
getsid() (*en el módulo os*), 608
getsignal() (*en el módulo signal*), 1081
getsitepackages() (*en el módulo site*), 1846
getsize() (*en el módulo os.path*), 433
getsize() (*método de chunk.Chunk*), 2006
getsizeof() (*en el módulo sys*), 1757
getsockname() (*método de socket.socket*), 1024
getsockopt() (*método de socket.socket*), 1024
getsource() (*en el módulo inspect*), 1832
getsourcefile() (*en el módulo inspect*), 1832
getsourcelines() (*en el módulo inspect*), 1832
getspall() (*en el módulo spwd*), 2065
getspnam() (*en el módulo spwd*), 2065
getstate() (*en el módulo random*), 361
getstate() (*método de codecs.IncrementalDecoder*), 181
getstate() (*método de codecs.IncrementalEncoder*), 180
getstatusoutput() (*en el módulo subprocess*), 906
getstr() (*método de curses.window*), 766
GetString() (*método de msilib.Record*), 2018

getSubject() (*método de logging.handlers.SMTPHandler*), 752
GetSummaryInformation() (*método de msilib.Database*), 2016
getswitchinterval() (*en el módulo sys*), 1757
getSystemId() (*método de xml.sax.xmlreader.InputSource*), 1246
getSystemId() (*método de xml.sax.xmlreader.Locator*), 1245
getsyx() (*en el módulo curses*), 758
gettarinfo() (*método de tarfile.TarFile*), 545
gettempdir() (*en el módulo tempfile*), 448
gettempdirb() (*en el módulo tempfile*), 449
gettempprefix() (*en el módulo tempfile*), 449
gettempprefixb() (*en el módulo tempfile*), 449
getTestCaseNames() (*método de unittest.TestLoader*), 1593
gettext
    módulo, 1391
gettext() (*en el módulo gettext*), 1392
gettext() (*en el módulo locale*), 1407
gettext() (*método de gettext.GNUTranslations*), 1396
gettext() (*método de gettext.NullTranslations*), 1394
gettimeout() (*método de socket.socket*), 1024
gettrace() (*en el módulo sys*), 1757
gettrace() (*en el módulo threading*), 823
getturtle() (*en el módulo turtle*), 1429
getType() (*método de xml.sax.xmlreader.Attributes*), 1246
getuid() (*en el módulo os*), 606
geturl() (*método de http.client.HTTPResponse*), 1306
geturl() (*método de urllib.parse.urllib.parse.SplitResult*), 1294
geturl() (*método de urllib.response.addinfourl*), 1288
getuser() (*en el módulo getpass*), 756
getuserbase() (*en el módulo site*), 1846
getusersitepackages() (*en el módulo site*), 1846
getvalue() (*método de io.BytesIO*), 665
getvalue() (*método de io.StringIO*), 668
getValue() (*método de xml.sax.xmlreader.Attributes*), 1246
getValueByQName() (*método de xml.sax.xmlreader.AttributesNS*), 1247
getwch() (*en el módulo msvcrt*), 1958
getwche() (*en el módulo msvcrt*), 1958
getweakrefcount() (*en el módulo weakref*), 274
getweakrefs() (*en el módulo weakref*), 274
getwelcome() (*método de ftplib.FTP*), 1310
getwelcome() (*método de nntplib.NNTP*), 2024
getwelcome() (*método de poplib.POP3*), 1314
getwin() (*en el módulo curses*), 758
getwindowsversion() (*en el módulo sys*), 1758
getwriter() (*en el módulo codecs*), 175
getxattr() (*en el módulo os*), 640
getyx() (*método de curses.window*), 766
gid (*atributo de tarfile.TarInfo*), 546
GIL, **2085**
glob
    módulo, 451, 452
glob() (*en el módulo glob*), 451
glob() (*método de msilib.Directory*), 2019
glob() (*método de pathlib.Path*), 425
Global (*clase en ast*), 1915
globals()
    función incorporada, 14
globs (*atributo de doctest.DocTest*), 1563
gmtime() (*en el módulo time*), 672
gname (*atributo de tarfile.TarInfo*), 547
GNOME, 1397
GNU_FORMAT (*en el módulo tarfile*), 541
gnu_getopt() (*en el módulo getopt*), 712
GNUTranslations (*clase en gettext*), 1396
go() (*método de tkinter.filedialog.FileDialog*), 1471
got (*atributo de doctest.DocTestFailure*), 1570
goto() (*en el módulo turtle*), 1415
Graphical User Interface, 1455
graphlib
    módulo, 313
GREATER (*en el módulo token*), 1925
GREATEREQUAL (*en el módulo token*), 1925
Greenwich Mean Time, 670
GRND_NONBLOCK (*en el módulo os*), 656
GRND_RANDOM (*en el módulo os*), 656
Group (*clase en email.headerregistry*), 1122
group() (*método de nntplib.NNTP*), 2026
group() (*método de pathlib.Path*), 425
group() (*método de re.Match*), 136
groupby() (*en el módulo itertools*), 386
groupdict() (*método de re.Match*), 138
groupindex (*atributo de re.Pattern*), 136
groups (*atributo de email.headerregistry.AddressHeader*), 1119
groups (*atributo de re.Pattern*), 136
groups() (*método de re.Match*), 137
grp
    módulo, 1973
Gt (*clase en ast*), 1897
gt() (*en el módulo operator*), 406
GtE (*clase en ast*), 1897
guess_all_extensions() (*en el módulo mimetypes*), 1179
guess_all_extensions() (*método de mimetypes.MimeTypes*), 1181
guess_extension() (*en el módulo mimetypes*), 1179
guess_extension() (*método de mimetypes.MimeTypes*), 1181
guess_scheme() (*en el módulo wsgiref.util*), 1260
guess_type() (*en el módulo mimetypes*), 1179
guess_type() (*método de mimetypes.MimeTypes*), 1181
GUI, 1455
gzip
    módulo, 517
GzipFile (*clase en gzip*), 517
gzipopción de línea de comando

--best, 519
-d, 519
--decompress, 519
--fast, 519
file, 519
-h, 520
--help, 520

H

-h
astopción de línea de comando, 1921
gzipopción de línea de comando, 520
json.toolopción de línea de comando, 1161
timeitopción de línea de comando, 1714
tokenizeopción de línea de comando, 1929
zipappopción de línea de comando, 1743
halfdelay() (*en el módulo curses*), 759
Handle (*clase en asyncio*), 970
handle() (*método de http.server.BaseHTTPRequestHandler*), 1342
handle() (*método de logging.Handler*), 719
handle() (*método de logging.handlers.QueueListener*), 755
handle() (*método de logging.Logger*), 718
handle() (*método de logging.NullHandler*), 743
handle() (*método de socketserver.BaseRequestHandler*), 1337
handle() (*método de wsgiref.simple_server.WSGIRequestHandler*), 1264
handle_accept() (*método de asyncore.dispatcher*), 1991
handle_accepted() (*método de asyncore.dispatcher*), 1991
handle_charref() (*método de html.parser.HTMLParser*), 1193
handle_close() (*método de asyncore.dispatcher*), 1991
handle_comment() (*método de html.parser.HTMLParser*), 1194
handle_connect() (*método de asyncore.dispatcher*), 1991
handle_data() (*método de html.parser.HTMLParser*), 1193
handle_decl() (*método de html.parser.HTMLParser*), 1194
handle_defect() (*método de email.policy.Policy*), 1111
handle_endtag() (*método de html.parser.HTMLParser*), 1193
handle_entityref() (*método de html.parser.HTMLParser*), 1193
handle_error() (*método de asyncore.dispatcher*), 1991
handle_error() (*método de socketserver.BaseServer*), 1336
handle_expect_100() (*método de http.server.BaseHTTPRequestHandler*), 1343
handle_expt() (*método de asyncore.dispatcher*), 1991
handle_one_request() (*método de http.server.BaseHTTPRequestHandler*), 1343
handle_pi() (*método de html.parser.HTMLParser*), 1194
handle_read() (*método de asyncore.dispatcher*), 1991
handle_request() (*método de socketserver.BaseServer*), 1335
handle_request() (*método de xmlrpc.server.CGIXMLRPCRequestHandler*), 1371
handle_startendtag() (*método de html.parser.HTMLParser*), 1193
handle_starttag() (*método de html.parser.HTMLParser*), 1193
handle_timeout() (*método de socketserver.BaseServer*), 1336
handle_write() (*método de asyncore.dispatcher*), 1991
handleError() (*método de logging.Handler*), 719
handleError() (*método de logging.handlers.SocketHandler*), 748
Handler (*clase en logging*), 718
handler() (*en el módulo cgitb*), 2005
hardlink_to() (*método de pathlib.Path*), 429
--hardlink-dupes
compileallopción de línea de comando, 1937
harmonic_mean() (*en el módulo statistics*), 371
HAS_ALPN (*en el módulo ssl*), 1044
has_children() (*método de symtable.SymbolTable*), 1922
has_colors() (*en el módulo curses*), 758
has_dualstack_ipv6() (*en el módulo socket*), 1017
HAS_ECDH (*en el módulo ssl*), 1044
has_extended_color_support() (*en el módulo curses*), 758
has_extn() (*método de smtplib.SMTP*), 1326
has_header() (*método de csv.Sniffer*), 558
has_header() (*método de urllib.request.Request*), 1276
has_ic() (*en el módulo curses*), 759
has_il() (*en el módulo curses*), 759
has_ipv6 (*en el módulo socket*), 1015
has_key (*2to3 fixer*), 1663
has_key() (*en el módulo curses*), 759
has_location (*atributo de importlib.machinery.ModuleSpec*), 1878
HAS_NEVER_CHECK_COMMON_NAME (*en el módulo ssl*), 1044
has_nonstandard_attr() (*método de*

http.cookiejar.Cookie), 1358
HAS_NPN (*en el módulo ssl*), 1044
has_option() (*método de configparser.ConfigParser*), 576
has_option() (*método de optparse.OptionParser*), 2047
has_section() (*método de configparser.ConfigParser*), 576
HAS_SNI (*en el módulo ssl*), 1044
HAS_SSLv2 (*en el módulo ssl*), 1044
HAS_SSLv3 (*en el módulo ssl*), 1044
has_ticket (*atributo de ssl.SSLSession*), 1065
HAS_TLSv1 (*en el módulo ssl*), 1044
HAS_TLSv1_1 (*en el módulo ssl*), 1044
HAS_TLSv1_2 (*en el módulo ssl*), 1045
HAS_TLSv1_3 (*en el módulo ssl*), 1045
hasattr()
    función incorporada, 14
hasAttribute() (*método de xml.dom.Element*), 1224
hasAttributeNS() (*método de xml.dom.Element*), 1224
hasAttributes() (*método de xml.dom.Node*), 1221
hasChildNodes() (*método de xml.dom.Node*), 1221
hascompare (*en el módulo dis*), 1953
hasconst (*en el módulo dis*), 1953
hasFeature() (*método de xml.dom.DOMImplementation*), 1220
hasfree (*en el módulo dis*), 1953
hash
    función incorporada, 44
hash()
    función incorporada, 14
hash-based pyc, **2085**
hash_info (*en el módulo sys*), 1758
hashable, **2085**
Hashable (*clase en collections.abc*), 260
Hashable (*clase en typing*), 1538
hasHandlers() (*método de logging.Logger*), 718
hash.block_size (*en el módulo hashlib*), 587
hash.digest_size (*en el módulo hashlib*), 587
hashlib
    módulo, 585
hasjabs (*en el módulo dis*), 1953
hasjrel (*en el módulo dis*), 1953
haslocal (*en el módulo dis*), 1953
hasname (*en el módulo dis*), 1953
HAVE_ARGUMENT (*opcode*), 1953
HAVE_CONTEXTVAR (*en el módulo decimal*), 348
HAVE_DOCSTRINGS (*en el módulo test.support*), 1670
HAVE_THREADS (*en el módulo decimal*), 348
HCI_DATA_DIR (*en el módulo socket*), 1015
HCI_FILTER (*en el módulo socket*), 1015
HCI_TIME_STAMP (*en el módulo socket*), 1015
head() (*método de nntplib.NNTP*), 2027
Header (*clase en email.header*), 1143
header_encode() (*método de email.charset.Charset*), 1146
header_encode_lines() (*método de email.charset.Charset*), 1146
header_encoding (*atributo de email.charset.Charset*), 1145
header_factory (*atributo de email.policy.EmailPolicy*), 1113
header_fetch_parse() (*método de email.policy.Compat32*), 1115
header_fetch_parse() (*método de email.policy.EmailPolicy*), 1114
header_fetch_parse() (*método de email.policy.Policy*), 1112
header_items() (*método de urllib.request.Request*), 1276
header_max_count() (*método de email.policy.EmailPolicy*), 1114
header_max_count() (*método de email.policy.Policy*), 1112
header_offset (*atributo de zipfile.ZipInfo*), 537
header_source_parse() (*método de email.policy.Compat32*), 1115
header_source_parse() (*método de email.policy.EmailPolicy*), 1114
header_source_parse() (*método de email.policy.Policy*), 1112
header_store_parse() (*método de email.policy.Compat32*), 1115
header_store_parse() (*método de email.policy.EmailPolicy*), 1114
header_store_parse() (*método de email.policy.Policy*), 1112
HeaderError, 541
HeaderParseError, 1116
HeaderParser (*clase en email.parser*), 1105
HeaderRegistry (*clase en email.headerregistry*), 1120
headers
    MIME, 1179, 1997
headers (*atributo de http.client.HTTPResponse*), 1306
headers (*atributo de http.server.BaseHTTPRequestHandler*), 1342
headers (*atributo de urllib.error.HTTPError*), 1297
headers (*atributo de urllib.response.addinfourl*), 1288
headers (*atributo de xmlrpc.client.ProtocolError*), 1364
Headers (*clase en wsgiref.headers*), 1262
heading() (*en el módulo turtle*), 1419
heading() (*método de tkinter.ttk.Treeview*), 1487
heapify() (*en el módulo heapq*), 263
heapmin() (*en el módulo msvcrt*), 1959
heappop() (*en el módulo heapq*), 263
heappush() (*en el módulo heapq*), 263
heappushpop() (*en el módulo heapq*), 263
heapq
    módulo, 262
heapreplace() (*en el módulo heapq*), 263
helo() (*método de smtplib.SMTP*), 1325
help

    online, 1544
--help
    astopción de línea de comando, 1921
    gzipopción de línea de comando, 520
    json.toolopción de línea de comando, 1161
    timeitopción de línea de comando, 1714
    tokenizeopción de línea de comando, 1929
    traceopción de línea de comando, 1716
    zipappopción de línea de comando, 1743
help (*atributo de optparse.Option*), 2042
help (*pdb command*), 1699
help()
    función incorporada, 14
help() (*método de nntplib.NNTP*), 2026
herror, 1011
hex (*atributo de uuid.UUID*), 1331
hex()
    función incorporada, 14
hex() (*método de bytearray*), 61
hex() (*método de bytes*), 60
hex() (*método de float*), 39
hex() (*método de memoryview*), 76
hexadecimal
    literals, 35
hexbin() (*en el módulo binhex*), 1185
hexdigest() (*método de hashlib.hash*), 587
hexdigest() (*método de hashlib.shake*), 587
hexdigest() (*método de hmac.HMAC*), 596
hexdigits (*en el módulo string*), 113
hexlify() (*en el módulo binascii*), 1187
hexversion (*en el módulo sys*), 1759
hidden() (*método de curses.panel.Panel*), 778
hide() (*método de curses.panel.Panel*), 778
hide() (*método de tkinter.ttk.Notebook*), 1482
hide_cookie2 (*atributo de http.cookiejar.CookiePolicy*), 1355
hideturtle() (*en el módulo turtle*), 1424
HierarchyRequestErr, 1226
HIGH_PRIORITY_CLASS (*en el módulo subprocess*), 901
HIGHEST_PROTOCOL (*en el módulo pickle*), 467
hits (*atributo de bdb.Breakpoint*), 1690
HKEY_CLASSES_ROOT (*en el módulo winreg*), 1965
HKEY_CURRENT_CONFIG (*en el módulo winreg*), 1965
HKEY_CURRENT_USER (*en el módulo winreg*), 1965
HKEY_DYN_DATA (*en el módulo winreg*), 1965
HKEY_LOCAL_MACHINE (*en el módulo winreg*), 1965
HKEY_PERFORMANCE_DATA (*en el módulo winreg*), 1965
HKEY_USERS (*en el módulo winreg*), 1965
hline() (*método de curses.window*), 766
HList (*clase en tkinter.tix*), 1495
hls_to_rgb() (*en el módulo colorsys*), 1390
hmac
    módulo, 595
HOME, 433, 1457
home() (*en el módulo turtle*), 1416
home() (*método de clase de pathlib.Path*), 424
HOMEDRIVE, 433
HOMEPATH, 433
hook_compressed() (*en el módulo fileinput*), 439
hook_encoded() (*en el módulo fileinput*), 439
host (*atributo de urllib.request.Request*), 1275
hostmask (*atributo de ipaddress.IPv4Network*), 1378
hostmask (*atributo de ipaddress.IPv6Network*), 1381
hostname_checks_common_name (*atributo de ssl.SSLContext*), 1057
hosts (*atributo de netrc.netrc*), 581
hosts() (*método de ipaddress.IPv4Network*), 1379
hosts() (*método de ipaddress.IPv6Network*), 1381
hour (*atributo de datetime.datetime*), 206
hour (*atributo de datetime.time*), 214
HRESULT (*clase en ctypes*), 818
hStdError (*atributo de subprocess.STARTUPINFO*), 900
hStdInput (*atributo de subprocess.STARTUPINFO*), 900
hStdOutput (*atributo de subprocess.STARTUPINFO*), 900
hsv_to_rgb() (*en el módulo colorsys*), 1390
ht() (*en el módulo turtle*), 1424
html
    módulo, 1191
HTML, 1192, 1287
html() (*en el módulo cgitb*), 2005
html5 (*en el módulo html.entities*), 1196
HTMLCalendar (*clase en calendar*), 236
HtmlDiff (*clase en difflib*), 145
html.entities
    módulo, 1196
html.parser
    módulo, 1192
HTMLParser (*clase en html.parser*), 1192
htonl() (*en el módulo socket*), 1019
htons() (*en el módulo socket*), 1019
http
    módulo, 1299
HTTP
    http (*standard module*), 1299
    http.client (*standard module*), 1301
    protocol, 1287, 1299, 1301, 1341, 1997
HTTP (*en el módulo email.policy*), 1115
http_error_301() (*método de urllib.request.HTTPRedirectHandler*), 1279
http_error_302() (*método de urllib.request.HTTPRedirectHandler*), 1279
http_error_303() (*método de urllib.request.HTTPRedirectHandler*), 1279
http_error_307() (*método de urllib.request.HTTPRedirectHandler*), 1279

http_error_401() (*método de urllib.request.HTTPBasicAuthHandler*), 1281
http_error_401() (*método de urllib.request.HTTPDigestAuthHandler*), 1281
http_error_407() (*método de urllib.request.ProxyBasicAuthHandler*), 1281
http_error_407() (*método de urllib.request.ProxyDigestAuthHandler*), 1281
http_error_auth_reqed() (*método de urllib.request.AbstractBasicAuthHandler*), 1280
http_error_auth_reqed() (*método de urllib.request.AbstractDigestAuthHandler*), 1281
http_error_default() (*método de urllib.request.BaseHandler*), 1278
http_open() (*método de urllib.request.HTTPHandler*), 1281
HTTP_PORT (*en el módulo http.client*), 1303
http_proxy, 1271, 1284
http_response() (*método de urllib.request.HTTPErrorProcessor*), 1282
http_version (*atributo de wsgiref.handlers.BaseHandler*), 1268
HTTPBasicAuthHandler (*clase en urllib.request*), 1274
http.client
    módulo, 1301
HTTPConnection (*clase en http.client*), 1301
http.cookiejar
    módulo, 1350
HTTPCookieProcessor (*clase en urllib.request*), 1273
http.cookies
    módulo, 1347
httpd, 1341
HTTPDefaultErrorHandler (*clase en urllib.request*), 1273
HTTPDigestAuthHandler (*clase en urllib.request*), 1274
HTTPError, 1297
HTTPErrorProcessor (*clase en urllib.request*), 1275
HTTPException, 1303
HTTPHandler (*clase en logging.handlers*), 753
HTTPHandler (*clase en urllib.request*), 1274
HTTPPasswordMgr (*clase en urllib.request*), 1273
HTTPPasswordMgrWithDefaultRealm (*clase en urllib.request*), 1273
HTTPPasswordMgrWithPriorAuth (*clase en urllib.request*), 1273
HTTPRedirectHandler (*clase en urllib.request*), 1273
HTTPResponse (*clase en http.client*), 1302
https_open() (*método de urllib.request.HTTPSHandler*), 1281
HTTPS_PORT (*en el módulo http.client*), 1303
https_response() (*método de urllib.request.HTTPErrorProcessor*), 1282
HTTPSConnection (*clase en http.client*), 1302
http.server
    módulo, 1341
    security, 1346
HTTPServer (*clase en http.server*), 1341
HTTPSHandler (*clase en urllib.request*), 1274
HTTPStatus (*clase en http*), 1299
hypot() (*en el módulo math*), 325

I

I (*en el módulo re*), 130
-i <indent>
    astopción de línea de comando, 1921
-i list
    compileallopción de línea de comando, 1936
I/O control
    buffering, 20, 1024
    POSIX, 1974
    tty, 1974
    UNIX, 1977
iadd() (*en el módulo operator*), 411
iand() (*en el módulo operator*), 411
iconcat() (*en el módulo operator*), 411
id (*atributo de ssl.SSLSession*), 1065
id()
    función incorporada, 15
id() (*método de unittest.TestCase*), 1588
idcok() (*método de curses.window*), 766
ident (*atributo de select.kevent*), 1073
ident (*atributo de threading.Thread*), 825
identchars (*atributo de cmd.Cmd*), 1445
identify() (*método de tkinter.ttk.Notebook*), 1482
identify() (*método de tkinter.ttk.Treeview*), 1488
identify() (*método de tkinter.ttk.Widget*), 1478
identify_column() (*método de tkinter.ttk.Treeview*), 1488
identify_element() (*método de tkinter.ttk.Treeview*), 1488
identify_region() (*método de tkinter.ttk.Treeview*), 1488
identify_row() (*método de tkinter.ttk.Treeview*), 1488
idioms (*2to3 fixer*), 1663
IDLE, 1498, **2085**
IDLE_PRIORITY_CLASS (*en el módulo subprocess*), 901
idlelib
    módulo, 1510
IDLESTARTUP, 1505
idlok() (*método de curses.window*), 766
if
    sentencia, 33
If (*clase en ast*), 1904
if_indextoname() (*en el módulo socket*), 1022
if_nameindex() (*en el módulo socket*), 1021
if_nametoindex() (*en el módulo socket*), 1021
IfExp (*clase en ast*), 1898
ifloordiv() (*en el módulo operator*), 411

iglob() (*en el módulo glob*), 451
ignorableWhitespace() (*método de xml.sax.handler.ContentHandler*), 1240
ignore
    error handler's name, 177
ignore (*atributo de bdb.Breakpoint*), 1690
ignore (*pdb command*), 1700
ignore_errors() (*en el módulo codecs*), 178
IGNORE_EXCEPTION_DETAIL (*en el módulo doctest*), 1555
ignore_patterns() (*en el módulo shutil*), 456
ignore_warnings() (*en el módulo test.support.warnings_helper*), 1683
IGNORECASE (*en el módulo re*), 130
--ignore-dir=<dir>
    traceopción de línea de comando, 1718
--ignore-module=<mod>
    traceopción de línea de comando, 1718
ihave() (*método de nntplib.NNTP*), 2027
IISCGIHandler (*clase en wsgiref.handlers*), 1265
ilshift() (*en el módulo operator*), 411
imag (*atributo de numbers.Complex*), 317
imap() (*método de multiprocessing.pool.Pool*), 862
IMAP4
    protocol, 1316
IMAP4 (*clase en imaplib*), 1316
IMAP4_SSL
    protocol, 1316
IMAP4_SSL (*clase en imaplib*), 1317
IMAP4_stream
    protocol, 1316
IMAP4_stream (*clase en imaplib*), 1317
IMAP4.abort, 1317
IMAP4.error, 1317
IMAP4.readonly, 1317
imap_unordered() (*método de multiprocessing.pool.Pool*), 863
imaplib
    módulo, 1316
imatmul() (*en el módulo operator*), 411
imghdr
    módulo, 2008
immedok() (*método de curses.window*), 766
immutable
    sequence types, 44
imod() (*en el módulo operator*), 411
imp
    módulo, 28, 2009
ImpImporter (*clase en pkgutil*), 1856
impl_detail() (*en el módulo test.support*), 1673
implementation (*en el módulo sys*), 1759
ImpLoader (*clase en pkgutil*), 1856
import
    sentencia, 28, 1844, 2009
import (*2to3 fixer*), 1663
Import (*clase en ast*), 1904
import_fresh_module() (*en el módulo test.support.import_helper*), 1682
IMPORT_FROM (*opcode*), 1949
import_module() (*en el módulo importlib*), 1863
import_module() (*en el módulo test.support.import_helper*), 1682
IMPORT_NAME (*opcode*), 1949
IMPORT_STAR (*opcode*), 1947
importador, **2085**
importar, **2085**
ImportError, 103
ImportFrom (*clase en ast*), 1904
importlib
    módulo, 1862
importlib.abc
    módulo, 1865
importlib.machinery
    módulo, 1874
importlib.metadata
    módulo, 1884
importlib.resources
    módulo, 1872
importlib.util
    módulo, 1878
imports (*2to3 fixer*), 1663
imports2 (*2to3 fixer*), 1663
ImportWarning, 109
ImproperConnectionState, 1303
imul() (*en el módulo operator*), 411
in
    operador, 34, 42
In (*clase en ast*), 1897
in_dll() (*método de ctypes._CData*), 815
in_table_a1() (*en el módulo stringprep*), 161
in_table_b1() (*en el módulo stringprep*), 161
in_table_c3() (*en el módulo stringprep*), 161
in_table_c4() (*en el módulo stringprep*), 161
in_table_c5() (*en el módulo stringprep*), 161
in_table_c6() (*en el módulo stringprep*), 161
in_table_c7() (*en el módulo stringprep*), 161
in_table_c8() (*en el módulo stringprep*), 161
in_table_c9() (*en el módulo stringprep*), 161
in_table_c11() (*en el módulo stringprep*), 161
in_table_c11_c12() (*en el módulo stringprep*), 161
in_table_c12() (*en el módulo stringprep*), 161
in_table_c21() (*en el módulo stringprep*), 161
in_table_c21_c22() (*en el módulo stringprep*), 161
in_table_c22() (*en el módulo stringprep*), 161
in_table_d1() (*en el módulo stringprep*), 161
in_table_d2() (*en el módulo stringprep*), 161
in_transaction (*atributo de sqlite3.Connection*), 500
inch() (*método de curses.window*), 767
--include-attributes
    astopción de línea de comando, 1921

inclusive (*atributo de tracemalloc.DomainFilter*), 1725
inclusive (*atributo de tracemalloc.Filter*), 1726
Incomplete, 1188
IncompleteRead, 1303
IncompleteReadError, 952
increment_lineno() (*en el módulo ast*), 1919
incrementaldecoder (*atributo de codecs.CodecInfo*), 174
IncrementalDecoder (*clase en codecs*), 180
incrementalencoder (*atributo de codecs.CodecInfo*), 174
IncrementalEncoder (*clase en codecs*), 180
IncrementalNewlineDecoder (*clase en io*), 669
IncrementalParser (*clase en xml.sax.xmlreader*), 1243
--indent
  json.toolopción de línea de comando, 1161
indent (*atributo de doctest.Example*), 1564
INDENT (*en el módulo token*), 1924
--indent <indent>
  astopción de línea de comando, 1921
indent() (*en el módulo textwrap*), 156
indent() (*en el módulo xml.etree.ElementTree*), 1207
IndentationError, 106
--indentlevel=<num>
  pickletoolsopción de línea de comando, 1954
index() (*en el módulo operator*), 407
index() (*método de array.array*), 271
index() (*método de bytearray*), 63
index() (*método de bytes*), 63
index() (*método de collections.deque*), 245
index() (*método de multiprocessing.shared_memory.ShareableList*), 880
index() (*método de str*), 51
index() (*método de tkinter.ttk.Notebook*), 1482
index() (*método de tkinter.ttk.Treeview*), 1488
index() (*sequence method*), 42
IndexError, 103
indexOf() (*en el módulo operator*), 408
IndexSizeErr, 1226
indicador de tipo, **2092**
inet_aton() (*en el módulo socket*), 1019
inet_ntoa() (*en el módulo socket*), 1020
inet_ntop() (*en el módulo socket*), 1020
inet_pton() (*en el módulo socket*), 1020
Inexact (*clase en decimal*), 349
inf (*en el módulo cmath*), 330
inf (*en el módulo math*), 327
infile
  json.toolopción de línea de comando, 1161
infile (*atributo de shlex.shlex*), 1451
Infinity, 13
infj (*en el módulo cmath*), 330
--info
  zipappopción de línea de comando, 1743
info() (*en el módulo logging*), 726
info() (*método de dis.Bytecode*), 1941
info() (*método de gettext.NullTranslations*), 1395
info() (*método de http.client.HTTPResponse*), 1306
info() (*método de logging.Logger*), 717
info() (*método de urllib.response.addinfourl*), 1288
infolist() (*método de zipfile.ZipFile*), 531
.ini
  file, 562
ini file, 562
init() (*en el módulo mimetypes*), 1179
init_color() (*en el módulo curses*), 759
init_database() (*en el módulo msilib*), 2016
init_pair() (*en el módulo curses*), 759
inited (*en el módulo mimetypes*), 1180
initgroups() (*en el módulo os*), 606
initial_indent (*atributo de textwrap.TextWrapper*), 157
initscr() (*en el módulo curses*), 759
inmutable, **2085**
inode() (*método de os.DirEntry*), 629
INPLACE_ADD (*opcode*), 1945
INPLACE_AND (*opcode*), 1945
INPLACE_FLOOR_DIVIDE (*opcode*), 1945
INPLACE_LSHIFT (*opcode*), 1945
INPLACE_MATRIX_MULTIPLY (*opcode*), 1945
INPLACE_MODULO (*opcode*), 1945
INPLACE_MULTIPLY (*opcode*), 1945
INPLACE_OR (*opcode*), 1945
INPLACE_POWER (*opcode*), 1945
INPLACE_RSHIFT (*opcode*), 1945
INPLACE_SUBTRACT (*opcode*), 1945
INPLACE_TRUE_DIVIDE (*opcode*), 1945
INPLACE_XOR (*opcode*), 1945
input (*2to3 fixer*), 1663
input()
  función incorporada, 15
input() (*en el módulo fileinput*), 437
input_charset (*atributo de email.charset.Charset*), 1145
input_codec (*atributo de email.charset.Charset*), 1146
InputOnly (*clase en tkinter.tix*), 1496
InputSource (*clase en xml.sax.xmlreader*), 1243
insch() (*método de curses.window*), 767
insdelln() (*método de curses.window*), 767
insert() (*método de array.array*), 272
insert() (*método de collections.deque*), 245
insert() (*método de tkinter.ttk.Notebook*), 1482
insert() (*método de tkinter.ttk.Treeview*), 1488
insert() (*método de xml.etree.ElementTree.Element*), 1212
insert() (*sequence method*), 44
insert_text() (*en el módulo readline*), 162
insertBefore() (*método de xml.dom.Node*), 1221
insertln() (*método de curses.window*), 767

insnstr() (*método de curses.window*), 767
insort() (*en el módulo bisect*), 267
insort_left() (*en el módulo bisect*), 267
insort_right() (*en el módulo bisect*), 267
inspect
    módulo, 1827
InspectLoader (*clase en importlib.abc*), 1869
inspectopción de línea de comando
    --details, 1844
insstr() (*método de curses.window*), 767
install() (*en el módulo gettext*), 1394
install() (*método de gettext.NullTranslations*), 1395
install_opener() (*en el módulo urllib.request*), 1271
install_scripts() (*método de venv.EnvBuilder*), 1738
installHandler() (*en el módulo unittest*), 1601
instate() (*método de tkinter.ttk.Widget*), 1478
instr() (*método de curses.window*), 767
instream (*atributo de shlex.shlex*), 1451
Instruction (*clase en dis*), 1943
Instruction.arg (*en el módulo dis*), 1943
Instruction.argrepr (*en el módulo dis*), 1943
Instruction.argval (*en el módulo dis*), 1943
Instruction.is_jump_target (*en el módulo dis*), 1943
Instruction.offset (*en el módulo dis*), 1943
Instruction.opcode (*en el módulo dis*), 1943
Instruction.opname (*en el módulo dis*), 1943
Instruction.starts_line (*en el módulo dis*), 1943
int
    función incorporada, 35
int (*atributo de uuid.UUID*), 1331
int (*clase incorporada*), 15
Int2AP() (*en el módulo imaplib*), 1317
int_info (*en el módulo sys*), 1760
integer
    literals, 35
    objeto, 35
    types, operations on, 36
Integral (*clase en numbers*), 318
Integrated Development Environment, 1498
IntegrityError, 504
Intel/DVI ADPCM, 1994
IntEnum (*clase en enum*), 294
interact (*pdb command*), 1702
interact() (*en el módulo code*), 1849
interact() (*método de code.InteractiveConsole*), 1851
interact() (*método de telnetlib.Telnet*), 2070
InteractiveConsole (*clase en code*), 1849
InteractiveInterpreter (*clase en code*), 1849
interactivo, **2085**
InterfaceError, 504
intern (*2to3 fixer*), 1663
intern() (*en el módulo sys*), 1760
internal_attr (*atributo de zipfile.ZipInfo*), 537
Internaldate2tuple() (*en el módulo imaplib*), 1317
InternalError, 505
internalSubset (*atributo de xml.dom.DocumentType*), 1222
Internet, 1257
INTERNET_TIMEOUT (*en el módulo test.support*), 1669
interpolation
    bytearray (%), 72
    bytes (%), 72
interpolation, string (%), 57
InterpolationDepthError, 580
InterpolationError, 580
InterpolationMissingOptionError, 580
InterpolationSyntaxError, 580
interpretado, **2085**
interpreter prompts, 1763
interpreter_requires_environment() (*en el módulo test.support.script_helper*), 1677
interrupt() (*método de sqlite3.Connection*), 498
interrupt_main() (*en el módulo _thread*), 916
InterruptedError, 108
intersection() (*método de frozenset*), 82
intersection_update() (*método de frozenset*), 83
IntFlag (*clase en enum*), 294
intro (*atributo de cmd.Cmd*), 1445
InuseAttributeErr, 1226
inv() (*en el módulo operator*), 407
inv_cdf() (*método de statistics.NormalDist*), 378
InvalidAccessErr, 1226
invalidate_caches() (*en el módulo importlib*), 1864
invalidate_caches() (*método de clase de importlib.machinery.PathFinder*), 1875
invalidate_caches() (*método de importlib.abc.MetaPathFinder*), 1866
invalidate_caches() (*método de importlib.abc.PathEntryFinder*), 1866
invalidate_caches() (*método de importlib.machinery.FileFinder*), 1876
invalidate_caches() (*método de zipimport.zipimporter*), 1855
--invalidation-mode [timestamp|checked-hash|unchecked-hash]
    compileallopción de línea de comando, 1937
InvalidCharacterErr, 1226
InvalidModificationErr, 1226
InvalidOperation (*clase en decimal*), 349
InvalidStateErr, 1226
InvalidStateError, 888, 952
InvalidTZPathWarning, 234
InvalidURL, 1303
Invert (*clase en ast*), 1896
invert() (*en el módulo operator*), 407

io
    módulo, 656
IO (*clase en typing*), 1536
IO_REPARSE_TAG_APPEXECLINK (*en el módulo stat*), 444
IO_REPARSE_TAG_MOUNT_POINT (*en el módulo stat*), 444
IO_REPARSE_TAG_SYMLINK (*en el módulo stat*), 444
IOBase (*clase en io*), 660
ioctl() (*en el módulo fcntl*), 1977
ioctl() (*método de socket.socket*), 1024
IOCTL_VM_SOCKETS_GET_LOCAL_CID (*en el módulo socket*), 1014
IOError, 107
ior() (*en el módulo operator*), 411
io.StringIO
    objeto, 48
ip (*atributo de ipaddress.IPv4Interface*), 1383
ip (*atributo de ipaddress.IPv6Interface*), 1384
ip_address() (*en el módulo ipaddress*), 1372
ip_interface() (*en el módulo ipaddress*), 1373
ip_network() (*en el módulo ipaddress*), 1372
ipaddress
    módulo, 1372
ipow() (*en el módulo operator*), 412
ipv4_mapped (*atributo de ipaddress.IPv6Address*), 1376
IPv4Address (*clase en ipaddress*), 1373
IPv4Interface (*clase en ipaddress*), 1383
IPv4Network (*clase en ipaddress*), 1378
IPV6_ENABLED (*en el módulo test.support.socket_helper*), 1677
IPv6Address (*clase en ipaddress*), 1375
IPv6Interface (*clase en ipaddress*), 1383
IPv6Network (*clase en ipaddress*), 1380
irshift() (*en el módulo operator*), 412
is
    operador, 34
Is (*clase en ast*), 1897
is not
    operador, 34
is_() (*en el módulo operator*), 406
is_absolute() (*método de pathlib.PurePath*), 420
is_active() (*método de asyncio.AbstractChildWatcher*), 994
is_active() (*método de graphlib.TopologicalSorter*), 314
is_alive() (*método de multiprocessing.Process*), 842
is_alive() (*método de threading.Thread*), 826
is_android (*en el módulo test.support*), 1668
is_annotated() (*método de symtable.Symbol*), 1923
is_assigned() (*método de symtable.Symbol*), 1923
is_async (*atributo de pyclbr.Function*), 1933
is_attachment() (*método de email.message.EmailMessage*), 1099
is_authenticated() (*método de urllib.request.HTTPPasswordMgrWithPriorAuth*), 1280
is_block_device() (*método de pathlib.Path*), 426
is_blocked() (*método de http.cookiejar.DefaultCookiePolicy*), 1356
is_canonical() (*método de decimal.Context*), 345
is_canonical() (*método de decimal.Decimal*), 339
is_char_device() (*método de pathlib.Path*), 426
IS_CHARACTER_JUNK() (*en el módulo difflib*), 148
is_check_supported() (*en el módulo lzma*), 527
is_closed() (*método de asyncio.loop*), 955
is_closing() (*método de asyncio.BaseTransport*), 980
is_closing() (*método de asyncio.StreamWriter*), 938
is_dataclass() (*en el módulo dataclasses*), 1791
is_declared_global() (*método de symtable.Symbol*), 1923
is_dir() (*método de importlib.abc.Traversable*), 1871
is_dir() (*método de os.DirEntry*), 629
is_dir() (*método de pathlib.Path*), 425
is_dir() (*método de zipfile.Path*), 534
is_dir() (*método de zipfile.ZipInfo*), 536
is_enabled() (*en el módulo faulthandler*), 1695
is_expired() (*método de http.cookiejar.Cookie*), 1358
is_fifo() (*método de pathlib.Path*), 426
is_file() (*método de importlib.abc.Traversable*), 1871
is_file() (*método de os.DirEntry*), 629
is_file() (*método de pathlib.Path*), 425
is_file() (*método de zipfile.Path*), 534
is_finalized() (*en el módulo gc*), 1826
is_finalizing() (*en el módulo sys*), 1760
is_finite() (*método de decimal.Context*), 345
is_finite() (*método de decimal.Decimal*), 339
is_free() (*método de symtable.Symbol*), 1923
is_global (*atributo de ipaddress.IPv4Address*), 1374
is_global (*atributo de ipaddress.IPv6Address*), 1375
is_global() (*método de symtable.Symbol*), 1923
is_hop_by_hop() (*en el módulo wsgiref.util*), 1261
is_imported() (*método de symtable.Symbol*), 1923
is_infinite() (*método de decimal.Context*), 346
is_infinite() (*método de decimal.Decimal*), 339
is_integer() (*método de float*), 38
is_jython (*en el módulo test.support*), 1668
IS_LINE_JUNK() (*en el módulo difflib*), 148
is_linetouched() (*método de curses.window*), 767
is_link_local (*atributo de ipaddress.IPv4Address*), 1374
is_link_local (*atributo de ipaddress.IPv4Network*), 1378
is_link_local (*atributo de ipaddress.IPv6Address*), 1376
is_link_local (*atributo de ipaddress.IPv6Network*), 1381
is_local() (*método de symtable.Symbol*), 1923

is_loopback (*atributo de ipaddress.IPv4Address*), 1374
is_loopback (*atributo de ipaddress.IPv4Network*), 1378
is_loopback (*atributo de ipaddress.IPv6Address*), 1376
is_loopback (*atributo de ipaddress.IPv6Network*), 1381
is_mount() (*método de pathlib.Path*), 425
is_multicast (*atributo de ipaddress.IPv4Address*), 1374
is_multicast (*atributo de ipaddress.IPv4Network*), 1378
is_multicast (*atributo de ipaddress.IPv6Address*), 1375
is_multicast (*atributo de ipaddress.IPv6Network*), 1381
is_multipart() (*método de email.message.EmailMessage*), 1096
is_multipart() (*método de email.message.Message*), 1133
is_namespace() (*método de symtable.Symbol*), 1923
is_nan() (*método de decimal.Context*), 346
is_nan() (*método de decimal.Decimal*), 339
is_nested() (*método de symtable.SymbolTable*), 1922
is_nonlocal() (*método de symtable.Symbol*), 1923
is_normal() (*método de decimal.Context*), 346
is_normal() (*método de decimal.Decimal*), 339
is_normalized() (*en el módulo unicodedata*), 160
is_not() (*en el módulo operator*), 406
is_not_allowed() (*método de http.cookiejar.DefaultCookiePolicy*), 1356
IS_OP (*opcode*), 1949
is_optimized() (*método de symtable.SymbolTable*), 1922
is_package() (*método de importlib.abc.InspectLoader*), 1869
is_package() (*método de importlib.abc.SourceLoader*), 1871
is_package() (*método de importlib.machinery.ExtensionFileLoader*), 1877
is_package() (*método de importlib.machinery.SourceFileLoader*), 1876
is_package() (*método de importlib.machinery.SourcelessFileLoader*), 1877
is_package() (*método de zipimport.zipimporter*), 1854
is_parameter() (*método de symtable.Symbol*), 1923
is_private (*atributo de ipaddress.IPv4Address*), 1374
is_private (*atributo de ipaddress.IPv4Network*), 1378
is_private (*atributo de ipaddress.IPv6Address*), 1375
is_private (*atributo de ipaddress.IPv6Network*), 1381
is_python_build() (*en el módulo sysconfig*), 1772
is_qnan() (*método de decimal.Context*), 346
is_qnan() (*método de decimal.Decimal*), 339
is_reading() (*método de asyncio.ReadTransport*), 981
is_referenced() (*método de symtable.Symbol*), 1923
is_relative_to() (*método de pathlib.PurePath*), 420
is_reserved (*atributo de ipaddress.IPv4Address*), 1374
is_reserved (*atributo de ipaddress.IPv4Network*), 1378
is_reserved (*atributo de ipaddress.IPv6Address*), 1376
is_reserved (*atributo de ipaddress.IPv6Network*), 1381
is_reserved() (*método de pathlib.PurePath*), 421
is_resource() (*en el módulo importlib.resources*), 1873
is_resource() (*método de importlib.abc.ResourceReader*), 1868
is_resource_enabled() (*en el módulo test.support*), 1670
is_running() (*método de asyncio.loop*), 955
is_safe (*atributo de uuid.UUID*), 1331
is_serving() (*método de asyncio.Server*), 972
is_set() (*método de asyncio.Event*), 943
is_set() (*método de threading.Event*), 831
is_signed() (*método de decimal.Context*), 346
is_signed() (*método de decimal.Decimal*), 339
is_site_local (*atributo de ipaddress.IPv6Address*), 1376
is_site_local (*atributo de ipaddress.IPv6Network*), 1382
is_skipped_line() (*método de bdb.Bdb*), 1692
is_snan() (*método de decimal.Context*), 346
is_snan() (*método de decimal.Decimal*), 339
is_socket() (*método de pathlib.Path*), 426
is_subnormal() (*método de decimal.Context*), 346
is_subnormal() (*método de decimal.Decimal*), 339
is_symlink() (*método de os.DirEntry*), 629
is_symlink() (*método de pathlib.Path*), 426
is_tarfile() (*en el módulo tarfile*), 541
is_term_resized() (*en el módulo curses*), 759
is_tracing() (*en el módulo tracemalloc*), 1724
is_tracked() (*en el módulo gc*), 1825
is_typeddict() (*en el módulo typing*), 1543
is_unspecified (*atributo de ipaddress.IPv4Address*), 1374
is_unspecified (*atributo de ipaddress.IPv4Network*), 1378
is_unspecified (*atributo de ipaddress.IPv6Address*), 1376

is_unspecified (*atributo de ipaddress.IPv6Network*), 1381
is_wintouched() (*método de curses.window*), 767
is_zero() (*método de decimal.Context*), 346
is_zero() (*método de decimal.Decimal*), 339
is_zipfile() (*en el módulo zipfile*), 530
isabs() (*en el módulo os.path*), 433
isabstract() (*en el módulo inspect*), 1831
IsADirectoryError, 108
isalnum() (*en el módulo curses.ascii*), 776
isalnum() (*método de bytearray*), 68
isalnum() (*método de bytes*), 68
isalnum() (*método de str*), 51
isalpha() (*en el módulo curses.ascii*), 776
isalpha() (*método de bytearray*), 68
isalpha() (*método de bytes*), 68
isalpha() (*método de str*), 51
isascii() (*en el módulo curses.ascii*), 776
isascii() (*método de bytearray*), 68
isascii() (*método de bytes*), 68
isascii() (*método de str*), 51
isasyncgen() (*en el módulo inspect*), 1830
isasyncgenfunction() (*en el módulo inspect*), 1830
isatty() (*en el módulo os*), 611
isatty() (*método de chunk.Chunk*), 2006
isatty() (*método de io.IOBase*), 661
isawaitable() (*en el módulo inspect*), 1830
isblank() (*en el módulo curses.ascii*), 776
isblk() (*método de tarfile.TarInfo*), 547
isbuiltin() (*en el módulo inspect*), 1831
ischr() (*método de tarfile.TarInfo*), 547
isclass() (*en el módulo inspect*), 1829
isclose() (*en el módulo cmath*), 330
isclose() (*en el módulo math*), 322
iscntrl() (*en el módulo curses.ascii*), 776
iscode() (*en el módulo inspect*), 1830
iscoroutine() (*en el módulo asyncio*), 934
iscoroutine() (*en el módulo inspect*), 1830
iscoroutinefunction() (*en el módulo asyncio*), 934
iscoroutinefunction() (*en el módulo inspect*), 1830
isctrl() (*en el módulo curses.ascii*), 777
isDaemon() (*método de threading.Thread*), 826
isdatadescriptor() (*en el módulo inspect*), 1831
isdecimal() (*método de str*), 51
isdev() (*método de tarfile.TarInfo*), 547
isdigit() (*en el módulo curses.ascii*), 776
isdigit() (*método de bytearray*), 69
isdigit() (*método de bytes*), 69
isdigit() (*método de str*), 51
isdir() (*en el módulo os.path*), 434
isdir() (*método de tarfile.TarInfo*), 547
isdisjoint() (*método de frozenset*), 81
isdown() (*en el módulo turtle*), 1421
iselement() (*en el módulo xml.etree.ElementTree*), 1207
isenabled() (*en el módulo gc*), 1824
isEnabledFor() (*método de logging.Logger*), 715
isendwin() (*en el módulo curses*), 759
ISEOF() (*en el módulo token*), 1924
isfifo() (*método de tarfile.TarInfo*), 547
isfile() (*en el módulo os.path*), 434
isfile() (*método de tarfile.TarInfo*), 547
isfinite() (*en el módulo cmath*), 330
isfinite() (*en el módulo math*), 322
isfirstline() (*en el módulo fileinput*), 438
isframe() (*en el módulo inspect*), 1830
isfunction() (*en el módulo inspect*), 1829
isfuture() (*en el módulo asyncio*), 975
isgenerator() (*en el módulo inspect*), 1830
isgeneratorfunction() (*en el módulo inspect*), 1830
isgetsetdescriptor() (*en el módulo inspect*), 1831
isgraph() (*en el módulo curses.ascii*), 776
isidentifier() (*método de str*), 51
isinf() (*en el módulo cmath*), 330
isinf() (*en el módulo math*), 322
isinstance (*2to3 fixer*), 1663
isinstance()
    función incorporada, 16
iskeyword() (*en el módulo keyword*), 1927
isleap() (*en el módulo calendar*), 237
islice() (*en el módulo itertools*), 387
islink() (*en el módulo os.path*), 434
islnk() (*método de tarfile.TarInfo*), 547
islower() (*en el módulo curses.ascii*), 776
islower() (*método de bytearray*), 69
islower() (*método de bytes*), 69
islower() (*método de str*), 52
ismemberdescriptor() (*en el módulo inspect*), 1831
ismeta() (*en el módulo curses.ascii*), 777
ismethod() (*en el módulo inspect*), 1829
ismethoddescriptor() (*en el módulo inspect*), 1831
ismodule() (*en el módulo inspect*), 1829
ismount() (*en el módulo os.path*), 434
isnan() (*en el módulo cmath*), 330
isnan() (*en el módulo math*), 322
ISNONTERMINAL() (*en el módulo token*), 1924
IsNot (*clase en ast*), 1897
isnumeric() (*método de str*), 52
isocalendar() (*método de datetime.date*), 201
isocalendar() (*método de datetime.datetime*), 210
isoformat() (*método de datetime.date*), 201
isoformat() (*método de datetime.datetime*), 210
isoformat() (*método de datetime.time*), 215
IsolatedAsyncioTestCase (*clase en unittest*), 1589
isolation_level (*atributo de sqlite3.Connection*), 500
isoweekday() (*método de datetime.date*), 201
isoweekday() (*método de datetime.datetime*), 210

isprint() (*en el módulo curses.ascii*), 777
isprintable() (*método de str*), 52
ispunct() (*en el módulo curses.ascii*), 777
isqrt() (*en el módulo math*), 322
isreadable() (*en el módulo pprint*), 288
isreadable() (*método de pprint.PrettyPrinter*), 289
isrecursive() (*en el módulo pprint*), 289
isrecursive() (*método de pprint.PrettyPrinter*), 289
isreg() (*método de tarfile.TarInfo*), 547
isReservedKey() (*método de http.cookies.Morsel*), 1349
isroutine() (*en el módulo inspect*), 1831
isSameNode() (*método de xml.dom.Node*), 1221
issoftkeyword() (*en el módulo keyword*), 1927
isspace() (*en el módulo curses.ascii*), 777
isspace() (*método de bytearray*), 69
isspace() (*método de bytes*), 69
isspace() (*método de str*), 52
isstdin() (*en el módulo fileinput*), 438
issubclass()
    función incorporada, 16
issubset() (*método de frozenset*), 81
issuperset() (*método de frozenset*), 82
issym() (*método de tarfile.TarInfo*), 547
ISTERMINAL() (*en el módulo token*), 1924
istitle() (*método de bytearray*), 69
istitle() (*método de bytes*), 69
istitle() (*método de str*), 52
istraceback() (*en el módulo inspect*), 1830
isub() (*en el módulo operator*), 412
isupper() (*en el módulo curses.ascii*), 777
isupper() (*método de bytearray*), 69
isupper() (*método de bytes*), 69
isupper() (*método de str*), 52
isvisible() (*en el módulo turtle*), 1424
isxdigit() (*en el módulo curses.ascii*), 777
ITALIC (*en el módulo tkinter.font*), 1468
item() (*método de tkinter.ttk.Treeview*), 1488
item() (*método de xml.dom.NamedNodeMap*), 1225
item() (*método de xml.dom.NodeList*), 1222
itemgetter() (*en el módulo operator*), 409
items() (*método de configparser.ConfigParser*), 577
items() (*método de contextvars.Context*), 915
items() (*método de dict*), 85
items() (*método de email.message.EmailMessage*), 1097
items() (*método de email.message.Message*), 1135
items() (*método de mailbox.Mailbox*), 1163
items() (*método de types.MappingProxyType*), 284
items() (*método de xml.etree.ElementTree.Element*), 1211
itemsize (*atributo de array.array*), 271
itemsize (*atributo de memoryview*), 80
ItemsView (*clase en collections.abc*), 260
ItemsView (*clase en typing*), 1537
iter()
    función incorporada, 16
iter() (*método de xml.etree.ElementTree.Element*), 1212
iter() (*método de xml.etree.ElementTree.ElementTree*), 1213
iter_attachments() (*método de email.message.EmailMessage*), 1100
iter_child_nodes() (*en el módulo ast*), 1919
iter_fields() (*en el módulo ast*), 1919
iter_importers() (*en el módulo pkgutil*), 1857
iter_modules() (*en el módulo pkgutil*), 1857
iter_parts() (*método de email.message.EmailMessage*), 1101
iter_unpack() (*en el módulo struct*), 168
iter_unpack() (*método de struct.Struct*), 173
iterable, **2086**
Iterable (*clase en collections.abc*), 260
Iterable (*clase en typing*), 1538
iterable asincrónico, **2080**
iterador, **2086**
iterador asincrónico, **2080**
iterador generador, **2084**
iterador generador asincrónico, **2080**
Iterator (*clase en collections.abc*), 260
Iterator (*clase en typing*), 1538
iterator protocol, 41
iterdecode() (*en el módulo codecs*), 176
iterdir() (*método de importlib.abc.Traversable*), 1871
iterdir() (*método de pathlib.Path*), 426
iterdir() (*método de zipfile.Path*), 534
iterdump() (*método de sqlite3.Connection*), 499
iterencode() (*en el módulo codecs*), 176
iterencode() (*método de json.JSONEncoder*), 1158
iterfind() (*método de xml.etree.ElementTree.Element*), 1212
iterfind() (*método de xml.etree.ElementTree.ElementTree*), 1213
iteritems() (*método de mailbox.Mailbox*), 1163
iterkeys() (*método de mailbox.Mailbox*), 1163
itermonthdates() (*método de calendar.Calendar*), 234
itermonthdays() (*método de calendar.Calendar*), 235
itermonthdays2() (*método de calendar.Calendar*), 235
itermonthdays3() (*método de calendar.Calendar*), 235
itermonthdays4() (*método de calendar.Calendar*), 235
iterparse() (*en el módulo xml.etree.ElementTree*), 1207
itertext() (*método de xml.etree.ElementTree.Element*), 1212
itertools
    módulo, 381
itertools (*2to3 fixer*), 1663
itertools_imports (*2to3 fixer*), 1663
itervalues() (*método de mailbox.Mailbox*), 1163

iterweekdays() (*método de calendar.Calendar*), 234
ITIMER_PROF (*en el módulo signal*), 1081
ITIMER_REAL (*en el módulo signal*), 1080
ITIMER_VIRTUAL (*en el módulo signal*), 1081
ItimerError, 1081
itruediv() (*en el módulo operator*), 412
ixor() (*en el módulo operator*), 412

J

-j N
    compileallopción de línea de comando, 1937
Jansen, Jack, 2071
java_ver() (*en el módulo platform*), 781
join() (*en el módulo os.path*), 434
join() (*en el módulo shlex*), 1448
join() (*método de asyncio.Queue*), 950
join() (*método de bytearray*), 64
join() (*método de bytes*), 64
join() (*método de multiprocessing.JoinableQueue*), 846
join() (*método de multiprocessing.pool.Pool*), 863
join() (*método de multiprocessing.Process*), 841
join() (*método de queue.Queue*), 911
join() (*método de str*), 52
join() (*método de threading.Thread*), 825
join_thread() (*en el módulo test.support.threading_helper*), 1679
join_thread() (*método de multiprocessing.Queue*), 845
JoinableQueue (*clase en multiprocessing*), 846
JoinedStr (*clase en ast*), 1894
joinpath() (*método de importlib.abc.Traversable*), 1871
joinpath() (*método de pathlib.PurePath*), 421
joinpath() (*método de zipfile.Path*), 535
js_output() (*método de http.cookies.BaseCookie*), 1348
js_output() (*método de http.cookies.Morsel*), 1349
json
    módulo, 1152
JSONDecodeError, 1158
JSONDecoder (*clase en json*), 1156
JSONEncoder (*clase en json*), 1156
--json-lines
    json.toolopción de línea de comando, 1161
json.tool
    módulo, 1160
json.toolopción de línea de comando
    --compact, 1161
    -h, 1161
    --help, 1161
    --indent, 1161
    infile, 1161
    --json-lines, 1161
    --no-ensure-ascii, 1161
    --no-indent, 1161
    outfile, 1161
    --sort-keys, 1161
    --tab, 1161
jump (*pdb command*), 1701
JUMP_ABSOLUTE (*opcode*), 1950
JUMP_FORWARD (*opcode*), 1950
JUMP_IF_FALSE_OR_POP (*opcode*), 1950
JUMP_IF_NOT_EXC_MATCH (*opcode*), 1950
JUMP_IF_TRUE_OR_POP (*opcode*), 1950

K

-k
    unittestopción de línea de comando, 1574
kbhit() (*en el módulo msvcrt*), 1958
KDEDIR, 1259
kevent() (*en el módulo select*), 1069
key (*atributo de http.cookies.Morsel*), 1348
key (*atributo de zoneinfo.ZoneInfo*), 232
KEY_ALL_ACCESS (*en el módulo winreg*), 1965
KEY_CREATE_LINK (*en el módulo winreg*), 1966
KEY_CREATE_SUB_KEY (*en el módulo winreg*), 1965
KEY_ENUMERATE_SUB_KEYS (*en el módulo winreg*), 1965
KEY_EXECUTE (*en el módulo winreg*), 1965
KEY_NOTIFY (*en el módulo winreg*), 1966
KEY_QUERY_VALUE (*en el módulo winreg*), 1965
KEY_READ (*en el módulo winreg*), 1965
KEY_SET_VALUE (*en el módulo winreg*), 1965
KEY_WOW64_32KEY (*en el módulo winreg*), 1966
KEY_WOW64_64KEY (*en el módulo winreg*), 1966
KEY_WRITE (*en el módulo winreg*), 1965
KeyboardInterrupt, 103
KeyError, 103
keylog_filename (*atributo de ssl.SSLContext*), 1056
keyname() (*en el módulo curses*), 759
keypad() (*método de curses.window*), 767
keyrefs() (*método de weakref.WeakKeyDictionary*), 275
keys() (*método de contextvars.Context*), 915
keys() (*método de dict*), 85
keys() (*método de email.message.EmailMessage*), 1097
keys() (*método de email.message.Message*), 1135
keys() (*método de mailbox.Mailbox*), 1163
keys() (*método de sqlite3.Row*), 504
keys() (*método de types.MappingProxyType*), 284
keys() (*método de xml.etree.ElementTree.Element*), 1211
KeysView (*clase en collections.abc*), 260
KeysView (*clase en typing*), 1537
keyword
    módulo, 1927
keyword (*clase en ast*), 1898
keywords (*atributo de functools.partial*), 405
kill() (*en el módulo os*), 644
kill() (*método de asyncio.subprocess.Process*), 948

kill() (*método de asyncio.SubprocessTransport*), 983
kill() (*método de multiprocessing.Process*), 843
kill() (*método de subprocess.Popen*), 899
kill_python() (*en el módulo test.support.script_helper*), 1678
killchar() (*en el módulo curses*), 759
killpg() (*en el módulo os*), 644
kind (*atributo de inspect.Parameter*), 1834
knownfiles (*en el módulo mimetypes*), 1180
kqueue() (*en el módulo select*), 1069
KqueueSelector (*clase en selectors*), 1077
KW_ONLY (*en el módulo dataclasses*), 1791
kwargs (*atributo de inspect.BoundArguments*), 1836
kwargs (*atributo de typing.ParamSpec*), 1529
kwlist (*en el módulo keyword*), 1927

L

-l
    compileallopción de línea de comando, 1936
    pickletoolsopción de línea de comando, 1954
    traceopción de línea de comando, 1717
L (*en el módulo re*), 130
-l <tarfile>
    tarfileopción de línea de comando, 551
-l <zipfile>
    zipfileopción de línea de comando, 538
LabelEntry (*clase en tkinter.tix*), 1494
LabelFrame (*clase en tkinter.tix*), 1494
lambda, **2086**
Lambda (*clase en ast*), 1914
LambdaType (*en el módulo types*), 281
LANG, 1391, 1393, 1401, 1403
language
    C, 35
LANGUAGE, 1391, 1393
large files, 1971
LARGEST (*en el módulo test.support*), 1670
LargeZipFile, 529
last() (*método de nntplib.NNTP*), 2027
last_accepted (*atributo de multiprocessing.connection.Listener*), 865
last_traceback (*en el módulo sys*), 1760
last_type (*en el módulo sys*), 1760
last_value (*en el módulo sys*), 1760
lastChild (*atributo de xml.dom.Node*), 1221
lastcmd (*atributo de cmd.Cmd*), 1445
lastgroup (*atributo de re.Match*), 138
lastindex (*atributo de re.Match*), 138
lastResort (*en el módulo logging*), 730
lastrowid (*atributo de sqlite3.Cursor*), 503
layout() (*método de tkinter.ttk.Style*), 1491
lazycache() (*en el módulo linecache*), 454
LazyLoader (*clase en importlib.util*), 1881
LBRACE (*en el módulo token*), 1925
LBYL, **2086**
LC_ALL, 1391, 1393
LC_ALL (*en el módulo locale*), 1405
LC_COLLATE (*en el módulo locale*), 1405
LC_CTYPE (*en el módulo locale*), 1405
LC_MESSAGES, 1391, 1393
LC_MESSAGES (*en el módulo locale*), 1405
LC_MONETARY (*en el módulo locale*), 1405
LC_NUMERIC (*en el módulo locale*), 1405
LC_TIME (*en el módulo locale*), 1405
lchflags() (*en el módulo os*), 622
lchmod() (*en el módulo os*), 623
lchmod() (*método de pathlib.Path*), 426
lchown() (*en el módulo os*), 623
lcm() (*en el módulo math*), 322
ldexp() (*en el módulo math*), 323
ldgettext() (*en el módulo gettext*), 1392
ldngettext() (*en el módulo gettext*), 1392
le() (*en el módulo operator*), 406
leapdays() (*en el módulo calendar*), 237
leaveok() (*método de curses.window*), 767
left (*atributo de filecmp.dircmp*), 445
left() (*en el módulo turtle*), 1414
left_list (*atributo de filecmp.dircmp*), 445
left_only (*atributo de filecmp.dircmp*), 445
LEFTSHIFT (*en el módulo token*), 1925
LEFTSHIFTEQUAL (*en el módulo token*), 1926
len
    función incorporada, 42, 83
len()
    función incorporada, 16
length (*atributo de xml.dom.NamedNodeMap*), 1225
length (*atributo de xml.dom.NodeList*), 1222
length_hint() (*en el módulo operator*), 408
LESS (*en el módulo token*), 1925
LESSEQUAL (*en el módulo token*), 1925
LexicalHandler (*clase en xml.sax.handler*), 1237
lexists() (*en el módulo os.path*), 433
lgamma() (*en el módulo math*), 326
lgettext() (*en el módulo gettext*), 1392
lgettext() (*método de gettext.GNUTranslations*), 1396
lgettext() (*método de gettext.NullTranslations*), 1394
lib2to3
    módulo, 1665
libc_ver() (*en el módulo platform*), 782
library (*atributo de ssl.SSLError*), 1036
library (*en el módulo dbm.ndbm*), 489
LibraryLoader (*clase en ctypes*), 808
license (*variable incorporada*), 32
LifoQueue (*clase en asyncio*), 951
LifoQueue (*clase en queue*), 910
light-weight processes, 916
limit_denominator() (*método de fractions.Fraction*), 359
LimitOverrunError, 953

lin2adpcm() (*en el módulo audioop*), 1995
lin2alaw() (*en el módulo audioop*), 1995
lin2lin() (*en el módulo audioop*), 1995
lin2ulaw() (*en el módulo audioop*), 1996
line (*atributo de bdb.Breakpoint*), 1690
line() (*método de msilib.Dialog*), 2020
line_buffering (*atributo de io.TextIOWrapper*), 668
line_num (*atributo de csv.csvreader*), 560
linear_regression() (*en el módulo statistics*), 376
line-buffered I/O, 20
linecache
    módulo, 453
lineno (*atributo de ast.AST*), 1892
lineno (*atributo de doctest.DocTest*), 1563
lineno (*atributo de doctest.Example*), 1564
lineno (*atributo de json.JSONDecodeError*), 1158
lineno (*atributo de pyclbr.Class*), 1933
lineno (*atributo de pyclbr.Function*), 1933
lineno (*atributo de re.error*), 135
lineno (*atributo de shlex.shlex*), 1451
lineno (*atributo de SyntaxError*), 106
lineno (*atributo de traceback.TracebackException*), 1818
lineno (*atributo de tracemalloc.Filter*), 1726
lineno (*atributo de tracemalloc.Frame*), 1726
lineno (*atributo de xml.parsers.expat.ExpatError*), 1252
lineno() (*en el módulo fileinput*), 438
LINES, 758, 763
lines (*atributo de os.terminal_size*), 618
linesep (*atributo de email.policy.Policy*), 1111
linesep (*en el módulo os*), 655
lineterminator (*atributo de csv.Dialect*), 559
LineTooLong, 1303
link() (*en el módulo os*), 623
link_to() (*método de pathlib.Path*), 429
linkname (*atributo de tarfile.TarInfo*), 546
LinkOutsideDestinationError, 541
list
    objeto, 44, 45
    type, operations on, 44
List (*clase en ast*), 1894
List (*clase en typing*), 1534
list (*clase incorporada*), 45
list (*pdb command*), 1701
--list <tarfile>
    tarfileopción de línea de comando, 551
--list <zipfile>
    zipfileopción de línea de comando, 538
list() (*método de imaplib.IMAP4*), 1319
list() (*método de multiprocessing.managers.SyncManager*), 857
list() (*método de nntplib.NNTP*), 2025
list() (*método de poplib.POP3*), 1315
list() (*método de tarfile.TarFile*), 543
LIST_APPEND (*opcode*), 1946
list_dialects() (*en el módulo csv*), 557
LIST_EXTEND (*opcode*), 1949
list_folders() (*método de mailbox.Maildir*), 1165
list_folders() (*método de mailbox.MH*), 1167
LIST_TO_TUPLE (*opcode*), 1949
lista, **2087**
ListComp (*clase en ast*), 1899
listdir() (*en el módulo os*), 623
listen() (*en el módulo logging.config*), 732
listen() (*en el módulo turtle*), 1432
listen() (*método de asyncore.dispatcher*), 1992
listen() (*método de socket.socket*), 1024
listener (*atributo de logging.handlers.QueueHandler*), 754
Listener (*clase en multiprocessing.connection*), 864
--listfuncs
    traceopción de línea de comando, 1717
listMethods() (*método de xmlrpc.client.ServerProxy.system*), 1361
ListNoteBook (*clase en tkinter.tix*), 1496
listxattr() (*en el módulo os*), 640
Literal (*en el módulo typing*), 1523
literal_eval() (*en el módulo ast*), 1918
literals
    binary, 35
    complex number, 35
    floating point, 35
    hexadecimal, 35
    integer, 35
    numeric, 35
    octal, 35
LittleEndianStructure (*clase en ctypes*), 818
ljust() (*método de bytearray*), 65
ljust() (*método de bytes*), 65
ljust() (*método de str*), 52
LK_LOCK (*en el módulo msvcrt*), 1957
LK_NBLCK (*en el módulo msvcrt*), 1957
LK_NBRLCK (*en el módulo msvcrt*), 1957
LK_RLCK (*en el módulo msvcrt*), 1957
LK_UNLCK (*en el módulo msvcrt*), 1958
ll (*pdb command*), 1701
LMTP (*clase en smtplib*), 1324
ln() (*método de decimal.Context*), 346
ln() (*método de decimal.Decimal*), 339
LNAME, 756
lngettext() (*en el módulo gettext*), 1392
lngettext() (*método de gettext.GNUTranslations*), 1396
lngettext() (*método de gettext.NullTranslations*), 1394
Load (*clase en ast*), 1895
load() (*en el módulo json*), 1155
load() (*en el módulo marshal*), 485
load() (*en el módulo pickle*), 468
load() (*en el módulo plistlib*), 582

load() (*método de clase de tracemalloc.Snapshot*), 1727
load() (*método de http.cookiejar.FileCookieJar*), 1353
load() (*método de http.cookies.BaseCookie*), 1348
load() (*método de pickle.Unpickler*), 470
LOAD_ASSERTION_ERROR (*opcode*), 1947
LOAD_ATTR (*opcode*), 1949
LOAD_BUILD_CLASS (*opcode*), 1947
load_cert_chain() (*método de ssl.SSLContext*), 1051
LOAD_CLASSDEREF (*opcode*), 1950
LOAD_CLOSURE (*opcode*), 1950
LOAD_CONST (*opcode*), 1948
load_default_certs() (*método de ssl.SSLContext*), 1052
LOAD_DEREF (*opcode*), 1950
load_dh_params() (*método de ssl.SSLContext*), 1054
load_extension() (*método de sqlite3.Connection*), 499
LOAD_FAST (*opcode*), 1950
LOAD_GLOBAL (*opcode*), 1950
LOAD_METHOD (*opcode*), 1951
load_module() (*en el módulo imp*), 2010
load_module() (*método de importlib.abc.FileLoader*), 1870
load_module() (*método de importlib.abc.InspectLoader*), 1869
load_module() (*método de importlib.abc.Loader*), 1867
load_module() (*método de importlib.abc.SourceLoader*), 1871
load_module() (*método de importlib.machinery.SourceFileLoader*), 1876
load_module() (*método de importlib.machinery.SourcelessFileLoader*), 1877
load_module() (*método de zipimport.zipimporter*), 1854
LOAD_NAME (*opcode*), 1948
load_package_tests() (*en el módulo test.support*), 1674
load_verify_locations() (*método de ssl.SSLContext*), 1052
loader (*atributo de importlib.machinery.ModuleSpec*), 1878
Loader (*clase en importlib.abc*), 1866
loader_state (*atributo de importlib.machinery.ModuleSpec*), 1878
LoadError, 1350
LoadFileDialog (*clase en tkinter.filedialog*), 1472
LoadKey() (*en el módulo winreg*), 1961
LoadLibrary() (*método de ctypes.LibraryLoader*), 808
loads() (*en el módulo json*), 1155
loads() (*en el módulo marshal*), 486
loads() (*en el módulo pickle*), 468
loads() (*en el módulo plistlib*), 582
loads() (*en el módulo xmlrpc.client*), 1365
loadTestsFromModule() (*método de unittest.TestLoader*), 1592
loadTestsFromName() (*método de unittest.TestLoader*), 1593
loadTestsFromNames() (*método de unittest.TestLoader*), 1593
loadTestsFromTestCase() (*método de unittest.TestLoader*), 1592
local (*clase en threading*), 824
localcontext() (*en el módulo decimal*), 342
locale
    módulo, 1400
LOCALE (*en el módulo re*), 130
localeconv() (*en el módulo locale*), 1401
LocaleHTMLCalendar (*clase en calendar*), 237
LocaleTextCalendar (*clase en calendar*), 237
localize() (*en el módulo locale*), 1405
localName (*atributo de xml.dom.Attr*), 1225
localName (*atributo de xml.dom.Node*), 1221
--locals
    unittestopción de línea de comando, 1574
locals()
    función incorporada, 17
localtime() (*en el módulo email.utils*), 1148
localtime() (*en el módulo time*), 672
Locator (*clase en xml.sax.xmlreader*), 1243
Lock (*clase en asyncio*), 941
Lock (*clase en multiprocessing*), 850
Lock (*clase en threading*), 826
lock() (*método de mailbox.Babyl*), 1169
lock() (*método de mailbox.Mailbox*), 1164
lock() (*método de mailbox.Maildir*), 1166
lock() (*método de mailbox.mbox*), 1166
lock() (*método de mailbox.MH*), 1168
lock() (*método de mailbox.MMDF*), 1169
Lock() (*método de multiprocessing.managers.SyncManager*), 857
lock_held() (*en el módulo imp*), 2012
locked() (*método de _thread.lock*), 918
locked() (*método de asyncio.Condition*), 943
locked() (*método de asyncio.Lock*), 942
locked() (*método de asyncio.Semaphore*), 944
locked() (*método de threading.Lock*), 827
lockf() (*en el módulo fcntl*), 1978
lockf() (*en el módulo os*), 611
locking() (*en el módulo msvcrt*), 1957
LockType (*en el módulo _thread*), 916
log() (*en el módulo cmath*), 329
log() (*en el módulo logging*), 727
log() (*en el módulo math*), 324
log() (*método de logging.Logger*), 717
log1p() (*en el módulo math*), 324
log2() (*en el módulo math*), 324
log10() (*en el módulo cmath*), 329
log10() (*en el módulo math*), 325
log10() (*método de decimal.Context*), 346

log10() (*método de decimal.Decimal*), 339
log_date_time_string() (*método de http.server.BaseHTTPRequestHandler*), 1344
log_error() (*método de http.server.BaseHTTPRequestHandler*), 1344
log_exception() (*método de wsgiref.handlers.BaseHandler*), 1267
log_message() (*método de http.server.BaseHTTPRequestHandler*), 1344
log_request() (*método de http.server.BaseHTTPRequestHandler*), 1344
log_to_stderr() (*en el módulo multiprocessing*), 867
logb() (*método de decimal.Context*), 346
logb() (*método de decimal.Decimal*), 339
Logger (*clase en logging*), 714
LoggerAdapter (*clase en logging*), 725
logging
    Errors, 714
    módulo, 714
logging.config
    módulo, 730
logging.handlers
    módulo, 742
logical_and() (*método de decimal.Context*), 346
logical_and() (*método de decimal.Decimal*), 339
logical_invert() (*método de decimal.Context*), 346
logical_invert() (*método de decimal.Decimal*), 340
logical_or() (*método de decimal.Context*), 346
logical_or() (*método de decimal.Decimal*), 340
logical_xor() (*método de decimal.Context*), 346
logical_xor() (*método de decimal.Decimal*), 340
login() (*método de ftplib.FTP*), 1310
login() (*método de imaplib.IMAP4*), 1319
login() (*método de nntplib.NNTP*), 2024
login() (*método de smtplib.SMTP*), 1326
login_cram_md5() (*método de imaplib.IMAP4*), 1319
LOGNAME, 605, 756
lognormvariate() (*en el módulo random*), 364
logout() (*método de imaplib.IMAP4*), 1319
LogRecord (*clase en logging*), 722
long (*2to3 fixer*), 1663
LONG_TIMEOUT (*en el módulo test.support*), 1669
longMessage (*atributo de unittest.TestCase*), 1588
longname() (*en el módulo curses*), 759
lookup() (*en el módulo codecs*), 174
lookup() (*en el módulo unicodedata*), 158
lookup() (*método de symtable.SymbolTable*), 1922
lookup() (*método de tkinter.ttk.Style*), 1491
lookup_error() (*en el módulo codecs*), 178
LookupError, 103
loop
    over mutable sequence, 42
loop() (*en el módulo asyncore*), 1990
LOOPBACK_TIMEOUT (*en el módulo test.support*), 1669
lower() (*método de bytearray*), 70
lower() (*método de bytes*), 70
lower() (*método de str*), 53
LPAR (*en el módulo token*), 1924
lpAttributeList (*atributo de subprocess.STARTUPINFO*), 900
lru_cache() (*en el módulo functools*), 398
lseek() (*en el módulo os*), 612
LShift (*clase en ast*), 1896
lshift() (*en el módulo operator*), 407
LSQB (*en el módulo token*), 1924
lstat() (*en el módulo os*), 624
lstat() (*método de pathlib.Path*), 426
lstrip() (*método de bytearray*), 65
lstrip() (*método de bytes*), 65
lstrip() (*método de str*), 53
lsub() (*método de imaplib.IMAP4*), 1319
Lt (*clase en ast*), 1897
lt() (*en el módulo operator*), 406
lt() (*en el módulo turtle*), 1414
LtE (*clase en ast*), 1897
LWPCookieJar (*clase en http.cookiejar*), 1354
lzma
    módulo, 524
LZMACompressor (*clase en lzma*), 525
LZMADecompressor (*clase en lzma*), 526
LZMAError, 524
LZMAFile (*clase en lzma*), 524

M

-m
    pickletoolsopción de línea de comando, 1954
    traceopción de línea de comando, 1717
M (*en el módulo re*), 130
-m <mainfn>
    zipappopción de línea de comando, 1743
-m <mode>
    astopción de línea de comando, 1921
mac_ver() (*en el módulo platform*), 781
machine() (*en el módulo platform*), 779
macros (*atributo de netrc.netrc*), 581
MADV_AUTOSYNC (*en el módulo mmap*), 1090
MADV_CORE (*en el módulo mmap*), 1090
MADV_DODUMP (*en el módulo mmap*), 1090
MADV_DOFORK (*en el módulo mmap*), 1090
MADV_DONTDUMP (*en el módulo mmap*), 1090
MADV_DONTFORK (*en el módulo mmap*), 1090
MADV_DONTNEED (*en el módulo mmap*), 1090
MADV_FREE (*en el módulo mmap*), 1090
MADV_FREE_REUSABLE (*en el módulo mmap*), 1090
MADV_FREE_REUSE (*en el módulo mmap*), 1090
MADV_HUGEPAGE (*en el módulo mmap*), 1090
MADV_HWPOISON (*en el módulo mmap*), 1090

MADV_MERGEABLE (*en el módulo mmap*), 1090
MADV_NOCORE (*en el módulo mmap*), 1090
MADV_NOHUGEPAGE (*en el módulo mmap*), 1090
MADV_NORMAL (*en el módulo mmap*), 1090
MADV_NOSYNC (*en el módulo mmap*), 1090
MADV_PROTECT (*en el módulo mmap*), 1090
MADV_RANDOM (*en el módulo mmap*), 1090
MADV_REMOVE (*en el módulo mmap*), 1090
MADV_SEQUENTIAL (*en el módulo mmap*), 1090
MADV_SOFT_OFFLINE (*en el módulo mmap*), 1090
MADV_UNMERGEABLE (*en el módulo mmap*), 1090
MADV_WILLNEED (*en el módulo mmap*), 1090
madvise() (*método de mmap.mmap*), 1089
magic
    method, 2087
MAGIC_NUMBER (*en el módulo importlib.util*), 1878
MagicMock (*clase en unittest.mock*), 1630
mailbox
    módulo, 1161
Mailbox (*clase en mailbox*), 1162
mailcap
    módulo, 2014
Maildir (*clase en mailbox*), 1165
MaildirMessage (*clase en mailbox*), 1170
mailfrom (*atributo de smtpd.SMTPChannel*), 2063
MailmanProxy (*clase en smtpd*), 2063
main() (*en el módulo site*), 1846
main() (*en el módulo unittest*), 1597
--main=<mainfn>
    zipappopción de línea de comando, 1743
main_thread() (*en el módulo threading*), 823
mainloop() (*en el módulo turtle*), 1434
maintype (*atributo de email.headerregistry.ContentTypeHeader*), 1120
major (*atributo de email.headerregistry.MIMEVersionHeader*), 1120
major() (*en el módulo os*), 625
make_alternative() (*método de email.message.EmailMessage*), 1101
make_archive() (*en el módulo shutil*), 460
make_bad_fd() (*en el módulo test.support.os_helper*), 1681
make_cookies() (*método de http.cookiejar.CookieJar*), 1352
make_dataclass() (*en el módulo dataclasses*), 1790
make_file() (*método de difflib.HtmlDiff*), 145
MAKE_FUNCTION (*opcode*), 1952
make_header() (*en el módulo email.header*), 1145
make_legacy_pyc() (*en el módulo test.support.import_helper*), 1683
make_mixed() (*método de email.message.EmailMessage*), 1101
make_msgid() (*en el módulo email.utils*), 1148
make_parser() (*en el módulo xml.sax*), 1235
make_pkg() (*en el módulo test.support.script_helper*), 1678
make_related() (*método de email.message.EmailMessage*), 1101
make_script() (*en el módulo test.support.script_helper*), 1678
make_server() (*en el módulo wsgiref.simple_server*), 1263
make_table() (*método de difflib.HtmlDiff*), 145
make_zip_pkg() (*en el módulo test.support.script_helper*), 1678
make_zip_script() (*en el módulo test.support.script_helper*), 1678
makedev() (*en el módulo os*), 625
makedirs() (*en el módulo os*), 624
makeelement() (*método de xml.etree.ElementTree.Element*), 1212
makefile() (*método de socket.socket*), 1024
makeLogRecord() (*en el módulo logging*), 728
makePickle() (*método de logging.handlers.SocketHandler*), 748
makeRecord() (*método de logging.Logger*), 718
makeSocket() (*método de logging.handlers.DatagramHandler*), 749
makeSocket() (*método de logging.handlers.SocketHandler*), 748
maketrans() (*método estático de bytearray*), 64
maketrans() (*método estático de bytes*), 64
maketrans() (*método estático de str*), 53
mangle_from_ (*atributo de email.policy.Compat32*), 1115
mangle_from_ (*atributo de email.policy.Policy*), 1111
map (*2to3 fixer*), 1663
map()
    función incorporada, 17
map() (*método de concurrent.futures.Executor*), 882
map() (*método de multiprocessing.pool.Pool*), 862
map() (*método de tkinter.ttk.Style*), 1490
MAP_ADD (*opcode*), 1946
MAP_ANON (*en el módulo mmap*), 1091
MAP_ANONYMOUS (*en el módulo mmap*), 1091
map_async() (*método de multiprocessing.pool.Pool*), 862
MAP_DENYWRITE (*en el módulo mmap*), 1091
MAP_EXECUTABLE (*en el módulo mmap*), 1091
MAP_POPULATE (*en el módulo mmap*), 1091
MAP_PRIVATE (*en el módulo mmap*), 1091
MAP_SHARED (*en el módulo mmap*), 1091
map_table_b2() (*en el módulo stringprep*), 161
map_table_b3() (*en el módulo stringprep*), 161
map_to_type() (*método de email.headerregistry.HeaderRegistry*), 1121
mapeado, **2087**
mapLogRecord() (*método de logging.handlers.HTTPHandler*), 753
mapping
    objeto, 83
    types, operations on, 83
Mapping (*clase en collections.abc*), 260
Mapping (*clase en typing*), 1537

mapping() (*método de msilib.Control*), 2020
MappingProxyType (*clase en types*), 284
MappingView (*clase en collections.abc*), 260
MappingView (*clase en typing*), 1537
mapPriority() (*método de logging.handlers.SysLogHandler*), 750
maps (*atributo de collections.ChainMap*), 239
maps() (*en el módulo nis*), 2021
máquina virtual, **2092**
marshal
    módulo, 485
marshalling
    objects, 465
masking
    operations, 36
master (*atributo de tkinter.Tk*), 1457
Match (*clase en ast*), 1908
Match (*clase en typing*), 1536
match() (*en el módulo nis*), 2021
match() (*en el módulo re*), 132
match() (*método de pathlib.PurePath*), 421
match() (*método de re.Pattern*), 135
match_case (*clase en ast*), 1908
MATCH_CLASS (*opcode*), 1952
match_hostname() (*en el módulo ssl*), 1037
MATCH_KEYS (*opcode*), 1948
MATCH_MAPPING (*opcode*), 1948
MATCH_SEQUENCE (*opcode*), 1948
match_test() (*en el módulo test.support*), 1670
match_value() (*método de test.support.Matcher*), 1676
MatchAs (*clase en ast*), 1912
MatchClass (*clase en ast*), 1911
Matcher (*clase en test.support*), 1676
matches() (*método de test.support.Matcher*), 1676
MatchMapping (*clase en ast*), 1911
MatchOr (*clase en ast*), 1913
MatchSequence (*clase en ast*), 1910
MatchSingleton (*clase en ast*), 1909
MatchStar (*clase en ast*), 1910
MatchValue (*clase en ast*), 1909
math
    módulo, 36, 320, 331
matmul() (*en el módulo operator*), 407
MatMult (*clase en ast*), 1896
max
    función incorporada, 42
max (*atributo de datetime.date*), 199
max (*atributo de datetime.datetime*), 206
max (*atributo de datetime.time*), 214
max (*atributo de datetime.timedelta*), 196
max()
    función incorporada, 17
max() (*en el módulo audioop*), 1996
max() (*método de decimal.Context*), 346
max() (*método de decimal.Decimal*), 340
max_count (*atributo de email.headerregistry.BaseHeader*), 1118
MAX_EMAX (*en el módulo decimal*), 348
MAX_INTERPOLATION_DEPTH (*en el módulo configparser*), 578
max_line_length (*atributo de email.policy.Policy*), 1111
max_lines (*atributo de textwrap.TextWrapper*), 158
max_mag() (*método de decimal.Context*), 346
max_mag() (*método de decimal.Decimal*), 340
max_memuse (*en el módulo test.support*), 1670
MAX_PREC (*en el módulo decimal*), 348
max_prefixlen (*atributo de ipaddress.IPv4Address*), 1373
max_prefixlen (*atributo de ipaddress.IPv4Network*), 1378
max_prefixlen (*atributo de ipaddress.IPv6Address*), 1375
max_prefixlen (*atributo de ipaddress.IPv6Network*), 1381
MAX_Py_ssize_t (*en el módulo test.support*), 1669
maxarray (*atributo de reprlib.Repr*), 293
maxdeque (*atributo de reprlib.Repr*), 293
maxdict (*atributo de reprlib.Repr*), 293
maxDiff (*atributo de unittest.TestCase*), 1588
maxfrozenset (*atributo de reprlib.Repr*), 293
MAXIMUM_SUPPORTED (*atributo de ssl.TLSVersion*), 1046
maximum_version (*atributo de ssl.SSLContext*), 1056
maxlen (*atributo de collections.deque*), 246
maxlevel (*atributo de reprlib.Repr*), 293
maxlist (*atributo de reprlib.Repr*), 293
maxlong (*atributo de reprlib.Repr*), 293
maxother (*atributo de reprlib.Repr*), 293
maxpp() (*en el módulo audioop*), 1996
maxset (*atributo de reprlib.Repr*), 293
maxsize (*atributo de asyncio.Queue*), 950
maxsize (*en el módulo sys*), 1761
maxstring (*atributo de reprlib.Repr*), 293
maxtuple (*atributo de reprlib.Repr*), 293
maxunicode (*en el módulo sys*), 1761
MAXYEAR (*en el módulo datetime*), 194
MB_ICONASTERISK (*en el módulo winsound*), 1969
MB_ICONEXCLAMATION (*en el módulo winsound*), 1969
MB_ICONHAND (*en el módulo winsound*), 1969
MB_ICONQUESTION (*en el módulo winsound*), 1969
MB_OK (*en el módulo winsound*), 1969
mbox (*clase en mailbox*), 1166
mboxMessage (*clase en mailbox*), 1172
mean (*atributo de statistics.NormalDist*), 377
mean() (*en el módulo statistics*), 370
measure() (*método de tkinter.font.Font*), 1469
median (*atributo de statistics.NormalDist*), 377
median() (*en el módulo statistics*), 371
median_grouped() (*en el módulo statistics*), 372
median_high() (*en el módulo statistics*), 372
median_low() (*en el módulo statistics*), 371
MemberDescriptorType (*en el módulo types*), 284
memfd_create() (*en el módulo os*), 638

memmove() (*en el módulo ctypes*), 814
--memo
    pickletoolsopción de línea de comando, 1954
MemoryBIO (*clase en ssl*), 1065
MemoryError, 104
MemoryHandler (*clase en logging.handlers*), 752
memoryview
    objeto, 59
memoryview (*clase incorporada*), 74
memset() (*en el módulo ctypes*), 814
merge() (*en el módulo heapq*), 263
Message (*clase en email.message*), 1132
Message (*clase en mailbox*), 1170
Message (*clase en tkinter.messagebox*), 1472
message digest, MD5, 585
message_factory (*atributo de email.policy.Policy*), 1111
message_from_binary_file() (*en el módulo email*), 1105
message_from_bytes() (*en el módulo email*), 1105
message_from_file() (*en el módulo email*), 1105
message_from_string() (*en el módulo email*), 1105
MessageBeep() (*en el módulo winsound*), 1968
MessageClass (*atributo de http.server.BaseHTTPRequestHandler*), 1342
MessageError, 1116
MessageParseError, 1116
messages (*en el módulo xml.parsers.expat.errors*), 1254
meta buscadores de ruta, **2087**
meta() (*en el módulo curses*), 760
meta_path (*en el módulo sys*), 1761
metaclase, **2087**
metaclass (*2to3 fixer*), 1663
MetaPathFinder (*clase en importlib.abc*), 1865
metavar (*atributo de optparse.Option*), 2042
MetavarTypeHelpFormatter (*clase en argparse*), 685
Meter (*clase en tkinter.tix*), 1494
method
    magic, 2087
    objeto, 95
    special, 2091
method (*atributo de urllib.request.Request*), 1275
METHOD_BLOWFISH (*en el módulo crypt*), 2007
method_calls (*atributo de unittest.mock.Mock*), 1610
METHOD_CRYPT (*en el módulo crypt*), 2007
METHOD_MD5 (*en el módulo crypt*), 2007
METHOD_SHA256 (*en el módulo crypt*), 2007
METHOD_SHA512 (*en el módulo crypt*), 2007
methodattrs (*2to3 fixer*), 1664
methodcaller() (*en el módulo operator*), 409
MethodDescriptorType (*en el módulo types*), 282
methodHelp() (*método de xmlrpc.client.ServerProxy.system*), 1361
methods
    bytearray, 62
    bytes, 62
    string, 49
methods (*atributo de pyclbr.Class*), 1934
methods (*en el módulo crypt*), 2007
methodSignature() (*método de xmlrpc.client.ServerProxy.system*), 1361
MethodType (*en el módulo types*), 282
MethodWrapperType (*en el módulo types*), 282
método, **2087**
método especial, **2091**
método mágico, **2087**
metrics() (*método de tkinter.font.Font*), 1469
MFD_ALLOW_SEALING (*en el módulo os*), 638
MFD_CLOEXEC (*en el módulo os*), 638
MFD_HUGE_1GB (*en el módulo os*), 638
MFD_HUGE_1MB (*en el módulo os*), 638
MFD_HUGE_2GB (*en el módulo os*), 638
MFD_HUGE_2MB (*en el módulo os*), 638
MFD_HUGE_8MB (*en el módulo os*), 638
MFD_HUGE_16GB (*en el módulo os*), 638
MFD_HUGE_16MB (*en el módulo os*), 638
MFD_HUGE_32MB (*en el módulo os*), 638
MFD_HUGE_64KB (*en el módulo os*), 638
MFD_HUGE_256MB (*en el módulo os*), 638
MFD_HUGE_512KB (*en el módulo os*), 638
MFD_HUGE_512MB (*en el módulo os*), 638
MFD_HUGE_MASK (*en el módulo os*), 638
MFD_HUGE_SHIFT (*en el módulo os*), 638
MFD_HUGETLB (*en el módulo os*), 638
MH (*clase en mailbox*), 1167
MHMessage (*clase en mailbox*), 1173
microsecond (*atributo de datetime.datetime*), 206
microsecond (*atributo de datetime.time*), 214
MIME
    base64 encoding, 1182
    content type, 1179
    headers, 1179, 1997
    quoted-printable encoding, 1188
MIMEApplication (*clase en email.mime.application*), 1141
MIMEAudio (*clase en email.mime.audio*), 1141
MIMEBase (*clase en email.mime.base*), 1140
MIMEImage (*clase en email.mime.image*), 1141
MIMEMessage (*clase en email.mime.message*), 1142
MIMEMultipart (*clase en email.mime.multipart*), 1140
MIMENonMultipart (*clase en email.mime.nonmultipart*), 1140
MIMEPart (*clase en email.message*), 1102
MIMEText (*clase en email.mime.text*), 1142
mimetypes
    módulo, 1179
MimeTypes (*clase en mimetypes*), 1181
MIMEVersionHeader (*clase en email.headerregistry*), 1120
min

función incorporada, 42
min (atributo de datetime.date), 199
min (atributo de datetime.datetime), 206
min (atributo de datetime.time), 214
min (atributo de datetime.timedelta), 196
min()
función incorporada, 17
min() (método de decimal.Context), 346
min() (método de decimal.Decimal), 340
MIN_EMIN (en el módulo decimal), 348
MIN_ETINY (en el módulo decimal), 348
min_mag() (método de decimal.Context), 346
min_mag() (método de decimal.Decimal), 340
MINEQUAL (en el módulo token), 1926
MINIMUM_SUPPORTED (atributo de ssl.TLSVersion), 1046
minimum_version (atributo de ssl.SSLContext), 1056
minmax() (en el módulo audioop), 1996
minor (atributo de email.headerregistry.MIMEVersionHeader), 1120
minor() (en el módulo os), 625
MINUS (en el módulo token), 1925
minus() (método de decimal.Context), 346
minute (atributo de datetime.datetime), 206
minute (atributo de datetime.time), 214
MINYEAR (en el módulo datetime), 194
mirrored() (en el módulo unicodedata), 159
misc_header (atributo de cmd.Cmd), 1445
--missing
traceopción de línea de comando, 1717
MISSING (atributo de contextvars.Token), 914
MISSING (en el módulo dataclasses), 1791
MISSING_C_DOCSTRINGS (en el módulo test.support), 1670
missing_compiler_executable() (en el módulo test.support), 1675
MissingSectionHeaderError, 580
MIXERDEV, 2056
mkd() (método de ftplib.FTP), 1312
mkdir() (en el módulo os), 624
mkdir() (método de pathlib.Path), 427
mkdtemp() (en el módulo tempfile), 448
mkfifo() (en el módulo os), 625
mknod() (en el módulo os), 625
mksalt() (en el módulo crypt), 2007
mkstemp() (en el módulo tempfile), 448
mktemp() (en el módulo tempfile), 450
mktime() (en el módulo time), 673
mktime_tz() (en el módulo email.utils), 1149
mlsd() (método de ftplib.FTP), 1312
mmap
módulo, 1087
mmap (clase en mmap), 1087
MMDF (clase en mailbox), 1169
MMDFMessage (clase en mailbox), 1176
Mock (clase en unittest.mock), 1604
mock_add_spec() (método de unittest.mock.Mock), 1607
mock_calls (atributo de unittest.mock.Mock), 1610
mock_open() (en el módulo unittest.mock), 1636
Mod (clase en ast), 1896
mod() (en el módulo operator), 407
mode (atributo de io.FileIO), 664
mode (atributo de ossaudiodev.oss_audio_device), 2058
mode (atributo de statistics.NormalDist), 377
mode (atributo de tarfile.TarInfo), 546
--mode <mode>
astopción de línea de comando, 1921
mode() (en el módulo statistics), 372
mode() (en el módulo turtle), 1435
modes
file, 18
modf() (en el módulo math), 323
modified() (método de urllib.robotparser.RobotFileParser), 1298
Modify() (método de msilib.View), 2017
modify() (método de select.devpoll), 1070
modify() (método de select.epoll), 1071
modify() (método de selectors.BaseSelector), 1076
modify() (método de select.poll), 1072
module
search path, 453, 1761, 1844
module (atributo de pyclbr.Class), 1933
module (atributo de pyclbr.Function), 1933
Module browser, 1498
module_for_loader() (en el módulo importlib.util), 1880
module_from_spec() (en el módulo importlib.util), 1880
module_repr() (método de importlib.abc.Loader), 1867
modulefinder
módulo, 1859
ModuleFinder (clase en modulefinder), 1859
ModuleInfo (clase en pkgutil), 1855
ModuleNotFoundError, 103
modules (atributo de modulefinder.ModuleFinder), 1859
modules (en el módulo sys), 1761
modules_cleanup() (en el módulo test.support.import_helper), 1682
modules_setup() (en el módulo test.support.import_helper), 1682
ModuleSpec (clase en importlib.machinery), 1877
ModuleType (clase en types), 282
módulo, **2087**
__future__, 1822
__main__, 1773, 1860, 1861
_locale, 1400
_thread, 916
abc, 1810
aifc, 1985
argparse, 680
array, 59, 270

ast, 1889
asynchat, 1987
asyncio, 919
asyncore, 1990
atexit, 1814
audioop, 1994
base64, 1182, 1186
bdb, 1689, 1696
binascii, 1186
binhex, 1185, 1186
bisect, 266
builtins, 1773
bz2, 520
calendar, 234
cgi, 1997
cgitb, 2004
chunk, 2005
cmath, 328
cmd, 1443, 1696
code, 1849
codecs, 174
codeop, 1851
collections, 239
collections.abc, 257
colorsys, 1389
compileall, 1936
concurrent.futures, 882
configparser, 562
contextlib, 1796
contextvars, 912
copy, 286, 481
copyreg, 481
cProfile, 1705
crypt, 1972, 2006
csv, 555
ctypes, 788
curses, 756
curses.ascii, 775
curses.panel, 778
curses.textpad, 774
dataclasses, 1785
datetime, 193
dbm, 486
dbm.dumb, 490
dbm.gnu, 483, 488
dbm.ndbm, 483, 489
decimal, 331
difflib, 144
dis, 1940
distutils, 1731
doctest, 1548
email, 1093
email.charset, 1145
email.contentmanager, 1123
email.encoders, 1147
email.errors, 1116
email.generator, 1106
email.header, 1142
email.headerregistry, 1117
email.iterators, 1150
email.message, 1094
email.mime, 1140
email.parser, 1102
email.policy, 1109
email.utils, 1148
encodings.idna, 190
encodings.mbcs, 190
encodings.utf_8_sig, 191
ensurepip, 1732
enum, 294
errno, 104, 782
faulthandler, 1694
fcntl, 1977
filecmp, 444
fileinput, 437
fnmatch, 452
fractions, 358
ftplib, 1308
functools, 396
gc, 1824
getopt, 711
getpass, 755
gettext, 1391
glob, 451, 452
graphlib, 313
grp, 1973
gzip, 517
hashlib, 585
heapq, 262
hmac, 595
html, 1191
html.entities, 1196
html.parser, 1192
http, 1299
http.client, 1301
http.cookiejar, 1350
http.cookies, 1347
http.server, 1341
idlelib, 1510
imaplib, 1316
imghdr, 2008
imp, 28, 2009
importlib, 1862
importlib.abc, 1865
importlib.machinery, 1874
importlib.metadata, 1884
importlib.resources, 1872
importlib.util, 1878
inspect, 1827
io, 656
ipaddress, 1372
itertools, 381
json, 1152
json.tool, 1160
keyword, 1927
lib2to3, 1665

linecache, 453
locale, 1400
logging, 714
logging.config, 730
logging.handlers, 742
lzma, 524
mailbox, 1161
mailcap, 2014
marshal, 485
math, 36, 320, 331
mimetypes, 1179
mmap, 1087
modulefinder, 1859
msilib, 2015
msvcrt, 1957
multiprocessing, 834
multiprocessing.connection, 864
multiprocessing.dummy, 868
multiprocessing.managers, 855
multiprocessing.pool, 861
multiprocessing.shared_memory, 877
multiprocessing.sharedctypes, 853
netrc, 580
nis, 2021
nntplib, 2022
numbers, 317
operator, 406
optparse, 2028
os, 601, 1971
os.path, 431
ossaudiodev, 2055
pathlib, 413
pdb, 1696
pickle, 286, 465, 481, 482, 485
pickletools, 1953
pipes, 2060
pkgutil, 1855
platform, 779
plistlib, 581
poplib, 1313
posix, 1971
pprint, 287
profile, 1705
pstats, 1707
pty, 613, 1975
pwd, 433, 1972
py_compile, 1934
pyclbr, 1932
pydoc, 1544
pyexpat, 1247
queue, 909
quopri, 1188
random, 360
re, 49, 124, 452
readline, 162
reprlib, 292
resource, 1979
rlcompleter, 166
runpy, 1860
sched, 908
secrets, 597
select, 1068
selectors, 1075
shelve, 482, 485
shlex, 1448
shutil, 454
signal, 918, 1078
site, 1844
sitecustomize, 1845
smtpd, 2061
smtplib, 1323
sndhdr, 2064
socket, 1008, 1257
socketserver, 1333
spwd, 2065
sqlite3, 491
ssl, 1033
stat, 439, 630
statistics, 368
string, 113, 1405
stringprep, 160
struct, 167, 1028
subprocess, 889
sunau, 2066
symtable, 1922
sys, 20, 1749
sysconfig, 1769
syslog, 1983
tabnanny, 1932
tarfile, 539
telnetlib, 2068
tempfile, 446
termios, 1974
test, 1666
test.support, 1668
test.support.bytecode_helper, 1679
test.support.import_helper, 1682
test.support.os_helper, 1680
test.support.script_helper, 1677
test.support.socket_helper, 1677
test.support.threading_helper, 1679
test.support.warnings_helper, 1683
textwrap, 155
threading, 821
time, 670
timeit, 1711
tkinter, 1455
tkinter.colorchooser, 1468
tkinter.commondialog, 1472
tkinter.dnd, 1473
tkinter.filedialog, 1470
tkinter.font, 1468
tkinter.messagebox, 1472
tkinter.scrolledtext, 1473
tkinter.simpledialog, 1469
tkinter.tix, 1493

tkinter.ttk, 1474
token, 1924
tokenize, 1928
trace, 1716
traceback, 1816
tracemalloc, 1719
tty, 1975
turtle, 1409
turtledemo, 1441
types, 96, 280
typing, 1511
unicodedata, 158
unittest, 1571
unittest.mock, 1601
urllib, 1270
urllib.error, 1297
urllib.parse, 1288
urllib.request, 1270, 1301
urllib.response, 1288
urllib.robotparser, 1298
usercustomize, 1845
uu, 1186, 2071
uuid, 1329
venv, 1733
warnings, 1779
wave, 1387
weakref, 272
webbrowser, 1257
winreg, 1959
winsound, 1968
wsgiref, 1260
wsgiref.handlers, 1265
wsgiref.headers, 1262
wsgiref.simple_server, 1263
wsgiref.util, 1260
wsgiref.validate, 1264
xdrlib, 2072
xml, 1197
xml.dom, 1218
xml.dom.minidom, 1228
xml.dom.pulldom, 1233
xml.etree.ElementTree, 1198
xml.parsers.expat, 1247
xml.parsers.expat.errors, 1254
xml.parsers.expat.model, 1253
xmlrpc.client, 1359
xmlrpc.server, 1366
xml.sax, 1235
xml.sax.handler, 1236
xml.sax.saxutils, 1242
xml.sax.xmlreader, 1243
zipapp, 1742
zipfile, 529
zipimport, 1853
zlib, 513
zoneinfo, 229
módulo de extensión, **2083**
MONDAY (*en el módulo calendar*), 238
monotonic() (*en el módulo time*), 673
monotonic_ns() (*en el módulo time*), 673
month (*atributo de datetime.date*), 200
month (*atributo de datetime.datetime*), 206
month() (*en el módulo calendar*), 238
month_abbr (*en el módulo calendar*), 238
month_name (*en el módulo calendar*), 238
monthcalendar() (*en el módulo calendar*), 238
monthdatescalendar() (*método de calendar.Calendar*), 235
monthdays2calendar() (*método de calendar.Calendar*), 235
monthdayscalendar() (*método de calendar.Calendar*), 235
monthrange() (*en el módulo calendar*), 238
Morsel (*clase en http.cookies*), 1348
most_common() (*método de collections.Counter*), 242
mouseinterval() (*en el módulo curses*), 760
mousemask() (*en el módulo curses*), 760
move() (*en el módulo shutil*), 458
move() (*método de curses.panel.Panel*), 778
move() (*método de curses.window*), 767
move() (*método de mmap.mmap*), 1089
move() (*método de tkinter.ttk.Treeview*), 1488
move_to_end() (*método de collections.OrderedDict*), 253
MozillaCookieJar (*clase en http.cookiejar*), 1354
MRO, **2087**
mro() (*método de class*), 97
msg (*atributo de http.client.HTTPResponse*), 1306
msg (*atributo de json.JSONDecodeError*), 1158
msg (*atributo de re.error*), 134
msg (*atributo de traceback.TracebackException*), 1819
msg() (*método de telnetlib.Telnet*), 2070
msi, 2015
msilib
    módulo, 2015
msvcrt
    módulo, 1957
mt_interact() (*método de telnetlib.Telnet*), 2070
mtime (*atributo de gzip.GzipFile*), 518
mtime (*atributo de tarfile.TarInfo*), 546
mtime() (*método de urllib.robotparser.RobotFileParser*), 1298
mul() (*en el módulo audioop*), 1996
mul() (*en el módulo operator*), 407
Mult (*clase en ast*), 1896
MultiCall (*clase en xmlrpc.client*), 1364
MULTILINE (*en el módulo re*), 130
MultiLoopChildWatcher (*clase en asyncio*), 994
multimode() (*en el módulo statistics*), 373
MultipartConversionError, 1116
multiply() (*método de decimal.Context*), 346
multiprocessing
    módulo, 834
multiprocessing.connection
    módulo, 864
multiprocessing.dummy

módulo, 868
multiprocessing.Manager()
función incorporada, 855
multiprocessing.managers
módulo, 855
multiprocessing.pool
módulo, 861
multiprocessing.shared_memory
módulo, 877
multiprocessing.sharedctypes
módulo, 853
mutable, **2087**
sequence types, 44
mutable sequence
loop over, 42
MutableMapping (*clase en collections.abc*), 260
MutableMapping (*clase en typing*), 1537
MutableSequence (*clase en collections.abc*), 260
MutableSequence (*clase en typing*), 1537
MutableSet (*clase en collections.abc*), 260
MutableSet (*clase en typing*), 1537
mvderwin() (*método de curses.window*), 767
mvwin() (*método de curses.window*), 767
myrights() (*método de imaplib.IMAP4*), 1320

N

-n N
timeitopción de línea de comando, 1714
N_TOKENS (*en el módulo token*), 1927
n_waiting (*atributo de threading.Barrier*), 834
name (*atributo de codecs.CodecInfo*), 174
name (*atributo de contextvars.ContextVar*), 913
name (*atributo de doctest.DocTest*), 1563
name (*atributo de email.headerregistry.BaseHeader*), 1118
name (*atributo de gzip.GzipFile*), 518
name (*atributo de hashlib.hash*), 587
name (*atributo de hmac.HMAC*), 596
name (*atributo de http.cookiejar.Cookie*), 1357
name (*atributo de importlib.abc.FileLoader*), 1870
name (*atributo de importlib.machinery.ExtensionFileLoader*), 1877
name (*atributo de importlib.machinery.ModuleSpec*), 1878
name (*atributo de importlib.machinery.SourceFileLoader*), 1876
name (*atributo de importlib.machinery.SourcelessFileLoader*), 1876
name (*atributo de inspect.Parameter*), 1834
name (*atributo de io.FileIO*), 664
name (*atributo de multiprocessing.Process*), 842
name (*atributo de multiprocessing.shared_memory.SharedMemory*), 878
name (*atributo de os.DirEntry*), 629
name (*atributo de ossaudiodev.oss_audio_device*), 2058
name (*atributo de pathlib.PurePath*), 419
name (*atributo de pyclbr.Class*), 1933
name (*atributo de pyclbr.Function*), 1933
name (*atributo de tarfile.TarInfo*), 546
name (*atributo de threading.Thread*), 825
name (*atributo de xml.dom.Attr*), 1225
name (*atributo de xml.dom.DocumentType*), 1222
name (*atributo de zipfile.Path*), 534
Name (*clase en ast*), 1895
name (*en el módulo os*), 601
NAME (*en el módulo token*), 1924
name() (*en el módulo unicodedata*), 159
name() (*método de importlib.abc.Traversable*), 1871
name2codepoint (*en el módulo html.entities*), 1196
Named Shared Memory, 877
NamedExpr (*clase en ast*), 1898
NamedTemporaryFile() (*en el módulo tempfile*), 447
NamedTuple (*clase en typing*), 1531
namedtuple() (*en el módulo collections*), 250
NameError, 104
namelist() (*método de zipfile.ZipFile*), 531
nameprep() (*en el módulo encodings.idna*), 190
namer (*atributo de logging.handlers.BaseRotatingHandler*), 744
namereplace
error handler's name, 177
namereplace_errors() (*en el módulo codecs*), 178
names() (*en el módulo tkinter.font*), 1469
Namespace (*clase en argparse*), 703
Namespace (*clase en multiprocessing.managers*), 857
namespace() (*método de imaplib.IMAP4*), 1320
Namespace() (*método de multiprocessing.managers.SyncManager*), 857
NAMESPACE_DNS (*en el módulo uuid*), 1332
NAMESPACE_OID (*en el módulo uuid*), 1332
NAMESPACE_URL (*en el módulo uuid*), 1332
NAMESPACE_X500 (*en el módulo uuid*), 1332
NamespaceErr, 1226
namespaceURI (*atributo de xml.dom.Node*), 1221
nametofont() (*en el módulo tkinter.font*), 1469
NaN, 13
nan (*en el módulo cmath*), 331
nan (*en el módulo math*), 327
nanj (*en el módulo cmath*), 331
NannyNag, 1932
napms() (*en el módulo curses*), 760
nargs (*atributo de optparse.Option*), 2042
native_id (*atributo de threading.Thread*), 825
nbytes (*atributo de memoryview*), 79
ncurses_version (*en el módulo curses*), 770
ndiff() (*en el módulo difflib*), 147
ndim (*atributo de memoryview*), 80
ne (*2to3 fixer*), 1664
ne() (*en el módulo operator*), 406
needs_input (*atributo de bz2.BZ2Decompressor*), 522

needs_input (*atributo de lzma.LZMADecompressor*), 527
neg() (*en el módulo operator*), 407
netmask (*atributo de ipaddress.IPv4Network*), 1379
netmask (*atributo de ipaddress.IPv6Network*), 1381
NetmaskValueError, 1385
netrc
    módulo, 580
netrc (*clase en netrc*), 580
NetrcParseError, 580
netscape (*atributo de http.cookiejar.CookiePolicy*), 1355
network (*atributo de ipaddress.IPv4Interface*), 1383
network (*atributo de ipaddress.IPv6Interface*), 1384
Network News Transfer Protocol, 2022
network_address (*atributo de ipaddress.IPv4Network*), 1378
network_address (*atributo de ipaddress.IPv6Network*), 1381
NEVER_EQ (*en el módulo test.support*), 1670
new() (*en el módulo hashlib*), 586
new() (*en el módulo hmac*), 595
new_child() (*método de collections.ChainMap*), 239
new_class() (*en el módulo types*), 280
new_event_loop() (*en el módulo asyncio*), 954
new_event_loop() (*método de asyncio.AbstractEventLoopPolicy*), 992
new_module() (*en el módulo imp*), 2010
new_panel() (*en el módulo curses.panel*), 778
newgroups() (*método de nntplib.NNTP*), 2025
NEWLINE (*en el módulo token*), 1924
newlines (*atributo de io.TextIOBase*), 666
newnews() (*método de nntplib.NNTP*), 2025
newpad() (*en el módulo curses*), 760
NewType (*clase en typing*), 1532
newwin() (*en el módulo curses*), 760
next (*2to3 fixer*), 1664
next (*pdb command*), 1701
next()
    función incorporada, 17
next() (*método de nntplib.NNTP*), 2027
next() (*método de tarfile.TarFile*), 543
next() (*método de tkinter.ttk.Treeview*), 1488
next_minus() (*método de decimal.Context*), 346
next_minus() (*método de decimal.Decimal*), 340
next_plus() (*método de decimal.Context*), 346
next_plus() (*método de decimal.Decimal*), 340
next_toward() (*método de decimal.Context*), 347
next_toward() (*método de decimal.Decimal*), 340
nextafter() (*en el módulo math*), 323
nextfile() (*en el módulo fileinput*), 438
nextkey() (*método de dbm.gnu.gdbm*), 488
nextSibling (*atributo de xml.dom.Node*), 1221
ngettext() (*en el módulo gettext*), 1392
ngettext() (*método de gettext.GNUTranslations*), 1396
ngettext() (*método de gettext.NullTranslations*), 1394
nice() (*en el módulo os*), 644
nis
    módulo, 2021
NL (*en el módulo token*), 1927
nl() (*en el módulo curses*), 760
nl_langinfo() (*en el módulo locale*), 1402
nlargest() (*en el módulo heapq*), 263
nlst() (*método de ftplib.FTP*), 1312
NNTP
    protocol, 2022
NNTP (*clase en nntplib*), 2022
nntp_implementation (*atributo de nntplib.NNTP*), 2024
NNTP_SSL (*clase en nntplib*), 2023
nntp_version (*atributo de nntplib.NNTP*), 2024
NNTPDataError, 2023
NNTPError, 2023
nntplib
    módulo, 2022
NNTPPermanentError, 2023
NNTPProtocolError, 2023
NNTPReplyError, 2023
NNTPTemporaryError, 2023
no_cache() (*método de clase de zoneinfo.ZoneInfo*), 231
no_proxy, 1273
no_tracing() (*en el módulo test.support*), 1674
no_type_check() (*en el módulo typing*), 1541
no_type_check_decorator() (*en el módulo typing*), 1541
nocbreak() (*en el módulo curses*), 760
NoDataAllowedErr, 1226
node() (*en el módulo platform*), 779
nodelay() (*método de curses.window*), 768
nodeName (*atributo de xml.dom.Node*), 1221
NodeTransformer (*clase en ast*), 1920
nodeType (*atributo de xml.dom.Node*), 1220
nodeValue (*atributo de xml.dom.Node*), 1221
NodeVisitor (*clase en ast*), 1919
noecho() (*en el módulo curses*), 760
--no-ensure-ascii
    json.toolopción de línea de comando, 1161
NOEXPR (*en el módulo locale*), 1403
--no-indent
    json.toolopción de línea de comando, 1161
nombre calificado, **2090**
NoModificationAllowedErr, 1227
nonblock() (*método de ossaudiodev.oss_audio_device*), 2057
NonCallableMagicMock (*clase en unittest.mock*), 1630
NonCallableMock (*clase en unittest.mock*), 1611
None (*Built-in object*), 33
None (*variable incorporada*), 31
NoneType (*en el módulo types*), 281
nonl() (*en el módulo curses*), 760

Nonlocal (*clase en ast*), 1915
nonzero (*2to3 fixer*), 1664
noop() (*método de imaplib.IMAP4*), 1320
noop() (*método de poplib.POP3*), 1315
NoOptionError, 580
NOP (*opcode*), 1943
noqiflush() (*en el módulo curses*), 760
noraw() (*en el módulo curses*), 760
--no-report
    traceopción de línea de comando, 1717
NoReturn (*en el módulo typing*), 1520
NORMAL (*en el módulo tkinter.font*), 1468
NORMAL_PRIORITY_CLASS (*en el módulo subprocess*), 901
NormalDist (*clase en statistics*), 377
normalize() (*en el módulo locale*), 1404
normalize() (*en el módulo unicodedata*), 159
normalize() (*método de decimal.Context*), 347
normalize() (*método de decimal.Decimal*), 340
normalize() (*método de xml.dom.Node*), 1222
NORMALIZE_WHITESPACE (*en el módulo doctest*), 1555
normalvariate() (*en el módulo random*), 364
normcase() (*en el módulo os.path*), 434
normpath() (*en el módulo os.path*), 434
NoSectionError, 579
NoSuchMailboxError, 1177
not
    operador, 34
Not (*clase en ast*), 1896
not in
    operador, 34, 42
not_() (*en el módulo operator*), 406
NotADirectoryError, 108
notationDecl() (*método de xml.sax.handler.DTDHandler*), 1241
NotationDeclHandler() (*método de xml.parsers.expat.xmlparser*), 1251
notations (*atributo de xml.dom.DocumentType*), 1223
NotConnected, 1303
NoteBook (*clase en tkinter.tix*), 1496
Notebook (*clase en tkinter.ttk*), 1482
NotEmptyError, 1177
NotEq (*clase en ast*), 1897
NOTEQUAL (*en el módulo token*), 1925
NotFoundErr, 1226
notify() (*método de asyncio.Condition*), 943
notify() (*método de threading.Condition*), 830
notify_all() (*método de asyncio.Condition*), 943
notify_all() (*método de threading.Condition*), 830
notimeout() (*método de curses.window*), 768
NotImplemented (*variable incorporada*), 31
NotImplementedError, 104
NotImplementedType (*en el módulo types*), 282
NotIn (*clase en ast*), 1897
NotStandaloneHandler() (*método de xml.parsers.expat.xmlparser*), 1251
NotSupportedErr, 1226
NotSupportedError, 505
--no-type-comments
    astopción de línea de comando, 1921
noutrefresh() (*método de curses.window*), 768
now() (*método de clase de datetime.datetime*), 203
npgettext() (*en el módulo gettext*), 1392
npgettext() (*método de gettext.GNUTranslations*), 1396
npgettext() (*método de gettext.NullTranslations*), 1394
NSIG (*en el módulo signal*), 1080
nsmallest() (*en el módulo heapq*), 263
NT_OFFSET (*en el módulo token*), 1927
NTEventLogHandler (*clase en logging.handlers*), 751
ntohl() (*en el módulo socket*), 1019
ntohs() (*en el módulo socket*), 1019
ntransfercmd() (*método de ftplib.FTP*), 1312
nullcontext() (*en el módulo contextlib*), 1798
NullHandler (*clase en logging*), 743
NullImporter (*clase en imp*), 2013
NullTranslations (*clase en gettext*), 1394
num_addresses (*atributo de ipaddress.IPv4Network*), 1379
num_addresses (*atributo de ipaddress.IPv6Network*), 1381
num_tickets (*atributo de ssl.SSLContext*), 1057
Number (*clase en numbers*), 317
NUMBER (*en el módulo token*), 1924
--number=N
    timeitopción de línea de comando, 1714
number_class() (*método de decimal.Context*), 347
number_class() (*método de decimal.Decimal*), 340
numbers
    módulo, 317
numerator (*atributo de fractions.Fraction*), 359
numerator (*atributo de numbers.Rational*), 318
numeric
    conversions, 36
    literals, 35
    object, 34
    objeto, 35
    types, operations on, 35
numeric() (*en el módulo unicodedata*), 159
número complejo, **2081**
numinput() (*en el módulo turtle*), 1434
numliterals (*2to3 fixer*), 1664

O

-o
    pickletoolsopción de línea de comando, 1954
-o <output>

zipappopción de línea de comando, 1743
-o level
compileallopción de línea de comando, 1937
O_APPEND (*en el módulo os*), 612
O_ASYNC (*en el módulo os*), 613
O_BINARY (*en el módulo os*), 613
O_CLOEXEC (*en el módulo os*), 612
O_CREAT (*en el módulo os*), 612
O_DIRECT (*en el módulo os*), 613
O_DIRECTORY (*en el módulo os*), 613
O_DSYNC (*en el módulo os*), 612
O_EVTONLY (*en el módulo os*), 613
O_EXCL (*en el módulo os*), 612
O_EXLOCK (*en el módulo os*), 613
O_FSYNC (*en el módulo os*), 613
O_NDELAY (*en el módulo os*), 612
O_NOATIME (*en el módulo os*), 613
O_NOCTTY (*en el módulo os*), 612
O_NOFOLLOW (*en el módulo os*), 613
O_NOFOLLOW_ANY (*en el módulo os*), 613
O_NOINHERIT (*en el módulo os*), 613
O_NONBLOCK (*en el módulo os*), 612
O_PATH (*en el módulo os*), 613
O_RANDOM (*en el módulo os*), 613
O_RDONLY (*en el módulo os*), 612
O_RDWR (*en el módulo os*), 612
O_RSYNC (*en el módulo os*), 612
O_SEQUENTIAL (*en el módulo os*), 613
O_SHLOCK (*en el módulo os*), 613
O_SHORT_LIVED (*en el módulo os*), 613
O_SYMLINK (*en el módulo os*), 613
O_SYNC (*en el módulo os*), 612
O_TEMPORARY (*en el módulo os*), 613
O_TEXT (*en el módulo os*), 613
O_TMPFILE (*en el módulo os*), 613
O_TRUNC (*en el módulo os*), 612
O_WRONLY (*en el módulo os*), 612
obj (*atributo de memoryview*), 79
object
code, 95, 485
numeric, 34
object (*atributo de UnicodeError*), 107
object (*clase incorporada*), 18
objects
comparing, 34
flattening, 465
marshalling, 465
persistent, 465
pickling, 465
serializing, 465
objeto, **2088**
Boolean, 35
bytearray, 44, 59, 61
bytes, 59
complex number, 35
dictionary, 83
floating point, 35
GenericAlias, 89
integer, 35
io.StringIO, 48
list, 44, 45
mapping, 83
memoryview, 59
method, 95
numeric, 35
range, 46
sequence, 42
set, 81
socket, 1008
string, 48
traceback, 1753, 1816
tuple, 44, 46
type, 26
Union, 93
objeto archivo, **2083**
objeto tipo ruta, **2089**
objetos tipo archivo, **2083**
objetos tipo binarios, **2081**
obufcount() (*método de ossaudiodev.oss_audio_device*), 2058
obuffree() (*método de ossaudiodev.oss_audio_device*), 2058
oct()
función incorporada, 18
octal
literals, 35
octdigits (*en el módulo string*), 113
offset (*atributo de SyntaxError*), 106
offset (*atributo de traceback.TracebackException*), 1818
offset (*atributo de xml.parsers.expat.ExpatError*), 1252
OK (*en el módulo curses*), 770
ok_command() (*método de tkinter.filedialog.LoadFileDialog*), 1472
ok_command() (*método de tkinter.filedialog.SaveFileDialog*), 1472
ok_event() (*método de tkinter.filedialog.FileDialog*), 1471
old_value (*atributo de contextvars.Token*), 913
OleDLL (*clase en ctypes*), 807
on_motion() (*método de tkinter.dnd.DndHandler*), 1474
on_release() (*método de tkinter.dnd.DndHandler*), 1474
onclick() (*en el módulo turtle*), 1433
ondrag() (*en el módulo turtle*), 1428
onecmd() (*método de cmd.Cmd*), 1444
onkey() (*en el módulo turtle*), 1432
onkeypress() (*en el módulo turtle*), 1433
onkeyrelease() (*en el módulo turtle*), 1432
onrelease() (*en el módulo turtle*), 1428
onscreenclick() (*en el módulo turtle*), 1433
ontimer() (*en el módulo turtle*), 1433

OP (*en el módulo token*), 1926
OP_ALL (*en el módulo ssl*), 1042
OP_CIPHER_SERVER_PREFERENCE (*en el módulo ssl*), 1043
OP_ENABLE_MIDDLEBOX_COMPAT (*en el módulo ssl*), 1043
OP_IGNORE_UNEXPECTED_EOF (*en el módulo ssl*), 1044
OP_NO_COMPRESSION (*en el módulo ssl*), 1044
OP_NO_RENEGOTIATION (*en el módulo ssl*), 1043
OP_NO_SSLv2 (*en el módulo ssl*), 1042
OP_NO_SSLv3 (*en el módulo ssl*), 1042
OP_NO_TICKET (*en el módulo ssl*), 1044
OP_NO_TLSv1 (*en el módulo ssl*), 1043
OP_NO_TLSv1_1 (*en el módulo ssl*), 1043
OP_NO_TLSv1_2 (*en el módulo ssl*), 1043
OP_NO_TLSv1_3 (*en el módulo ssl*), 1043
OP_SINGLE_DH_USE (*en el módulo ssl*), 1043
OP_SINGLE_ECDH_USE (*en el módulo ssl*), 1043
Open (*clase en tkinter.filedialog*), 1471
open()
    función incorporada, 18
open() (*en el módulo aifc*), 1985
open() (*en el módulo bz2*), 520
open() (*en el módulo codecs*), 175
open() (*en el módulo dbm*), 486
open() (*en el módulo dbm.dumb*), 490
open() (*en el módulo dbm.gnu*), 488
open() (*en el módulo dbm.ndbm*), 489
open() (*en el módulo gzip*), 517
open() (*en el módulo io*), 658
open() (*en el módulo lzma*), 524
open() (*en el módulo os*), 612
open() (*en el módulo ossaudiodev*), 2056
open() (*en el módulo shelve*), 482
open() (*en el módulo sunau*), 2066
open() (*en el módulo tarfile*), 539
open() (*en el módulo tokenize*), 1929
open() (*en el módulo wave*), 1387
open() (*en el módulo webbrowser*), 1258
open() (*método de clase de tarfile.TarFile*), 543
open() (*método de imaplib.IMAP4*), 1320
open() (*método de importlib.abc.Traversable*), 1871
open() (*método de pathlib.Path*), 427
open() (*método de pipes.Template*), 2061
open() (*método de telnetlib.Telnet*), 2070
open() (*método de urllib.request.OpenerDirector*), 1277
open() (*método de urllib.request.URLopener*), 1286
open() (*método de webbrowser.controller*), 1260
open() (*método de zipfile.Path*), 534
open() (*método de zipfile.ZipFile*), 531
open_binary() (*en el módulo importlib.resources*), 1873
open_code() (*en el módulo io*), 658
open_connection() (*en el módulo asyncio*), 935
open_new() (*en el módulo webbrowser*), 1258
open_new() (*método de webbrowser.controller*), 1260
open_new_tab() (*en el módulo webbrowser*), 1258
open_new_tab() (*método de webbrowser.controller*), 1260
open_osfhandle() (*en el módulo msvcrt*), 1958
open_resource() (*método de importlib.abc.ResourceReader*), 1868
open_text() (*en el módulo importlib.resources*), 1873
open_unix_connection() (*en el módulo asyncio*), 936
open_unknown() (*método de urllib.request.URLopener*), 1286
open_urlresource() (*en el módulo test.support*), 1674
OpenDatabase() (*en el módulo msilib*), 2015
OpenerDirector (*clase en urllib.request*), 1273
OpenKey() (*en el módulo winreg*), 1962
OpenKeyEx() (*en el módulo winreg*), 1962
openlog() (*en el módulo syslog*), 1983
openmixer() (*en el módulo ossaudiodev*), 2056
openpty() (*en el módulo os*), 613
openpty() (*en el módulo pty*), 1975
OpenSSL
    (use in module hashlib), 586
    (use in module ssl), 1033
OPENSSL_VERSION (*en el módulo ssl*), 1045
OPENSSL_VERSION_INFO (*en el módulo ssl*), 1045
OPENSSL_VERSION_NUMBER (*en el módulo ssl*), 1045
OpenView() (*método de msilib.Database*), 2016
operador
    % (*percent*), 35
    & (*ampersand*), 36
    * (*asterisk*), 35
    **, 35
    / (*slash*), 35
    //, 35
    < (*less*), 34
    <<, 36
    <=, 34
    !=, 34
    ==, 34
    > (*greater*), 34
    >=, 34
    >>, 36
    ^ (*caret*), 36
    | (*vertical bar*), 36
    ~ (*tilde*), 36
    and, 33, 34
    in, 34, 42
    is, 34
    is not, 34
    not, 34
    not in, 34, 42
    or, 33, 34
operation
    concatenation, 42
    repetition, 42
    slice, 42

subscript, 42
OperationalError, 504
operations
bitwise, 36
Boolean, 33, 34
masking, 36
shifting, 36
operations on
dictionary type, 83
integer types, 36
list type, 44
mapping types, 83
numeric types, 35
sequence types, 42, 44
operator
- (*minus*), 35
+ (*plus*), 35
comparison, 34
módulo, 406
operator (*2to3 fixer*), 1664
opmap (*en el módulo dis*), 1953
opname (*en el módulo dis*), 1953
optim_args_from_interpreter_flags() (*en el módulo test.support*), 1671
optimize() (*en el módulo pickletools*), 1954
OPTIMIZED_BYTECODE_SUFFIXES (*en el módulo importlib.machinery*), 1874
Optional (*en el módulo typing*), 1521
OptionGroup (*clase en optparse*), 2036
OptionMenu (*clase en tkinter.tix*), 1494
OptionParser (*clase en optparse*), 2039
options (*atributo de doctest.Example*), 1564
options (*atributo de ssl.SSLContext*), 1057
Options (*clase en ssl*), 1044
options() (*método de configparser.ConfigParser*), 576
optionxform() (*método de configparser.ConfigParser*), 578
optparse
módulo, 2028
or
operador, 33, 34
Or (*clase en ast*), 1897
or_() (*en el módulo operator*), 407
ord()
función incorporada, 21
orden de resolución de métodos, **2087**
ordered_attributes (*atributo de xml.parsers.expat.xmlparser*), 1249
OrderedDict (*clase en collections*), 253
OrderedDict (*clase en typing*), 1535
orig_argv (*en el módulo sys*), 1761
origin (*atributo de importlib.machinery.ModuleSpec*), 1878
origin_req_host (*atributo de urllib.request.Request*), 1275
origin_server (*atributo de wsgiref.handlers.BaseHandler*), 1268
os
módulo, 601, 1971
os_environ (*atributo de wsgiref.handlers.BaseHandler*), 1267
OSError, 104
os.path
módulo, 431
ossaudiodev
módulo, 2055
OSSAudioError, 2055
outfile
json.toolopción de línea de comando, 1161
output (*atributo de subprocess.CalledProcessError*), 891
output (*atributo de subprocess.TimeoutExpired*), 891
output (*atributo de unittest.TestCase*), 1585
output() (*método de http.cookies.BaseCookie*), 1347
output() (*método de http.cookies.Morsel*), 1349
--output=<file>
pickletoolsopción de línea de comando, 1954
--output=<output>
zipappopción de línea de comando, 1743
output_charset (*atributo de email.charset.Charset*), 1146
output_charset() (*método de gettext.NullTranslations*), 1395
output_codec (*atributo de email.charset.Charset*), 1146
output_difference() (*método de doctest.OutputChecker*), 1567
OutputChecker (*clase en doctest*), 1567
OutputString() (*método de http.cookies.Morsel*), 1349
OutsideDestinationError, 541
over() (*método de nntplib.NNTP*), 2026
Overflow (*clase en decimal*), 350
OverflowError, 105
overlap() (*método de statistics.NormalDist*), 378
overlaps() (*método de ipaddress.IPv4Network*), 1379
overlaps() (*método de ipaddress.IPv6Network*), 1381
overlay() (*método de curses.window*), 768
overload() (*en el módulo typing*), 1541
overwrite() (*método de curses.window*), 768
owner() (*método de pathlib.Path*), 427

P

-p
pickletoolsopción de línea de comando, 1954
timeitopción de línea de comando, 1714
unittest-discoveropción de línea de comando, 1575

p (*pdb command*), 1701
-p <interpreter>
    zipappopción de línea de comando, 1743
-p prepend_prefix
    compileallopción de línea de comando, 1936
P_ALL (*en el módulo os*), 649
P_DETACH (*en el módulo os*), 648
P_NOWAIT (*en el módulo os*), 647
P_NOWAITO (*en el módulo os*), 647
P_OVERLAY (*en el módulo os*), 648
P_PGID (*en el módulo os*), 649
P_PID (*en el módulo os*), 649
P_PIDFD (*en el módulo os*), 650
P_WAIT (*en el módulo os*), 647
pack() (*en el módulo struct*), 168
pack() (*método de mailbox.MH*), 1167
pack() (*método de struct.Struct*), 173
pack_array() (*método de xdrlib.Packer*), 2073
pack_bytes() (*método de xdrlib.Packer*), 2073
pack_double() (*método de xdrlib.Packer*), 2073
pack_farray() (*método de xdrlib.Packer*), 2073
pack_float() (*método de xdrlib.Packer*), 2072
pack_fopaque() (*método de xdrlib.Packer*), 2073
pack_fstring() (*método de xdrlib.Packer*), 2073
pack_into() (*en el módulo struct*), 168
pack_into() (*método de struct.Struct*), 173
pack_list() (*método de xdrlib.Packer*), 2073
pack_opaque() (*método de xdrlib.Packer*), 2073
pack_string() (*método de xdrlib.Packer*), 2073
package, 1844
Package (*en el módulo importlib.resources*), 1872
packed (*atributo de ipaddress.IPv4Address*), 1374
packed (*atributo de ipaddress.IPv6Address*), 1375
Packer (*clase en xdrlib*), 2072
packing
    binary data, 167
packing (*widgets*), 1463
PAGER, 1544
pair_content() (*en el módulo curses*), 761
pair_number() (*en el módulo curses*), 761
pairwise() (*en el módulo itertools*), 388
PanedWindow (*clase en tkinter.tix*), 1496
paquete, **2088**
paquete de espacios de nombres, **2088**
paquete provisorio, **2089**
paquete regular, **2090**
Parameter (*clase en inspect*), 1834
ParameterizedMIMEHeader (*clase en email.headerregistry*), 1120
parameters (*atributo de inspect.Signature*), 1833
parámetro, **2088**
params (*atributo de email.headerregistry.ParameterizedMIMEHeader*), 1120
ParamSpec (*clase en typing*), 1528
ParamSpecArgs (*en el módulo typing*), 1529
ParamSpecKwargs (*en el módulo typing*), 1529
paramstyle (*en el módulo sqlite3*), 495
pardir (*en el módulo os*), 654
paren (*2to3 fixer*), 1664
parent (*atributo de importlib.machinery.ModuleSpec*), 1878
parent (*atributo de pathlib.PurePath*), 419
parent (*atributo de pyclbr.Class*), 1933
parent (*atributo de pyclbr.Function*), 1933
parent (*atributo de urllib.request.BaseHandler*), 1277
parent() (*método de tkinter.ttk.Treeview*), 1488
parent_process() (*en el módulo multiprocessing*), 847
parentNode (*atributo de xml.dom.Node*), 1220
parents (*atributo de collections.ChainMap*), 240
parents (*atributo de pathlib.PurePath*), 419
paretovariate() (*en el módulo random*), 364
parse() (*en el módulo ast*), 1917
parse() (*en el módulo cgi*), 2001
parse() (*en el módulo xml.dom.minidom*), 1228
parse() (*en el módulo xml.dom.pulldom*), 1234
parse() (*en el módulo xml.etree.ElementTree*), 1207
parse() (*en el módulo xml.sax*), 1235
parse() (*método de doctest.DocTestParser*), 1565
parse() (*método de email.parser.BytesParser*), 1104
parse() (*método de email.parser.Parser*), 1104
parse() (*método de string.Formatter*), 114
parse() (*método de urllib.robotparser.RobotFileParser*), 1298
parse() (*método de xml.etree.ElementTree.ElementTree*), 1213
Parse() (*método de xml.parsers.expat.xmlparser*), 1248
parse() (*método de xml.sax.xmlreader.XMLReader*), 1244
parse_and_bind() (*en el módulo readline*), 162
parse_args() (*método de argparse.ArgumentParser*), 700
PARSE_COLNAMES (*en el módulo sqlite3*), 495
parse_config_h() (*en el módulo sysconfig*), 1772
PARSE_DECLTYPES (*en el módulo sqlite3*), 495
parse_header() (*en el módulo cgi*), 2001
parse_headers() (*en el módulo http.client*), 1302
parse_intermixed_args() (*método de argparse.ArgumentParser*), 710
parse_known_args() (*método de argparse.ArgumentParser*), 709
parse_known_intermixed_args() (*método de argparse.ArgumentParser*), 710
parse_multipart() (*en el módulo cgi*), 2001
parse_qs() (*en el módulo urllib.parse*), 1290
parse_qsl() (*en el módulo urllib.parse*), 1291
parseaddr() (*en el módulo email.utils*), 1149
parsebytes() (*método de email.parser.BytesParser*), 1104
parsedate() (*en el módulo email.utils*), 1149
parsedate_to_datetime() (*en el módulo email.utils*), 1149

parsedate_tz() (*en el módulo email.utils*), 1149
ParseError (*clase en xml.etree.ElementTree*), 1217
ParseFile() (*método de xml.parsers.expat.xmlparser*), 1248
ParseFlags() (*en el módulo imaplib*), 1317
Parser (*clase en email.parser*), 1104
ParserCreate() (*en el módulo xml.parsers.expat*), 1247
ParseResult (*clase en urllib.parse*), 1295
ParseResultBytes (*clase en urllib.parse*), 1295
parsestr() (*método de email.parser.Parser*), 1104
parseString() (*en el módulo xml.dom.minidom*), 1229
parseString() (*en el módulo xml.dom.pulldom*), 1234
parseString() (*en el módulo xml.sax*), 1235
parsing
    URL, 1288
ParsingError, 580
partial (*atributo de asyncio.IncompleteReadError*), 953
partial() (*en el módulo functools*), 400
partial() (*método de imaplib.IMAP4*), 1320
partialmethod (*clase en functools*), 400
parties (*atributo de threading.Barrier*), 834
partition() (*método de bytearray*), 64
partition() (*método de bytes*), 64
partition() (*método de str*), 53
parts (*atributo de pathlib.PurePath*), 417
Pass (*clase en ast*), 1903
pass_() (*método de poplib.POP3*), 1315
Paste, 1501
patch() (*en el módulo test.support*), 1675
patch() (*en el módulo unittest.mock*), 1620
patch.dict() (*en el módulo unittest.mock*), 1623
patch.multiple() (*en el módulo unittest.mock*), 1625
patch.object() (*en el módulo unittest.mock*), 1623
patch.stopall() (*en el módulo unittest.mock*), 1627
path
    configuration file, 1844
    module search, 453, 1761, 1844
    operations, 413, 431
PATH, 641, 642, 646, 647, 655, 893, 1257, 1844, 2002, 2004
path (*atributo de http.cookiejar.Cookie*), 1357
path (*atributo de http.server.BaseHTTPRequestHandler*), 1342
path (*atributo de importlib.abc.FileLoader*), 1870
path (*atributo de importlib.machinery.ExtensionFileLoader*), 1877
path (*atributo de importlib.machinery.FileFinder*), 1876
path (*atributo de importlib.machinery.SourceFileLoader*), 1876
path (*atributo de importlib.machinery.SourcelessFileLoader*), 1877
path (*atributo de os.DirEntry*), 629
Path (*clase en pathlib*), 423
Path (*clase en zipfile*), 534
path (*en el módulo sys*), 1761
Path browser, 1498
path() (*en el módulo importlib.resources*), 1873
path_hook() (*método de clase de importlib.machinery.FileFinder*), 1876
path_hooks (*en el módulo sys*), 1761
path_importer_cache (*en el módulo sys*), 1762
path_mtime() (*método de importlib.abc.SourceLoader*), 1870
path_return_ok() (*método de http.cookiejar.CookiePolicy*), 1354
path_stats() (*método de importlib.abc.SourceLoader*), 1870
path_stats() (*método de importlib.machinery.SourceFileLoader*), 1876
pathconf() (*en el módulo os*), 625
pathconf_names (*en el módulo os*), 626
PathEntryFinder (*clase en importlib.abc*), 1866
PathFinder (*clase en importlib.machinery*), 1875
pathlib
    módulo, 413
PathLike (*clase en os*), 604
pathname2url() (*en el módulo urllib.request*), 1271
pathsep (*en el módulo os*), 655
pattern (*atributo de re.error*), 134
pattern (*atributo de re.Pattern*), 136
Pattern (*clase en typing*), 1536
--pattern pattern
    unittest-discoveropción de línea de comando, 1575
pause() (*en el módulo signal*), 1082
pause_reading() (*método de asyncio.ReadTransport*), 981
pause_writing() (*método de asyncio.BaseProtocol*), 984
PAX_FORMAT (*en el módulo tarfile*), 541
pax_headers (*atributo de tarfile.TarFile*), 545
pax_headers (*atributo de tarfile.TarInfo*), 547
pbkdf2_hmac() (*en el módulo hashlib*), 588
pd() (*en el módulo turtle*), 1420
pdb
    módulo, 1696
Pdb (*clase en pdb*), 1698
Pdb (*class in pdb*), 1696
.pdbrc
    file, 1699
pdf() (*método de statistics.NormalDist*), 377
peek() (*método de bz2.BZ2File*), 521
peek() (*método de gzip.GzipFile*), 518
peek() (*método de io.BufferedReader*), 665
peek() (*método de lzma.LZMAFile*), 525
peek() (*método de weakref.finalize*), 276
peer (*atributo de smtpd.SMTPChannel*), 2064

PEM_cert_to_DER_cert() (*en el módulo ssl*), 1039
pen() (*en el módulo turtle*), 1420
pencolor() (*en el módulo turtle*), 1422
pending (*atributo de ssl.MemoryBIO*), 1065
pending() (*método de ssl.SSLSocket*), 1050
PendingDeprecationWarning, 109
pendown() (*en el módulo turtle*), 1420
pensize() (*en el módulo turtle*), 1420
penup() (*en el módulo turtle*), 1420
PEP, **2089**
PERCENT (*en el módulo token*), 1925
PERCENTEQUAL (*en el módulo token*), 1926
perf_counter() (*en el módulo time*), 673
perf_counter_ns() (*en el módulo time*), 673
Performance, 1711
perm() (*en el módulo math*), 323
PermissionError, 108
permutations() (*en el módulo itertools*), 388
Persist() (*método de msilib.SummaryInformation*), 2017
persistence, 465
persistent
    objects, 465
persistent_id (*pickle protocol*), 474
persistent_id() (*método de pickle.Pickler*), 469
persistent_load (*pickle protocol*), 474
persistent_load() (*método de pickle.Unpickler*), 470
PF_CAN (*en el módulo socket*), 1013
PF_PACKET (*en el módulo socket*), 1014
PF_RDS (*en el módulo socket*), 1014
pformat() (*en el módulo pprint*), 288
pformat() (*método de pprint.PrettyPrinter*), 289
pgettext() (*en el módulo gettext*), 1392
pgettext() (*método de gettext.GNUTranslations*), 1396
pgettext() (*método de gettext.NullTranslations*), 1394
PGO (*en el módulo test.support*), 1669
phase() (*en el módulo cmath*), 328
pi (*en el módulo cmath*), 330
pi (*en el módulo math*), 327
pi() (*método de xml.etree.ElementTree.TreeBuilder*), 1215
pickle
    módulo, 286, 465, 481, 482, 485
pickle() (*en el módulo copyreg*), 482
PickleBuffer (*clase en pickle*), 470
PickleError, 468
Pickler (*clase en pickle*), 468
pickletools
    módulo, 1953
pickletoolsopción de línea de comando
    -a, 1954
    --annotate, 1954
    --indentlevel=<num>, 1954
    -l, 1954
    -m, 1954
    --memo, 1954
    -o, 1954
    --output=<file>, 1954
    -p, 1954
    --preamble=<preamble>, 1954
pickling
    objects, 465
PicklingError, 468
pid (*atributo de asyncio.subprocess.Process*), 948
pid (*atributo de multiprocessing.Process*), 842
pid (*atributo de subprocess.Popen*), 899
pidfd_open() (*en el módulo os*), 644
pidfd_send_signal() (*en el módulo signal*), 1082
PidfdChildWatcher (*clase en asyncio*), 995
PIPE (*en el módulo subprocess*), 890
Pipe() (*en el módulo multiprocessing*), 844
pipe() (*en el módulo os*), 613
pipe2() (*en el módulo os*), 613
PIPE_BUF (*en el módulo select*), 1069
pipe_connection_lost() (*método de asyncio.SubprocessProtocol*), 986
pipe_data_received() (*método de asyncio.SubprocessProtocol*), 986
PIPE_MAX_SIZE (*en el módulo test.support*), 1669
pipes
    módulo, 2060
PKG_DIRECTORY (*en el módulo imp*), 2013
pkgutil
    módulo, 1855
placeholder (*atributo de textwrap.TextWrapper*), 158
platform
    módulo, 779
platform (*en el módulo sys*), 1762
platform() (*en el módulo platform*), 779
platlibdir (*en el módulo sys*), 1762
PlaySound() (*en el módulo winsound*), 1968
plist
    file, 581
plistlib
    módulo, 581
plock() (*en el módulo os*), 644
PLUS (*en el módulo token*), 1925
plus() (*método de decimal.Context*), 347
PLUSEQUAL (*en el módulo token*), 1926
pm() (*en el módulo pdb*), 1698
pointer() (*en el módulo ctypes*), 814
POINTER() (*en el módulo ctypes*), 814
polar() (*en el módulo cmath*), 328
Policy (*clase en email.policy*), 1110
poll() (*en el módulo select*), 1068
poll() (*método de multiprocessing.connection.Connection*), 849
poll() (*método de select.devpoll*), 1070
poll() (*método de select.epoll*), 1071
poll() (*método de select.poll*), 1072

poll() (*método de subprocess.Popen*), 898
PollSelector (*clase en selectors*), 1077
Pool (*clase en multiprocessing.pool*), 861
pop() (*método de array.array*), 272
pop() (*método de collections.deque*), 245
pop() (*método de dict*), 85
pop() (*método de frozenset*), 83
pop() (*método de mailbox.Mailbox*), 1164
pop() (*sequence method*), 44
POP3
    protocol, 1313
POP3 (*clase en poplib*), 1313
POP3_SSL (*clase en poplib*), 1314
pop_all() (*método de contextlib.ExitStack*), 1803
POP_BLOCK (*opcode*), 1947
POP_EXCEPT (*opcode*), 1947
POP_JUMP_IF_FALSE (*opcode*), 1950
POP_JUMP_IF_TRUE (*opcode*), 1950
pop_source() (*método de shlex.shlex*), 1450
POP_TOP (*opcode*), 1943
Popen (*clase en subprocess*), 893
popen() (*en el módulo os*), 645
popen() (*in module os*), 1069
popitem() (*método de collections.OrderedDict*), 253
popitem() (*método de dict*), 85
popitem() (*método de mailbox.Mailbox*), 1164
popleft() (*método de collections.deque*), 246
poplib
    módulo, 1313
PopupMenu (*clase en tkinter.tix*), 1494
porción, **2089**
port (*atributo de http.cookiejar.Cookie*), 1357
port_specified (*atributo de http.cookiejar.Cookie*), 1357
pos (*atributo de json.JSONDecodeError*), 1158
pos (*atributo de re.error*), 134
pos (*atributo de re.Match*), 138
pos() (*en el módulo operator*), 407
pos() (*en el módulo turtle*), 1418
position (*atributo de xml.etree.ElementTree.ParseError*), 1217
position() (*en el módulo turtle*), 1418
posix
    módulo, 1971
POSIX
    I/O control, 1974
    threads, 916
POSIX Shared Memory, 877
POSIX_FADV_DONTNEED (*en el módulo os*), 614
POSIX_FADV_NOREUSE (*en el módulo os*), 614
POSIX_FADV_NORMAL (*en el módulo os*), 614
POSIX_FADV_RANDOM (*en el módulo os*), 614
POSIX_FADV_SEQUENTIAL (*en el módulo os*), 614
POSIX_FADV_WILLNEED (*en el módulo os*), 614
posix_fadvise() (*en el módulo os*), 614
posix_fallocate() (*en el módulo os*), 614
posix_spawn() (*en el módulo os*), 645
POSIX_SPAWN_CLOSE (*en el módulo os*), 645
POSIX_SPAWN_DUP2 (*en el módulo os*), 645
POSIX_SPAWN_OPEN (*en el módulo os*), 645
posix_spawnp() (*en el módulo os*), 646
POSIXLY_CORRECT, 712
PosixPath (*clase en pathlib*), 423
post() (*método de nntplib.NNTP*), 2027
post() (*método de ossaudiodev.oss_audio_device*), 2058
post_handshake_auth (*atributo de ssl.SSLContext*), 1057
post_mortem() (*en el módulo pdb*), 1698
post_setup() (*método de venv.EnvBuilder*), 1738
postcmd() (*método de cmd.Cmd*), 1444
postloop() (*método de cmd.Cmd*), 1444
Pow (*clase en ast*), 1896
pow()
    función incorporada, 21
pow() (*en el módulo math*), 325
pow() (*en el módulo operator*), 407
power() (*método de decimal.Context*), 347
pp (*pdb command*), 1701
pp() (*en el módulo pprint*), 288
pprint
    módulo, 287
pprint() (*en el módulo pprint*), 288
pprint() (*método de pprint.PrettyPrinter*), 289
prcal() (*en el módulo calendar*), 238
pread() (*en el módulo os*), 614
preadv() (*en el módulo os*), 614
preamble (*atributo de email.message.EmailMessage*), 1102
preamble (*atributo de email.message.Message*), 1139
--preamble=<preamble>
    pickletoolsopción de línea de comando, 1954
precmd() (*método de cmd.Cmd*), 1444
prefix (*atributo de xml.dom.Attr*), 1225
prefix (*atributo de xml.dom.Node*), 1221
prefix (*atributo de zipimport.zipimporter*), 1855
prefix (*en el módulo sys*), 1763
PREFIXES (*en el módulo site*), 1846
prefixlen (*atributo de ipaddress.IPv4Network*), 1379
prefixlen (*atributo de ipaddress.IPv6Network*), 1381
preloop() (*método de cmd.Cmd*), 1444
prepare() (*método de graphlib.TopologicalSorter*), 314
prepare() (*método de logging.handlers.QueueHandler*), 754
prepare() (*método de logging.handlers.QueueListener*), 755
prepare_class() (*en el módulo types*), 280
prepare_input_source() (*en el módulo xml.sax.saxutils*), 1243
PrepareProtocol (*clase en sqlite3*), 504
prepend() (*método de pipes.Template*), 2061
PrettyPrinter (*clase en pprint*), 287
prev() (*método de tkinter.ttk.Treeview*), 1489
previousSibling (*atributo de xml.dom.Node*), 1220

print (*2to3 fixer*), 1664
print()
    función incorporada, 21
print_callees() (*método de pstats.Stats*), 1709
print_callers() (*método de pstats.Stats*), 1709
print_directory() (*en el módulo cgi*), 2001
print_environ() (*en el módulo cgi*), 2001
print_environ_usage() (*en el módulo cgi*), 2001
print_exc() (*en el módulo traceback*), 1816
print_exc() (*método de timeit.Timer*), 1713
print_exception() (*en el módulo traceback*), 1816
PRINT_EXPR (*opcode*), 1946
print_form() (*en el módulo cgi*), 2001
print_help() (*método de argparse.ArgumentParser*), 709
print_last() (*en el módulo traceback*), 1816
print_stack() (*en el módulo traceback*), 1817
print_stack() (*método de asyncio.Task*), 933
print_stats() (*método de profile.Profile*), 1706
print_stats() (*método de pstats.Stats*), 1708
print_tb() (*en el módulo traceback*), 1816
print_usage() (*método de argparse.ArgumentParser*), 709
print_usage() (*método de optparse.OptionParser*), 2048
print_version() (*método de optparse.OptionParser*), 2037
print_warning() (*en el módulo test.support*), 1672
printable (*en el módulo string*), 114
printdir() (*método de zipfile.ZipFile*), 533
printf-style formatting, 57, 72
PRIO_PGRP (*en el módulo os*), 606
PRIO_PROCESS (*en el módulo os*), 606
PRIO_USER (*en el módulo os*), 606
PriorityQueue (*clase en asyncio*), 951
PriorityQueue (*clase en queue*), 910
prlimit() (*en el módulo resource*), 1980
prmonth() (*en el módulo calendar*), 238
prmonth() (*método de calendar.TextCalendar*), 235
ProactorEventLoop (*clase en asyncio*), 972
process
    group, 605, 606
    id, 606
    id of parent, 606
    killing, 644
    scheduling priority, 606, 607
    signalling, 644
--process
    timeitopción de línea de comando, 1714
Process (*clase en multiprocessing*), 841
process() (*método de logging.LoggerAdapter*), 725
process_exited() (*método de asyncio.SubprocessProtocol*), 986
process_message() (*método de smtpd.SMTPServer*), 2061
process_request() (*método de socketserver.BaseServer*), 1336
process_time() (*en el módulo time*), 673
process_time_ns() (*en el módulo time*), 673
process_tokens() (*en el módulo tabnanny*), 1932
ProcessError, 843
processes, light-weight, 916
ProcessingInstruction() (*en el módulo xml.etree.ElementTree*), 1207
processingInstruction() (*método de xml.sax.handler.ContentHandler*), 1240
ProcessingInstructionHandler() (*método de xml.parsers.expat.xmlparser*), 1251
ProcessLookupError, 109
processor time, 673, 676
processor() (*en el módulo platform*), 780
ProcessPoolExecutor (*clase en concurrent.futures*), 885
prod() (*en el módulo math*), 323
product() (*en el módulo itertools*), 389
profile
    módulo, 1705
Profile (*clase en profile*), 1705
profile function, 823, 1757, 1763
profiler, 1757, 1763
profiling, deterministic, 1703
ProgrammingError, 505
Progressbar (*clase en tkinter.ttk*), 1483
prompt (*atributo de cmd.Cmd*), 1444
prompt_user_passwd() (*método de urllib.request.FancyURLopener*), 1287
prompts, interpreter, 1763
propagate (*atributo de logging.Logger*), 714
property (*clase incorporada*), 22
property list, 581
property_declaration_handler (*en el módulo xml.sax.handler*), 1238
property_dom_node (*en el módulo xml.sax.handler*), 1238
property_lexical_handler (*en el módulo xml.sax.handler*), 1238
property_xml_string (*en el módulo xml.sax.handler*), 1238
PropertyMock (*clase en unittest.mock*), 1612
prot_c() (*método de ftplib.FTP_TLS*), 1313
prot_p() (*método de ftplib.FTP_TLS*), 1313
proto (*atributo de socket.socket*), 1029
protocol
    CGI, 1997
    context management, 88
    copy, 473
    FTP, 1287, 1308
    HTTP, 1287, 1299, 1301, 1341, 1997
    IMAP4, 1316
    IMAP4_SSL, 1316
    IMAP4_stream, 1316
    iterator, 41
    NNTP, 2022
    POP3, 1313
    SMTP, 1323

Telnet, 2068
protocol (*atributo de ssl.SSLContext*), 1057
Protocol (*clase en asyncio*), 984
Protocol (*clase en typing*), 1530
PROTOCOL_SSLv2 (*en el módulo ssl*), 1042
PROTOCOL_SSLv3 (*en el módulo ssl*), 1042
PROTOCOL_SSLv23 (*en el módulo ssl*), 1041
PROTOCOL_TLS (*en el módulo ssl*), 1041
PROTOCOL_TLS_CLIENT (*en el módulo ssl*), 1041
PROTOCOL_TLS_SERVER (*en el módulo ssl*), 1041
PROTOCOL_TLSv1 (*en el módulo ssl*), 1042
PROTOCOL_TLSv1_1 (*en el módulo ssl*), 1042
PROTOCOL_TLSv1_2 (*en el módulo ssl*), 1042
protocol_version (*atributo de http.server.BaseHTTPRequestHandler*), 1342
PROTOCOL_VERSION (*atributo de imaplib.IMAP4*), 1322
ProtocolError (*clase en xmlrpc.client*), 1364
proxy() (*en el módulo weakref*), 274
proxyauth() (*método de imaplib.IMAP4*), 1320
ProxyBasicAuthHandler (*clase en urllib.request*), 1274
ProxyDigestAuthHandler (*clase en urllib.request*), 1274
ProxyHandler (*clase en urllib.request*), 1273
ProxyType (*en el módulo weakref*), 276
ProxyTypes (*en el módulo weakref*), 276
pryear() (*método de calendar.TextCalendar*), 236
ps1 (*en el módulo sys*), 1763
ps2 (*en el módulo sys*), 1763
pstats
    módulo, 1707
pstdev() (*en el módulo statistics*), 373
pthread_getcpuclockid() (*en el módulo time*), 671
pthread_kill() (*en el módulo signal*), 1082
pthread_sigmask() (*en el módulo signal*), 1082
pthreads, 916
pty
    módulo, 613, 1975
pu() (*en el módulo turtle*), 1420
publicId (*atributo de xml.dom.DocumentType*), 1222
PullDom (*clase en xml.dom.pulldom*), 1234
punctuation (*en el módulo string*), 114
punctuation_chars (*atributo de shlex.shlex*), 1451
PurePath (*clase en pathlib*), 415
PurePosixPath (*clase en pathlib*), 416
PureProxy (*clase en smtpd*), 2062
PureWindowsPath (*clase en pathlib*), 416
purge() (*en el módulo re*), 134
Purpose.CLIENT_AUTH (*en el módulo ssl*), 1046
Purpose.SERVER_AUTH (*en el módulo ssl*), 1045
push() (*método de asynchat.async_chat*), 1988
push() (*método de code.InteractiveConsole*), 1851
push() (*método de contextlib.ExitStack*), 1803
push_async_callback() (*método de contextlib.AsyncExitStack*), 1804
push_async_exit() (*método de contextlib.AsyncExitStack*), 1804
push_source() (*método de shlex.shlex*), 1450
push_token() (*método de shlex.shlex*), 1450
push_with_producer() (*método de asynchat.async_chat*), 1988
pushbutton() (*método de msilib.Dialog*), 2020
put() (*método de asyncio.Queue*), 950
put() (*método de multiprocessing.Queue*), 845
put() (*método de multiprocessing.SimpleQueue*), 846
put() (*método de queue.Queue*), 911
put() (*método de queue.SimpleQueue*), 912
put_nowait() (*método de asyncio.Queue*), 950
put_nowait() (*método de multiprocessing.Queue*), 845
put_nowait() (*método de queue.Queue*), 911
put_nowait() (*método de queue.SimpleQueue*), 912
putch() (*en el módulo msvcrt*), 1958
putenv() (*en el módulo os*), 606
putheader() (*método de http.client.HTTPConnection*), 1305
putp() (*en el módulo curses*), 761
putrequest() (*método de http.client.HTTPConnection*), 1305
putwch() (*en el módulo msvcrt*), 1958
putwin() (*método de curses.window*), 768
pvariance() (*en el módulo statistics*), 373
pwd
    módulo, 433, 1972
pwd() (*método de ftplib.FTP*), 1312
pwrite() (*en el módulo os*), 615
pwritev() (*en el módulo os*), 615
py_compile
    módulo, 1934
PY_COMPILED (*en el módulo imp*), 2013
PY_FROZEN (*en el módulo imp*), 2013
py_object (*clase en ctypes*), 818
PY_SOURCE (*en el módulo imp*), 2013
pycache_prefix (*en el módulo sys*), 1753
PyCF_ALLOW_TOP_LEVEL_AWAIT (*en el módulo ast*), 1921
PyCF_ONLY_AST (*en el módulo ast*), 1921
PyCF_TYPE_COMMENTS (*en el módulo ast*), 1921
PycInvalidationMode (*clase en py_compile*), 1935
pyclbr
    módulo, 1932
PyCompileError, 1934
PyDLL (*clase en ctypes*), 807
pydoc
    módulo, 1544
pyexpat
    módulo, 1247
PYFUNCTYPE() (*en el módulo ctypes*), 810
Python 3000, **2090**
Python Editor, 1498
Python Enhancement Proposals
    PEP 1, 2089

PEP 8, 24
PEP 205, 276
PEP 227, 1823
PEP 235, 1863
PEP 237, 59, 74
PEP 238, 1823, 2084
PEP 246, 504
PEP 249, 491, 504, 507, 512
PEP 255, 1823
PEP 263, 1863, 1928, 1929
PEP 273, 1853
PEP 278, 2092
PEP 282, 460, 730
PEP 292, 122
PEP 302, 28, 454, 1762, 1856, 1857, 1860, 1863, 1865, 1866, 1868, 1869, 2013, 2083, 2087
PEP 305, 555
PEP 307, 467
PEP 324, 889
PEP 328, 28, 1823
PEP 338, 1862
PEP 342, 260
PEP 343, 1807, 1823, 2081
PEP 362, 1837, 2080, 2089
PEP 366, 1862, 1863
PEP 370, 1847
PEP 378, 117
PEP 383, 177, 1009
PEP 387, 109
PEP 393, 183, 1761
PEP 405, 1733
PEP 411, 1758, 1765, 1766, 2089
PEP 412, 397
PEP 420, 1863, 2083, 2088, 2089
PEP 421, 1759, 1760
PEP 428, 414
PEP 434, 1510
PEP 442, 1826
PEP 443, 2084
PEP 451, 1761, 1856, 1857, 18611863, 2083
PEP 453, 1732
PEP 461, 74
PEP 468, 254
PEP 475, 21, 108, 612, 616, 618, 650, 674, 1022, 1023, 10251027, 10691073, 1076, 1084, 1085
PEP 479, 105, 1823
PEP 483, 1511, 1512, 2084
PEP 484, 93, 1511, 1512, 1514, 1519, 1523, 1528, 1541, 1917, 1918, 1921, 2079, 2084, 2091, 2092
PEP 485, 322, 330
PEP 488, 1683, 1863, 1878, 1879, 1934
PEP 489, 1863, 1874, 1877
PEP 492, 261, 1843, 2080, 2082
PEP 495, 229
PEP 498, 2083
PEP 506, 597
PEP 515, 118
PEP 519, 2089
PEP 524, 656
PEP 525, 261, 1758, 1765, 1843, 2080
PEP 526, 1512, 1523, 1532, 1785, 1792, 1917, 1921, 2079, 2092
PEP 529, 622, 1756, 1766
PEP 540, 602
PEP 544, 1512, 1519, 1530
PEP 552, 1863, 1935
PEP 557, 1785
PEP 560, 281
PEP 563, 1542, 1543, 1823
PEP 565, 109
PEP 566, 1884, 1886, 1888
PEP 567, 913, 956, 957, 977
PEP 574, 467, 479
PEP 578, 1685, 1750
PEP 584, 240, 248, 254, 275, 284, 603, 604
PEP 585, 93, 258, 1512, 15191521, 1523, 15341540, 1543, 2084
PEP 586, 1512
PEP 589, 1512, 1534
PEP 591, 1512, 1524, 1541
PEP 593, 1512, 1524, 1542
PEP 594, 2022
PEP 594#aifc, 1985
PEP 594#asynchat, 1987
PEP 594#asyncore, 1990
PEP 594#audioop, 1994
PEP 594#cgi, 1997
PEP 594#cgitb, 2004
PEP 594#chunk, 2005
PEP 594#crypt, 2006
PEP 594#imghdr, 2008
PEP 594#mailcap, 2014
PEP 594#msilib, 2015
PEP 594#nis, 2021
PEP 594#ossaudiodev, 2055
PEP 594#pipes, 2060
PEP 594#smtpd, 2061
PEP 594#sndhdr, 2064
PEP 594#spwd, 2065
PEP 594#sunau, 2066
PEP 594#telnetlib, 2068
PEP 594#uu-and-the-uu-encoding, 2071
PEP 594#xdrlib, 2072
PEP 597, 658
PEP 604, 94, 1512
PEP 612, 1512, 1514, 1517, 1522, 1529
PEP 613, 1512, 1520
PEP 615, 229
PEP 617, 1665
PEP 626, 1943
PEP 634, 1665
PEP 644, 1034
PEP 647, 1512, 1526
PEP 649, 1823

PEP 706, 548
PEP 3101, 114
PEP 3105, 1823
PEP 3112, 1823
PEP 3115, 281
PEP 3116, 2092
PEP 3118, 75
PEP 3119, 262, 1810
PEP 3120, 1863
PEP 3134, 102
PEP 3141, 317, 1810
PEP 3147, 1683, 1861, 1863, 1878, 1879, 19341936, 1938, 1939, 2011, 2012
PEP 3148, 888
PEP 3149, 1749
PEP 3151, 109, 1011, 1068, 1979
PEP 3154, 467
PEP 3155, 2090
PEP 3333, 12601264, 1267, 1268
--python=<interpreter>
zipappopción de línea de comando, 1743
python_branch() (*en el módulo platform*), 780
python_build() (*en el módulo platform*), 780
python_compiler() (*en el módulo platform*), 780
PYTHON_DOM, 1219
python_implementation() (*en el módulo platform*), 780
python_is_optimized() (*en el módulo test.support*), 1670
python_revision() (*en el módulo platform*), 780
python_version() (*en el módulo platform*), 780
python_version_tuple() (*en el módulo platform*), 780
PYTHONASYNCIODEBUG, 969, 1005, 1546
PYTHONBREAKPOINT, 1752
PYTHONCASEOK, 29
PYTHONCOERCECLOCALE, 603
PYTHONDEVMODE, 1545
PYTHONDOCS, 1545
PYTHONDONTWRITEBYTECODE, 1753
PYTHONFAULTHANDLER, 1546, 1694
PYTHONHOME, 1678
Pythónico, **2090**
PYTHONINTMAXSTRDIGITS, 99, 1760
PYTHONIOENCODING, 602, 1766
PYTHONLEGACYWINDOWSFSENCODING, 1766
PYTHONLEGACYWINDOWSSTDIO, 1766
PYTHONMALLOC, 1546
python--m-py_compileopción de línea de comando
-, 1935
<file> ... <fileN>, 1935
-q, 1935
--quiet, 1935
PYTHONNOUSERSITE, 1846
PYTHONPATH, 1678, 1761, 2002
PYTHONPYCACHEPREFIX, 1753
PYTHONSTARTUP, 165, 1505, 1760, 1845
PYTHONTRACEMALLOC, 1719, 1724
PYTHONTRACEMALLOC`comienzo, configura la variable del entorno a ``25`, 1719
PYTHONTZPATH, 234
PYTHONUNBUFFERED, 1766
PYTHONUSERBASE, 1846
PYTHONUSERSITE, 1678
PYTHONUTF8, 603, 1766
PYTHONWARNDEFAULTENCODING, 658
PYTHONWARNINGS, 1545, 1780, 1781
PyZipFile (*clase en zipfile*), 535

Q

-q
compileallopción de línea de comando, 1936
python--m-py_compileopción de línea de comando, 1935
qiflush() (*en el módulo curses*), 761
QName (*clase en xml.etree.ElementTree*), 1214
qsize() (*método de asyncio.Queue*), 950
qsize() (*método de multiprocessing.Queue*), 845
qsize() (*método de queue.Queue*), 910
qsize() (*método de queue.SimpleQueue*), 912
quantiles() (*en el módulo statistics*), 375
quantiles() (*método de statistics.NormalDist*), 378
quantize() (*método de decimal.Context*), 347
quantize() (*método de decimal.Decimal*), 341
QueryInfoKey() (*en el módulo winreg*), 1962
QueryReflectionKey() (*en el módulo winreg*), 1964
QueryValue() (*en el módulo winreg*), 1962
QueryValueEx() (*en el módulo winreg*), 1963
queue
módulo, 909
queue (*atributo de sched.scheduler*), 909
Queue (*clase en asyncio*), 950
Queue (*clase en multiprocessing*), 845
Queue (*clase en queue*), 909
Queue() (*método de multiprocessing.managers.SyncManager*), 857
QueueEmpty, 951
QueueFull, 951
QueueHandler (*clase en logging.handlers*), 753
QueueListener (*clase en logging.handlers*), 754
quick_ratio() (*método de difflib.SequenceMatcher*), 151
--quiet
python--m-py_compileopción de línea de comando, 1935
quit (*pdb command*), 1702
quit (*variable incorporada*), 32
quit() (*método de ftplib.FTP*), 1313
quit() (*método de nntplib.NNTP*), 2024
quit() (*método de poplib.POP3*), 1315
quit() (*método de smtplib.SMTP*), 1329

quit() (*método de tkinter.filedialog.FileDialog*), 1471
quopri
    módulo, 1188
quote() (*en el módulo email.utils*), 1148
quote() (*en el módulo shlex*), 1448
quote() (*en el módulo urllib.parse*), 1295
QUOTE_ALL (*en el módulo csv*), 559
quote_from_bytes() (*en el módulo urllib.parse*), 1295
QUOTE_MINIMAL (*en el módulo csv*), 559
QUOTE_NONE (*en el módulo csv*), 559
QUOTE_NONNUMERIC (*en el módulo csv*), 559
quote_plus() (*en el módulo urllib.parse*), 1295
quoteattr() (*en el módulo xml.sax.saxutils*), 1242
quotechar (*atributo de csv.Dialect*), 560
quoted-printable
    encoding, 1188
quotes (*atributo de shlex.shlex*), 1451
quoting (*atributo de csv.Dialect*), 560

R

-r
    compileallopción de línea de comando, 1936
    traceopción de línea de comando, 1717
-R
    traceopción de línea de comando, 1717
-r N
    timeitopción de línea de comando, 1714
R_OK (*en el módulo os*), 620
radians() (*en el módulo math*), 326
radians() (*en el módulo turtle*), 1420
RadioButtonGroup (*clase en msilib*), 2020
radiogroup() (*método de msilib.Dialog*), 2020
radix() (*método de decimal.Context*), 347
radix() (*método de decimal.Decimal*), 341
RADIXCHAR (*en el módulo locale*), 1402
raise
    sentencia, 101
raise (*2to3 fixer*), 1664
Raise (*clase en ast*), 1903
raise_on_defect (*atributo de email.policy.Policy*), 1111
raise_signal() (*en el módulo signal*), 1082
RAISE_VARARGS (*opcode*), 1951
RAND_add() (*en el módulo ssl*), 1037
RAND_bytes() (*en el módulo ssl*), 1037
RAND_pseudo_bytes() (*en el módulo ssl*), 1037
RAND_status() (*en el módulo ssl*), 1037
randbelow() (*en el módulo secrets*), 597
randbits() (*en el módulo secrets*), 597
randbytes() (*en el módulo random*), 362
randint() (*en el módulo random*), 362
random
    módulo, 360
Random (*clase en random*), 365
random() (*en el módulo random*), 364
randrange() (*en el módulo random*), 362
range
    objeto, 46
range (*clase incorporada*), 46
RARROW (*en el módulo token*), 1926
ratecv() (*en el módulo audioop*), 1996
ratio() (*método de difflib.SequenceMatcher*), 151
Rational (*clase en numbers*), 318
raw (*atributo de io.BufferedIOBase*), 663
raw() (*en el módulo curses*), 761
raw() (*método de pickle.PickleBuffer*), 471
raw_data_manager (*en el módulo email.contentmanager*), 1124
raw_decode() (*método de json.JSONDecoder*), 1156
raw_input (*2to3 fixer*), 1664
raw_input() (*método de code.InteractiveConsole*), 1851
RawArray() (*en el módulo multiprocessing.sharedctypes*), 853
RawConfigParser (*clase en configparser*), 579
RawDescriptionHelpFormatter (*clase en argparse*), 685
RawIOBase (*clase en io*), 662
RawPen (*clase en turtle*), 1437
RawTextHelpFormatter (*clase en argparse*), 685
RawTurtle (*clase en turtle*), 1437
RawValue() (*en el módulo multiprocessing.sharedctypes*), 853
RBRACE (*en el módulo token*), 1925
rcpttos (*atributo de smtpd.SMTPChannel*), 2063
re
    módulo, 49, 124, 452
re (*atributo de re.Match*), 138
read() (*en el módulo os*), 616
read() (*método de asyncio.StreamReader*), 936
read() (*método de chunk.Chunk*), 2006
read() (*método de codecs.StreamReader*), 182
read() (*método de configparser.ConfigParser*), 576
read() (*método de http.client.HTTPResponse*), 1306
read() (*método de imaplib.IMAP4*), 1320
read() (*método de io.BufferedIOBase*), 663
read() (*método de io.BufferedReader*), 665
read() (*método de io.RawIOBase*), 662
read() (*método de io.TextIOBase*), 667
read() (*método de mimetypes.MimeTypes*), 1181
read() (*método de mmap.mmap*), 1089
read() (*método de ossaudiodev.oss_audio_device*), 2056
read() (*método de ssl.MemoryBIO*), 1065
read() (*método de ssl.SSLSocket*), 1047
read() (*método de urllib.robotparser.RobotFileParser*), 1298
read() (*método de zipfile.ZipFile*), 533
read1() (*método de io.BufferedIOBase*), 663
read1() (*método de io.BufferedReader*), 665
read1() (*método de io.BytesIO*), 665

read_all() (*método de telnetlib.Telnet*), 2069
read_binary() (*en el módulo importlib.resources*), 1873
read_byte() (*método de mmap.mmap*), 1089
read_bytes() (*método de importlib.abc.Traversable*), 1871
read_bytes() (*método de pathlib.Path*), 427
read_bytes() (*método de zipfile.Path*), 535
read_dict() (*método de configparser.ConfigParser*), 576
read_eager() (*método de telnetlib.Telnet*), 2069
read_environ() (*en el módulo wsgiref.handlers*), 1268
read_events() (*método de xml.etree.ElementTree.XMLPullParser*), 1217
read_file() (*método de configparser.ConfigParser*), 576
read_history_file() (*en el módulo readline*), 163
read_init_file() (*en el módulo readline*), 162
read_lazy() (*método de telnetlib.Telnet*), 2069
read_mime_types() (*en el módulo mimetypes*), 1180
read_sb_data() (*método de telnetlib.Telnet*), 2070
read_some() (*método de telnetlib.Telnet*), 2069
read_string() (*método de configparser.ConfigParser*), 576
read_text() (*en el módulo importlib.resources*), 1873
read_text() (*método de importlib.abc.Traversable*), 1872
read_text() (*método de pathlib.Path*), 427
read_text() (*método de zipfile.Path*), 535
read_token() (*método de shlex.shlex*), 1450
read_until() (*método de telnetlib.Telnet*), 2069
read_very_eager() (*método de telnetlib.Telnet*), 2069
read_very_lazy() (*método de telnetlib.Telnet*), 2069
read_windows_registry() (*método de mimetypes.MimeTypes*), 1181
READABLE (*en el módulo tkinter*), 1467
readable() (*método de asyncore.dispatcher*), 1992
readable() (*método de io.IOBase*), 661
readall() (*método de io.RawIOBase*), 662
reader() (*en el módulo csv*), 556
ReadError, 541
readexactly() (*método de asyncio.StreamReader*), 937
readfp() (*método de configparser.ConfigParser*), 578
readfp() (*método de mimetypes.MimeTypes*), 1181
readframes() (*método de aifc.aifc*), 1986
readframes() (*método de sunau.AU_read*), 2067
readframes() (*método de wave.Wave_read*), 1388
readinto() (*método de http.client.HTTPResponse*), 1306
readinto() (*método de io.BufferedIOBase*), 663
readinto() (*método de io.RawIOBase*), 662
readinto1() (*método de io.BufferedIOBase*), 663
readinto1() (*método de io.BytesIO*), 665
readline
    módulo, 162
readline() (*método de asyncio.StreamReader*), 936
readline() (*método de codecs.StreamReader*), 182
readline() (*método de imaplib.IMAP4*), 1320
readline() (*método de io.IOBase*), 661
readline() (*método de io.TextIOBase*), 667
readline() (*método de mmap.mmap*), 1089
readlines() (*método de codecs.StreamReader*), 182
readlines() (*método de io.IOBase*), 661
readlink() (*en el módulo os*), 626
readlink() (*método de pathlib.Path*), 427
readmodule() (*en el módulo pyclbr*), 1932
readmodule_ex() (*en el módulo pyclbr*), 1932
readonly (*atributo de memoryview*), 80
ReadTransport (*clase en asyncio*), 979
readuntil() (*método de asyncio.StreamReader*), 937
readv() (*en el módulo os*), 617
ready() (*método de multiprocessing.pool.AsyncResult*), 863
real (*atributo de numbers.Complex*), 317
Real (*clase en numbers*), 317
Real Media File Format, 2005
real_max_memuse (*en el módulo test.support*), 1670
real_quick_ratio() (*método de difflib.SequenceMatcher*), 151
realpath() (*en el módulo os.path*), 435
REALTIME_PRIORITY_CLASS (*en el módulo subprocess*), 901
reap_children() (*en el módulo test.support*), 1674
reap_threads() (*en el módulo test.support.threading_helper*), 1679
reason (*atributo de http.client.HTTPResponse*), 1306
reason (*atributo de ssl.SSLError*), 1036
reason (*atributo de UnicodeError*), 107
reason (*atributo de urllib.error.HTTPError*), 1297
reason (*atributo de urllib.error.URLError*), 1297
reattach() (*método de tkinter.ttk.Treeview*), 1489
rebanada, **2091**
reccontrols() (*método de ossaudiodev.oss_mixer_device*), 2059
received_data (*atributo de smtpd.SMTPChannel*), 2064
received_lines (*atributo de smtpd.SMTPChannel*), 2063
recent() (*método de imaplib.IMAP4*), 1320
recolección de basura, **2084**
reconfigure() (*método de io.TextIOWrapper*), 668
record_original_stdout() (*en el módulo test.support*), 1671
records (*atributo de unittest.TestCase*), 1585
rect() (*en el módulo cmath*), 329
rectangle() (*en el módulo curses.textpad*), 774
RecursionError, 105
recursive_repr() (*en el módulo reprlib*), 293
recv() (*método de asyncore.dispatcher*), 1992

recv() (*método de multiprocessing.connection.Connection*), 848
recv() (*método de socket.socket*), 1024
recv_bytes() (*método de multiprocessing.connection.Connection*), 849
recv_bytes_into() (*método de multiprocessing.connection.Connection*), 849
recv_fds() (*en el módulo socket*), 1022
recv_into() (*método de socket.socket*), 1026
recvfrom() (*método de socket.socket*), 1025
recvfrom_into() (*método de socket.socket*), 1026
recvmsg() (*método de socket.socket*), 1025
recvmsg_into() (*método de socket.socket*), 1026
redirect_request() (*método de urllib.request.HTTPRedirectHandler*), 1279
redirect_stderr() (*en el módulo contextlib*), 1800
redirect_stdout() (*en el módulo contextlib*), 1800
redisplay() (*en el módulo readline*), 162
redrawln() (*método de curses.window*), 768
redrawwin() (*método de curses.window*), 768
reduce (*2to3 fixer*), 1664
reduce() (*en el módulo functools*), 401
reducer_override() (*método de pickle.Pickler*), 469
ref (*clase en weakref*), 273
refcount_test() (*en el módulo test.support*), 1674
ReferenceError, 105
ReferenceType (*en el módulo weakref*), 276
referencia fuerte, **2091**
referencia prestada, **2081**
refold_source (*atributo de email.policy.EmailPolicy*), 1113
refresh() (*método de curses.window*), 768
REG_BINARY (*en el módulo winreg*), 1966
REG_DWORD (*en el módulo winreg*), 1966
REG_DWORD_BIG_ENDIAN (*en el módulo winreg*), 1966
REG_DWORD_LITTLE_ENDIAN (*en el módulo winreg*), 1966
REG_EXPAND_SZ (*en el módulo winreg*), 1966
REG_FULL_RESOURCE_DESCRIPTOR (*en el módulo winreg*), 1966
REG_LINK (*en el módulo winreg*), 1966
REG_MULTI_SZ (*en el módulo winreg*), 1966
REG_NONE (*en el módulo winreg*), 1966
REG_QWORD (*en el módulo winreg*), 1966
REG_QWORD_LITTLE_ENDIAN (*en el módulo winreg*), 1966
REG_RESOURCE_LIST (*en el módulo winreg*), 1966
REG_RESOURCE_REQUIREMENTS_LIST (*en el módulo winreg*), 1967
REG_SZ (*en el módulo winreg*), 1967
register() (*en el módulo atexit*), 1814
register() (*en el módulo codecs*), 175
register() (*en el módulo faulthandler*), 1696
register() (*en el módulo webbrowser*), 1258
register() (*método de abc.ABCMeta*), 1810
register() (*método de multiprocessing.managers.BaseManager*), 856
register() (*método de select.devpoll*), 1070
register() (*método de select.epoll*), 1071
register() (*método de selectors.BaseSelector*), 1076
register() (*método de select.poll*), 1072
register_adapter() (*en el módulo sqlite3*), 494
register_archive_format() (*en el módulo shutil*), 461
register_at_fork() (*en el módulo os*), 646
register_converter() (*en el módulo sqlite3*), 494
register_defect() (*método de email.policy.Policy*), 1111
register_dialect() (*en el módulo csv*), 556
register_error() (*en el módulo codecs*), 178
register_function() (*método de xmlrpc.server.CGIXMLRPCRequestHandler*), 1370
register_function() (*método de xmlrpc.server.SimpleXMLRPCServer*), 1367
register_instance() (*método de xmlrpc.server.CGIXMLRPCRequestHandler*), 1370
register_instance() (*método de xmlrpc.server.SimpleXMLRPCServer*), 1367
register_introspection_functions() (*método de xmlrpc.server.CGIXMLRPCRequestHandler*), 1370
register_introspection_functions() (*método de xmlrpc.server.SimpleXMLRPCServer*), 1368
register_multicall_functions() (*método de xmlrpc.server.CGIXMLRPCRequestHandler*), 1371
register_multicall_functions() (*método de xmlrpc.server.SimpleXMLRPCServer*), 1368
register_namespace() (*en el módulo xml.etree.ElementTree*), 1208
register_optionflag() (*en el módulo doctest*), 1557
register_shape() (*en el módulo turtle*), 1435
register_unpack_format() (*en el módulo shutil*), 461
registerDOMImplementation() (*en el módulo xml.dom*), 1219
registerResult() (*en el módulo unittest*), 1601
relative URL, 1288
relative_to() (*método de pathlib.PurePath*), 421
release() (*en el módulo platform*), 780
release() (*método de _thread.lock*), 918
release() (*método de asyncio.Condition*), 943
release() (*método de asyncio.Lock*), 942
release() (*método de asyncio.Semaphore*), 944
release() (*método de logging.Handler*), 719
release() (*método de memoryview*), 77

release() (*método de multiprocessing.Lock*), 851
release() (*método de multiprocessing.RLock*), 851
release() (*método de pickle.PickleBuffer*), 471
release() (*método de threading.Condition*), 829
release() (*método de threading.Lock*), 827
release() (*método de threading.RLock*), 828
release() (*método de threading.Semaphore*), 831
release_lock() (*en el módulo imp*), 2012
reload (*2to3 fixer*), 1664
reload() (*en el módulo imp*), 2010
reload() (*en el módulo importlib*), 1864
relpath() (*en el módulo os.path*), 435
remainder() (*en el módulo math*), 323
remainder() (*método de decimal.Context*), 347
remainder_near() (*método de decimal.Context*), 347
remainder_near() (*método de decimal.Decimal*), 341
RemoteDisconnected, 1303
remove() (*en el módulo os*), 626
remove() (*método de array.array*), 272
remove() (*método de collections.deque*), 246
remove() (*método de frozenset*), 83
remove() (*método de mailbox.Mailbox*), 1162
remove() (*método de mailbox.MH*), 1168
remove() (*método de xml.etree.ElementTree.Element*), 1212
remove() (*sequence method*), 44
remove_child_handler() (*método de asyncio.AbstractChildWatcher*), 994
remove_done_callback() (*método de asyncio.Future*), 977
remove_done_callback() (*método de asyncio.Task*), 933
remove_flag() (*método de mailbox.MaildirMessage*), 1171
remove_flag() (*método de mailbox.mboxMessage*), 1172
remove_flag() (*método de mailbox.MMDFMessage*), 1176
remove_folder() (*método de mailbox.Maildir*), 1165
remove_folder() (*método de mailbox.MH*), 1167
remove_header() (*método de urllib.request.Request*), 1276
remove_history_item() (*en el módulo readline*), 163
remove_label() (*método de mailbox.BabylMessage*), 1175
remove_option() (*método de configparser.ConfigParser*), 578
remove_option() (*método de optparse.OptionParser*), 2047
remove_pyc() (*método de msilib.Directory*), 2019
remove_reader() (*método de asyncio.loop*), 964
remove_section() (*método de configparser.ConfigParser*), 578
remove_sequence() (*método de mailbox.MHMessage*), 1174
remove_signal_handler() (*método de asyncio.loop*), 966
remove_writer() (*método de asyncio.loop*), 964
removeAttribute() (*método de xml.dom.Element*), 1224
removeAttributeNode() (*método de xml.dom.Element*), 1224
removeAttributeNS() (*método de xml.dom.Element*), 1224
removeChild() (*método de xml.dom.Node*), 1221
removedirs() (*en el módulo os*), 626
removeFilter() (*método de logging.Handler*), 719
removeFilter() (*método de logging.Logger*), 717
removeHandler() (*en el módulo unittest*), 1601
removeHandler() (*método de logging.Logger*), 717
removeprefix() (*método de bytearray*), 62
removeprefix() (*método de bytes*), 62
removeprefix() (*método de str*), 53
removeResult() (*en el módulo unittest*), 1601
removesuffix() (*método de bytearray*), 62
removesuffix() (*método de bytes*), 62
removesuffix() (*método de str*), 53
removexattr() (*en el módulo os*), 640
rename() (*en el módulo os*), 626
rename() (*método de ftplib.FTP*), 1312
rename() (*método de imaplib.IMAP4*), 1320
rename() (*método de pathlib.Path*), 428
renames (*2to3 fixer*), 1664
renames() (*en el módulo os*), 627
reopenIfNeeded() (*método de logging.handlers.WatchedFileHandler*), 744
reorganize() (*método de dbm.gnu.gdbm*), 489
repeat() (*en el módulo itertools*), 389
repeat() (*en el módulo timeit*), 1712
repeat() (*método de timeit.Timer*), 1713
--repeat=N
    timeitopción de línea de comando, 1714
repetition
    operation, 42
replace
    error handler's name, 177
replace() (*en el módulo dataclasses*), 1790
replace() (*en el módulo os*), 627
replace() (*método de bytearray*), 64
replace() (*método de bytes*), 64
replace() (*método de curses.panel.Panel*), 778
replace() (*método de datetime.date*), 200
replace() (*método de datetime.datetime*), 207
replace() (*método de datetime.time*), 215
replace() (*método de inspect.Parameter*), 1835
replace() (*método de inspect.Signature*), 1833
replace() (*método de pathlib.Path*), 428
replace() (*método de str*), 53
replace() (*método de types.CodeType*), 281
replace_errors() (*en el módulo codecs*), 178

replace_header() (*método de email.message.EmailMessage*), 1097
replace_header() (*método de email.message.Message*), 1136
replace_history_item() (*en el módulo readline*), 163
replace_whitespace (*atributo de textwrap.TextWrapper*), 157
replaceChild() (*método de xml.dom.Node*), 1222
ReplacePackage() (*en el módulo modulefinder*), 1859
--report
    traceopción de línea de comando, 1717
report() (*método de filecmp.dircmp*), 445
report() (*método de modulefinder.ModuleFinder*), 1859
REPORT_CDIFF (*en el módulo doctest*), 1556
report_failure() (*método de doctest.DocTestRunner*), 1566
report_full_closure() (*método de filecmp.dircmp*), 445
REPORT_NDIFF (*en el módulo doctest*), 1556
REPORT_ONLY_FIRST_FAILURE (*en el módulo doctest*), 1556
report_partial_closure() (*método de filecmp.dircmp*), 445
report_start() (*método de doctest.DocTestRunner*), 1566
report_success() (*método de doctest.DocTestRunner*), 1566
REPORT_UDIFF (*en el módulo doctest*), 1556
report_unexpected_exception() (*método de doctest.DocTestRunner*), 1566
REPORTING_FLAGS (*en el módulo doctest*), 1556
repr (*2to3 fixer*), 1664
Repr (*clase en reprlib*), 292
repr()
    función incorporada, 23
repr() (*en el módulo reprlib*), 292
repr() (*método de reprlib.Repr*), 293
repr1() (*método de reprlib.Repr*), 293
reprlib
    módulo, 292
Request (*clase en urllib.request*), 1272
request() (*método de http.client.HTTPConnection*), 1304
request_queue_size (*atributo de socketserver.BaseServer*), 1336
request_rate() (*método de urllib.robotparser.RobotFileParser*), 1298
request_uri() (*en el módulo wsgiref.util*), 1260
request_version (*atributo de http.server.BaseHTTPRequestHandler*), 1342
RequestHandlerClass (*atributo de socketserver.BaseServer*), 1335
requestline (*atributo de http.server.BaseHTTPRequestHandler*), 1341
requires() (*en el módulo test.support*), 1670
requires_bz2() (*en el módulo test.support*), 1673
requires_docstrings() (*en el módulo test.support*), 1673
requires_freebsd_version() (*en el módulo test.support*), 1673
requires_gzip() (*en el módulo test.support*), 1673
requires_IEEE_754() (*en el módulo test.support*), 1673
requires_linux_version() (*en el módulo test.support*), 1673
requires_lzma() (*en el módulo test.support*), 1673
requires_mac_version() (*en el módulo test.support*), 1673
requires_resource() (*en el módulo test.support*), 1673
requires_zlib() (*en el módulo test.support*), 1673
RERAISE (*opcode*), 1947
reserved (*atributo de zipfile.ZipInfo*), 537
RESERVED_FUTURE (*en el módulo uuid*), 1332
RESERVED_MICROSOFT (*en el módulo uuid*), 1332
RESERVED_NCS (*en el módulo uuid*), 1332
reset() (*en el módulo turtle*), 1424
reset() (*método de bdb.Bdb*), 1691
reset() (*método de codecs.IncrementalDecoder*), 180
reset() (*método de codecs.IncrementalEncoder*), 180
reset() (*método de codecs.StreamReader*), 182
reset() (*método de codecs.StreamWriter*), 181
reset() (*método de contextvars.ContextVar*), 913
reset() (*método de html.parser.HTMLParser*), 1193
reset() (*método de ossaudiodev.oss_audio_device*), 2058
reset() (*método de pipes.Template*), 2060
reset() (*método de threading.Barrier*), 833
reset() (*método de xdrlib.Packer*), 2072
reset() (*método de xdrlib.Unpacker*), 2073
reset() (*método de xml.dom.pulldom.DOMEventStream*), 1234
reset() (*método de xml.sax.xmlreader.IncrementalParser*), 1245
reset_mock() (*método de unittest.mock.AsyncMock*), 1616
reset_mock() (*método de unittest.mock.Mock*), 1606
reset_peak() (*en el módulo tracemalloc*), 1724
reset_prog_mode() (*en el módulo curses*), 761
reset_shell_mode() (*en el módulo curses*), 761
reset_tzpath() (*en el módulo zoneinfo*), 233
resetbuffer() (*método de code.InteractiveConsole*), 1851
resetlocale() (*en el módulo locale*), 1404
resetscreen() (*en el módulo turtle*), 1431
resetty() (*en el módulo curses*), 761
resetwarnings() (*en el módulo warnings*), 1784
resize() (*en el módulo ctypes*), 814
resize() (*método de curses.window*), 768
resize() (*método de mmap.mmap*), 1090
resize_term() (*en el módulo curses*), 761
resizemode() (*en el módulo turtle*), 1425

resizeterm() (*en el módulo curses*), 761
resolution (*atributo de datetime.date*), 200
resolution (*atributo de datetime.datetime*), 206
resolution (*atributo de datetime.time*), 214
resolution (*atributo de datetime.timedelta*), 196
resolve() (*método de pathlib.Path*), 428
resolve_bases() (*en el módulo types*), 281
resolve_name() (*en el módulo importlib.util*), 1879
resolve_name() (*en el módulo pkgutil*), 1858
resolveEntity() (*método de xml.sax.handler.EntityResolver*), 1241
resource
    módulo, 1979
Resource (*en el módulo importlib.resources*), 1872
resource_path() (*método de importlib.abc.ResourceReader*), 1868
ResourceDenied, 1668
ResourceLoader (*clase en importlib.abc*), 1868
ResourceReader (*clase en importlib.abc*), 1868
ResourceWarning, 110
response (*atributo de nntplib.NNTPError*), 2023
response() (*método de imaplib.IMAP4*), 1320
ResponseNotReady, 1303
responses (*atributo de http.server.BaseHTTPRequestHandler*), 1342
responses (*en el módulo http.client*), 1303
restart (*pdb command*), 1702
restore() (*en el módulo difflib*), 147
restore() (*método de test.support.SaveSignals*), 1676
restype (*atributo de ctypes._FuncPtr*), 809
result() (*método de asyncio.Future*), 976
result() (*método de asyncio.Task*), 932
result() (*método de concurrent.futures.Future*), 886
results() (*método de trace.Trace*), 1718
resume_reading() (*método de asyncio.ReadTransport*), 981
resume_writing() (*método de asyncio.BaseProtocol*), 984
retr() (*método de poplib.POP3*), 1315
retrbinary() (*método de ftplib.FTP*), 1311
retrieve() (*método de urllib.request.URLopener*), 1286
retrlines() (*método de ftplib.FTP*), 1311
retrollamada, **2081**
Return (*clase en ast*), 1915
return (*pdb command*), 1701
return_annotation (*atributo de inspect.Signature*), 1833
return_ok() (*método de http.cookiejar.CookiePolicy*), 1354
return_value (*atributo de unittest.mock.Mock*), 1608
RETURN_VALUE (*opcode*), 1946
returncode (*atributo de asyncio.subprocess.Process*), 948
returncode (*atributo de subprocess.CalledProcessError*), 891
returncode (*atributo de subprocess.CompletedProcess*), 890
returncode (*atributo de subprocess.Popen*), 899
retval (*pdb command*), 1702
reverse() (*en el módulo audioop*), 1996
reverse() (*método de array.array*), 272
reverse() (*método de collections.deque*), 246
reverse() (*sequence method*), 44
reverse_order() (*método de pstats.Stats*), 1708
reverse_pointer (*atributo de ipaddress.IPv4Address*), 1374
reverse_pointer (*atributo de ipaddress.IPv6Address*), 1375
reversed()
    función incorporada, 23
Reversible (*clase en collections.abc*), 260
Reversible (*clase en typing*), 1538
revert() (*método de http.cookiejar.FileCookieJar*), 1353
rewind() (*método de aifc.aifc*), 1986
rewind() (*método de sunau.AU_read*), 2067
rewind() (*método de wave.Wave_read*), 1388
RFC
    RFC 821, 1323, 1325
    RFC 822, 674, 1125, 1142, 1305, 1326, 1327, 1329, 1395
    RFC 854, 2068, 2069
    RFC 977, 2022
    RFC 1014, 2072
    RFC 1123, 674
    RFC 1321, 585
    RFC 1422, 1058, 1067
    RFC 1521, 1185, 1188
    RFC 1522, 1188
    RFC 1524, 2014
    RFC 1730, 1316
    RFC 1738, 1297
    RFC 1750, 1037
    RFC 1766, 1403, 1404
    RFC 1808, 1289, 1297
    RFC 1832, 2072
    RFC 1869, 1323, 1325
    RFC 1870, 2061, 2064
    RFC 1939, 1313
    RFC 2045, 1093, 1098, 1120, 1136, 1137, 1143, 1182, 1184
    RFC 2045#section-6.8, 1363
    RFC 2046, 1093, 1124, 1143
    RFC 2047, 1093, 1113, 1118, 1119, 1143, 1144, 1149
    RFC 2060, 1321
    RFC 2068, 1347
    RFC 2104, 595
    RFC 2109, 13471351, 1356, 1357
    RFC 2183, 1093, 1099, 1138
    RFC 2231, 1093, 1097, 1098, 1136, 1137, 1143, 1150
    RFC 2295, 1301
    RFC 2324, 1300
    RFC 2342, 1320

RFC 2368, 1297
RFC 2373, 1374
RFC 2396, 1292, 1295, 1297
RFC 2397, 1282
RFC 2449, 1315
RFC 2518, 1300
RFC 2595, 1313, 1316
RFC 2616, 1261, 1264, 1279, 1287, 1297
RFC 2640, 1308, 1309
RFC 2732, 1297
RFC 2774, 1301
RFC 2818, 1037
RFC 2821, 1093
RFC 2822, 674, 675, 1134, 11421144, 11481150, 1170, 1302, 1342
RFC 2964, 1352
RFC 2965, 1272, 1275, 13501352, 13541358
RFC 2980, 2022, 2028
RFC 3056, 1376
RFC 3171, 1374
RFC 3229, 1300
RFC 3280, 1048
RFC 3330, 1374
RFC 3454, 160
RFC 3490, 188, 190
RFC 3490#section-3.1, 190
RFC 3492, 188, 190
RFC 3493, 1033
RFC 3501, 1321
RFC 3542, 1020
RFC 3548, 1186
RFC 3659, 1312
RFC 3879, 1376
RFC 3927, 1374
RFC 3977, 2022, 2024, 2025, 2028
RFC 3986, 1290, 1293, 1295, 1297
RFC 4007, 1375, 1376
RFC 4086, 1067
RFC 4122, 13291332
RFC 4180, 555
RFC 4193, 1376
RFC 4217, 1309
RFC 4291, 1375
RFC 4380, 1376
RFC 4627, 1152, 1160
RFC 4642, 2023
RFC 4648, 1182, 1183, 1185, 2077
RFC 4918, 1300, 1301
RFC 4954, 1326
RFC 5161, 1319
RFC 5246, 1045, 1067
RFC 5280, 1037, 1038, 1067
RFC 5321, 1122, 2061, 2062
RFC 5322, 1093, 1094, 1104, 1107, 1108, 1111, 1113, 11161119, 1122, 1131, 1328
RFC 5424, 749
RFC 5735, 1374
RFC 5842, 1300, 1301
RFC 5891, 190
RFC 5895, 190
RFC 5929, 1049
RFC 6066, 1044, 1054, 1067
RFC 6125, 1037, 1038
RFC 6152, 2061
RFC 6531, 1095, 1113, 1324, 2061, 2063
RFC 6532, 1093, 1094, 1104, 1113
RFC 6585, 1300, 1301
RFC 6855, 1319
RFC 6856, 1315
RFC 7159, 1152, 1158, 1160
RFC 7230, 1272, 1305
RFC 7231, 1300, 1301
RFC 7232, 1300
RFC 7233, 1300
RFC 7235, 1300
RFC 7238, 1300
RFC 7301, 1044, 1053
RFC 7525, 1067
RFC 7540, 1300
RFC 7693, 588
RFC 7725, 1301
RFC 7914, 588
RFC 8297, 1300
RFC 8305, 959
RFC 8470, 1300
rfc2109 (*atributo de http.cookiejar.Cookie*), 1357
rfc2109_as_netscape (*atributo de http.cookiejar.DefaultCookiePolicy*), 1356
rfc2965 (*atributo de http.cookiejar.CookiePolicy*), 1355
RFC_4122 (*en el módulo uuid*), 1332
rfile (*atributo de http.server.BaseHTTPRequestHandler*), 1342
rfind() (*método de bytearray*), 64
rfind() (*método de bytes*), 64
rfind() (*método de mmap.mmap*), 1090
rfind() (*método de str*), 54
rgb_to_hls() (*en el módulo colorsys*), 1390
rgb_to_hsv() (*en el módulo colorsys*), 1390
rgb_to_yiq() (*en el módulo colorsys*), 1390
rglob() (*método de pathlib.Path*), 428
right (*atributo de filecmp.dircmp*), 445
right() (*en el módulo turtle*), 1414
right_list (*atributo de filecmp.dircmp*), 445
right_only (*atributo de filecmp.dircmp*), 445
RIGHTSHIFT (*en el módulo token*), 1925
RIGHTSHIFTEQUAL (*en el módulo token*), 1926
rindex() (*método de bytearray*), 64
rindex() (*método de bytes*), 64
rindex() (*método de str*), 54
rjust() (*método de bytearray*), 66
rjust() (*método de bytes*), 66
rjust() (*método de str*), 54
rlcompleter
    módulo, 166
rlecode_hqx() (*en el módulo binascii*), 1187

rledecode_hqx() (*en el módulo binascii*), 1187
RLIM_INFINITY (*en el módulo resource*), 1979
RLIMIT_AS (*en el módulo resource*), 1981
RLIMIT_CORE (*en el módulo resource*), 1980
RLIMIT_CPU (*en el módulo resource*), 1980
RLIMIT_DATA (*en el módulo resource*), 1980
RLIMIT_FSIZE (*en el módulo resource*), 1980
RLIMIT_KQUEUES (*en el módulo resource*), 1981
RLIMIT_MEMLOCK (*en el módulo resource*), 1980
RLIMIT_MSGQUEUE (*en el módulo resource*), 1981
RLIMIT_NICE (*en el módulo resource*), 1981
RLIMIT_NOFILE (*en el módulo resource*), 1980
RLIMIT_NPROC (*en el módulo resource*), 1980
RLIMIT_NPTS (*en el módulo resource*), 1981
RLIMIT_OFILE (*en el módulo resource*), 1980
RLIMIT_RSS (*en el módulo resource*), 1980
RLIMIT_RTPRIO (*en el módulo resource*), 1981
RLIMIT_RTTIME (*en el módulo resource*), 1981
RLIMIT_SBSIZE (*en el módulo resource*), 1981
RLIMIT_SIGPENDING (*en el módulo resource*), 1981
RLIMIT_STACK (*en el módulo resource*), 1980
RLIMIT_SWAP (*en el módulo resource*), 1981
RLIMIT_VMEM (*en el módulo resource*), 1981
RLock (*clase en multiprocessing*), 851
RLock (*clase en threading*), 827
RLock() (*método de multiprocessing.managers.SyncManager*), 857
rmd() (*método de ftplib.FTP*), 1312
rmdir() (*en el módulo os*), 627
rmdir() (*en el módulo test.support.os_helper*), 1681
rmdir() (*método de pathlib.Path*), 429
RMFF, 2005
rms() (*en el módulo audioop*), 1996
rmtree() (*en el módulo shutil*), 457
rmtree() (*en el módulo test.support.os_helper*), 1681
RobotFileParser (*clase en urllib.robotparser*), 1298
robots.txt, 1298
rollback() (*método de sqlite3.Connection*), 496
ROMAN (*en el módulo tkinter.font*), 1468
root (*atributo de pathlib.PurePath*), 418
ROT_FOUR (*opcode*), 1944
ROT_N (*opcode*), 1953
ROT_THREE (*opcode*), 1944
ROT_TWO (*opcode*), 1943
rotate() (*método de collections.deque*), 246
rotate() (*método de decimal.Context*), 347
rotate() (*método de decimal.Decimal*), 341
rotate() (*método de logging.handlers.BaseRotatingHandler*), 745
RotatingFileHandler (*clase en logging.handlers*), 745
rotation_filename() (*método de logging.handlers.BaseRotatingHandler*), 745
rotator (*atributo de logging.handlers.BaseRotatingHandler*), 745
round()
    función incorporada, 23
ROUND_05UP (*en el módulo decimal*), 349
ROUND_CEILING (*en el módulo decimal*), 349
ROUND_DOWN (*en el módulo decimal*), 349
ROUND_FLOOR (*en el módulo decimal*), 349
ROUND_HALF_DOWN (*en el módulo decimal*), 349
ROUND_HALF_EVEN (*en el módulo decimal*), 349
ROUND_HALF_UP (*en el módulo decimal*), 349
ROUND_UP (*en el módulo decimal*), 349
Rounded (*clase en decimal*), 350
Row (*clase en sqlite3*), 504
row_factory (*atributo de sqlite3.Connection*), 501
row_factory (*atributo de sqlite3.Cursor*), 503
rowcount (*atributo de sqlite3.Cursor*), 503
RPAR (*en el módulo token*), 1924
rpartition() (*método de bytearray*), 64
rpartition() (*método de bytes*), 64
rpartition() (*método de str*), 54
rpc_paths (*atributo de xmlrpc.server.SimpleXMLRPCRequestHandler*), 1368
rpop() (*método de poplib.POP3*), 1315
rset() (*método de poplib.POP3*), 1315
RShift (*clase en ast*), 1896
rshift() (*en el módulo operator*), 407
rsplit() (*método de bytearray*), 66
rsplit() (*método de bytes*), 66
rsplit() (*método de str*), 54
RSQB (*en el módulo token*), 1924
rstrip() (*método de bytearray*), 66
rstrip() (*método de bytes*), 66
rstrip() (*método de str*), 54
rt() (*en el módulo turtle*), 1414
RTLD_DEEPBIND (*en el módulo os*), 655
RTLD_GLOBAL (*en el módulo os*), 655
RTLD_LAZY (*en el módulo os*), 655
RTLD_LOCAL (*en el módulo os*), 655
RTLD_NODELETE (*en el módulo os*), 655
RTLD_NOLOAD (*en el módulo os*), 655
RTLD_NOW (*en el módulo os*), 655
ruler (*atributo de cmd.Cmd*), 1445
run (*pdb command*), 1702
Run script, 1500
run() (*en el módulo asyncio*), 923
run() (*en el módulo pdb*), 1697
run() (*en el módulo profile*), 1705
run() (*en el módulo subprocess*), 889
run() (*método de bdb.Bdb*), 1693
run() (*método de contextvars.Context*), 914
run() (*método de doctest.DocTestRunner*), 1566
run() (*método de multiprocessing.Process*), 841
run() (*método de pdb.Pdb*), 1698
run() (*método de profile.Profile*), 1706
run() (*método de sched.scheduler*), 909
run() (*método de test.support.BasicTestRunner*), 1676
run() (*método de threading.Thread*), 825
run() (*método de trace.Trace*), 1718
run() (*método de unittest.IsolatedAsyncioTestCase*), 1590

run() (*método de unittest.TestCase*), 1581
run() (*método de unittest.TestSuite*), 1591
run() (*método de unittest.TextTestRunner*), 1597
run() (*método de wsgiref.handlers.BaseHandler*), 1266
run_coroutine_threadsafe() (*en el módulo asyncio*), 930
run_docstring_examples() (*en el módulo doctest*), 1560
run_doctest() (*en el módulo test.support*), 1671
run_forever() (*método de asyncio.loop*), 955
run_in_executor() (*método de asyncio.loop*), 967
run_in_subinterp() (*en el módulo test.support*), 1675
run_module() (*en el módulo runpy*), 1860
run_path() (*en el módulo runpy*), 1861
run_python_until_end() (*en el módulo test.support.script_helper*), 1678
run_script() (*método de modulefinder.ModuleFinder*), 1859
run_unittest() (*en el módulo test.support*), 1671
run_until_complete() (*método de asyncio.loop*), 955
run_with_locale() (*en el módulo test.support*), 1673
run_with_tz() (*en el módulo test.support*), 1673
runcall() (*en el módulo pdb*), 1698
runcall() (*método de bdb.Bdb*), 1694
runcall() (*método de pdb.Pdb*), 1698
runcall() (*método de profile.Profile*), 1706
runcode() (*método de code.InteractiveInterpreter*), 1850
runctx() (*en el módulo profile*), 1705
runctx() (*método de bdb.Bdb*), 1694
runctx() (*método de profile.Profile*), 1706
runctx() (*método de trace.Trace*), 1718
runeval() (*en el módulo pdb*), 1698
runeval() (*método de bdb.Bdb*), 1693
runeval() (*método de pdb.Pdb*), 1698
runfunc() (*método de trace.Trace*), 1718
running() (*método de concurrent.futures.Future*), 886
runpy
    módulo, 1860
runsource() (*método de code.InteractiveInterpreter*), 1850
runtime_checkable() (*en el módulo typing*), 1530
RuntimeError, 105
RuntimeWarning, 109
RUSAGE_BOTH (*en el módulo resource*), 1983
RUSAGE_CHILDREN (*en el módulo resource*), 1983
RUSAGE_SELF (*en el módulo resource*), 1983
RUSAGE_THREAD (*en el módulo resource*), 1983
ruta de importación, **2085**
RWF_APPEND (*en el módulo os*), 616
RWF_DSYNC (*en el módulo os*), 615
RWF_HIPRI (*en el módulo os*), 615
RWF_NOWAIT (*en el módulo os*), 614
RWF_SYNC (*en el módulo os*), 615

S

-s
    traceopción de línea de comando, 1717
    unittest-discoveropción de línea de comando, 1575
S (*en el módulo re*), 131
-s S
    timeitopción de línea de comando, 1714
-s strip_prefix
    compileallopción de línea de comando, 1936
S_ENFMT (*en el módulo stat*), 443
S_IEXEC (*en el módulo stat*), 443
S_IFBLK (*en el módulo stat*), 441
S_IFCHR (*en el módulo stat*), 441
S_IFDIR (*en el módulo stat*), 441
S_IFDOOR (*en el módulo stat*), 441
S_IFIFO (*en el módulo stat*), 441
S_IFLNK (*en el módulo stat*), 441
S_IFMT() (*en el módulo stat*), 440
S_IFPORT (*en el módulo stat*), 442
S_IFREG (*en el módulo stat*), 441
S_IFSOCK (*en el módulo stat*), 441
S_IFWHT (*en el módulo stat*), 442
S_IMODE() (*en el módulo stat*), 440
S_IREAD (*en el módulo stat*), 443
S_IRGRP (*en el módulo stat*), 442
S_IROTH (*en el módulo stat*), 442
S_IRUSR (*en el módulo stat*), 442
S_IRWXG (*en el módulo stat*), 442
S_IRWXO (*en el módulo stat*), 442
S_IRWXU (*en el módulo stat*), 442
S_ISBLK() (*en el módulo stat*), 439
S_ISCHR() (*en el módulo stat*), 439
S_ISDIR() (*en el módulo stat*), 439
S_ISDOOR() (*en el módulo stat*), 439
S_ISFIFO() (*en el módulo stat*), 439
S_ISGID (*en el módulo stat*), 442
S_ISLNK() (*en el módulo stat*), 439
S_ISPORT() (*en el módulo stat*), 440
S_ISREG() (*en el módulo stat*), 439
S_ISSOCK() (*en el módulo stat*), 439
S_ISUID (*en el módulo stat*), 442
S_ISVTX (*en el módulo stat*), 442
S_ISWHT() (*en el módulo stat*), 440
S_IWGRP (*en el módulo stat*), 442
S_IWOTH (*en el módulo stat*), 442
S_IWRITE (*en el módulo stat*), 443
S_IWUSR (*en el módulo stat*), 442
S_IXGRP (*en el módulo stat*), 442
S_IXOTH (*en el módulo stat*), 442
S_IXUSR (*en el módulo stat*), 442
safe (*atributo de uuid.SafeUUID*), 1330
safe_substitute() (*método de string.Template*), 122
SafeChildWatcher (*clase en asyncio*), 994

saferepr() (*en el módulo pprint*), 289
SafeUUID (*clase en uuid*), 1330
saltos de líneas universales, **2092**
same_files (*atributo de filecmp.dircmp*), 445
same_quantum() (*método de decimal.Context*), 347
same_quantum() (*método de decimal.Decimal*), 341
samefile() (*en el módulo os.path*), 435
samefile() (*método de pathlib.Path*), 429
SameFileError, 455
sameopenfile() (*en el módulo os.path*), 435
samestat() (*en el módulo os.path*), 435
sample() (*en el módulo random*), 363
samples() (*método de statistics.NormalDist*), 377
SATURDAY (*en el módulo calendar*), 238
save() (*método de http.cookiejar.FileCookieJar*), 1353
save() (*método de test.support.SaveSignals*), 1676
SaveAs (*clase en tkinter.filedialog*), 1471
SAVEDCWD (*en el módulo test.support.os_helper*), 1680
SaveFileDialog (*clase en tkinter.filedialog*), 1472
SaveKey() (*en el módulo winreg*), 1963
SaveSignals (*clase en test.support*), 1676
savetty() (*en el módulo curses*), 761
SAX2DOM (*clase en xml.dom.pulldom*), 1234
SAXException, 1235
SAXNotRecognizedException, 1236
SAXNotSupportedException, 1236
SAXParseException, 1236
scaleb() (*método de decimal.Context*), 347
scaleb() (*método de decimal.Decimal*), 341
scandir() (*en el módulo os*), 627
scanf(), 140
sched
    módulo, 908
SCHED_BATCH (*en el módulo os*), 652
SCHED_FIFO (*en el módulo os*), 653
sched_get_priority_max() (*en el módulo os*), 653
sched_get_priority_min() (*en el módulo os*), 653
sched_getaffinity() (*en el módulo os*), 653
sched_getparam() (*en el módulo os*), 653
sched_getscheduler() (*en el módulo os*), 653
SCHED_IDLE (*en el módulo os*), 652
SCHED_OTHER (*en el módulo os*), 652
sched_param (*clase en os*), 653
sched_priority (*atributo de os.sched_param*), 653
SCHED_RESET_ON_FORK (*en el módulo os*), 653
SCHED_RR (*en el módulo os*), 653
sched_rr_get_interval() (*en el módulo os*), 653
sched_setaffinity() (*en el módulo os*), 653
sched_setparam() (*en el módulo os*), 653
sched_setscheduler() (*en el módulo os*), 653
SCHED_SPORADIC (*en el módulo os*), 653
sched_yield() (*en el módulo os*), 653
scheduler (*clase en sched*), 908
schema (*en el módulo msilib*), 2021
scope_id (*atributo de ipaddress.IPv6Address*), 1376
Screen (*clase en turtle*), 1437
screensize() (*en el módulo turtle*), 1431
script_from_examples() (*en el módulo doctest*), 1568
scroll() (*método de curses.window*), 768
ScrolledCanvas (*clase en turtle*), 1437
ScrolledText (*clase en tkinter.scrolledtext*), 1473
scrollok() (*método de curses.window*), 768
scrypt() (*en el módulo hashlib*), 588
seal() (*en el módulo unittest.mock*), 1640
search
    path, module, 453, 1761, 1844
search() (*en el módulo re*), 131
search() (*método de imaplib.IMAP4*), 1320
search() (*método de re.Pattern*), 135
second (*atributo de datetime.datetime*), 206
second (*atributo de datetime.time*), 214
seconds since the epoch, 670
secrets
    módulo, 597
SECTCRE (*atributo de configparser.ConfigParser*), 572
sections() (*método de configparser.ConfigParser*), 575
secuencia, **2090**
secure (*atributo de http.cookiejar.Cookie*), 1357
secure hash algorithm, SHA1, SHA224, SHA256, SHA384, SHA512, 585
Secure Sockets Layer, 1033
security
    CGI, 2002
    http.server, 1346
security considerations, 2075
security_level (*atributo de ssl.SSLContext*), 1057
see() (*método de tkinter.ttk.Treeview*), 1489
seed() (*en el módulo random*), 361
seek() (*método de chunk.Chunk*), 2006
seek() (*método de io.IOBase*), 661
seek() (*método de io.TextIOBase*), 667
seek() (*método de mmap.mmap*), 1090
SEEK_CUR (*en el módulo os*), 612
SEEK_END (*en el módulo os*), 612
SEEK_SET (*en el módulo os*), 612
seekable() (*método de io.IOBase*), 661
seen_greeting (*atributo de smtpd.SMTPChannel*), 2063
select
    módulo, 1068
Select (*clase en tkinter.tix*), 1494
select() (*en el módulo select*), 1069
select() (*método de imaplib.IMAP4*), 1320
select() (*método de selectors.BaseSelector*), 1076
select() (*método de tkinter.ttk.Notebook*), 1482
selected_alpn_protocol() (*método de ssl.SSLSocket*), 1049
selected_npn_protocol() (*método de ssl.SSLSocket*), 1049
selection() (*método de tkinter.ttk.Treeview*), 1489

selection_add() (*método de tkinter.ttk.Treeview*), 1489
selection_remove() (*método de tkinter.ttk.Treeview*), 1489
selection_set() (*método de tkinter.ttk.Treeview*), 1489
selection_toggle() (*método de tkinter.ttk.Treeview*), 1489
selector (*atributo de urllib.request.Request*), 1275
SelectorEventLoop (*clase en asyncio*), 972
SelectorKey (*clase en selectors*), 1075
selectors
    módulo, 1075
SelectSelector (*clase en selectors*), 1077
Semaphore (*clase en asyncio*), 944
Semaphore (*clase en multiprocessing*), 852
Semaphore (*clase en threading*), 830
Semaphore() (*método de multiprocessing.managers.SyncManager*), 857
semaphores, binary, 916
SEMI (*en el módulo token*), 1925
send() (*método de asyncore.dispatcher*), 1992
send() (*método de http.client.HTTPConnection*), 1305
send() (*método de imaplib.IMAP4*), 1321
send() (*método de logging.handlers.DatagramHandler*), 749
send() (*método de logging.handlers.SocketHandler*), 748
send() (*método de multiprocessing.connection.Connection*), 848
send() (*método de socket.socket*), 1026
send_bytes() (*método de multiprocessing.connection.Connection*), 849
send_error() (*método de http.server.BaseHTTPRequestHandler*), 1343
send_fds() (*en el módulo socket*), 1022
send_header() (*método de http.server.BaseHTTPRequestHandler*), 1343
send_message() (*método de smtplib.SMTP*), 1328
send_response() (*método de http.server.BaseHTTPRequestHandler*), 1343
send_response_only() (*método de http.server.BaseHTTPRequestHandler*), 1343
send_signal() (*método de asyncio.subprocess.Process*), 948
send_signal() (*método de asyncio.SubprocessTransport*), 983
send_signal() (*método de subprocess.Popen*), 898
sendall() (*método de socket.socket*), 1027
sendcmd() (*método de ftplib.FTP*), 1311
sendfile() (*en el módulo os*), 616
sendfile() (*método de asyncio.loop*), 963
sendfile() (*método de socket.socket*), 1028
sendfile() (*método de wsgiref.handlers.BaseHandler*), 1268
SendfileNotAvailableError, 952
sendmail() (*método de smtplib.SMTP*), 1327
sendmsg() (*método de socket.socket*), 1027
sendmsg_afalg() (*método de socket.socket*), 1028
sendto() (*método de asyncio.DatagramTransport*), 982
sendto() (*método de socket.socket*), 1027
sentencia, **2091**
    assert, 103
    del, 44, 83
    except, 101
    if, 33
    import, 28, 1844, 2009
    raise, 101
    try, 101
    while, 33
sentinel (*atributo de multiprocessing.Process*), 842
sentinel (*en el módulo unittest.mock*), 1632
sep (*en el módulo os*), 654
sequence
    iteration, 41
    objeto, 42
    types, immutable, 44
    types, mutable, 44
    types, operations on, 42, 44
Sequence (*clase en collections.abc*), 260
Sequence (*clase en typing*), 1537
sequence (*en el módulo msilib*), 2021
SequenceMatcher (*clase en difflib*), 149
serializing
    objects, 465
serve_forever() (*método de asyncio.Server*), 971
serve_forever() (*método de socketserver.BaseServer*), 1335
server
    WWW, 1341, 1997
server (*atributo de http.server.BaseHTTPRequestHandler*), 1341
Server (*clase en asyncio*), 971
server_activate() (*método de socketserver.BaseServer*), 1336
server_address (*atributo de socketserver.BaseServer*), 1336
server_bind() (*método de socketserver.BaseServer*), 1337
server_close() (*método de socketserver.BaseServer*), 1335
server_hostname (*atributo de ssl.SSLSocket*), 1050
server_side (*atributo de ssl.SSLSocket*), 1050
server_software (*atributo de wsgiref.handlers.BaseHandler*), 1267
server_version (*atributo de http.server.BaseHTTPRequestHandler*), 1342
server_version (*atributo de http.server.SimpleHTTPRequestHandler*), 1344
ServerProxy (*clase en xmlrpc.client*), 1359
service_actions() (*método de socketserver.BaseServer*), 1335
session (*atributo de ssl.SSLSocket*), 1050
session_reused (*atributo de ssl.SSLSocket*), 1050

session_stats() (*método de ssl.SSLContext*), 1056
set
    objeto, 81
Set (*clase en ast*), 1894
Set (*clase en collections.abc*), 260
Set (*clase en typing*), 1534
set (*clase incorporada*), 81
Set Breakpoint, 1501
set() (*método de asyncio.Event*), 942
set() (*método de configparser.ConfigParser*), 577
set() (*método de configparser.RawConfigParser*), 579
set() (*método de contextvars.ContextVar*), 913
set() (*método de http.cookies.Morsel*), 1348
set() (*método de ossaudiodev.oss_mixer_device*), 2059
set() (*método de test.support.os_helper.EnvironmentVarGuard*), 1681
set() (*método de threading.Event*), 831
set() (*método de tkinter.ttk.Combobox*), 1479
set() (*método de tkinter.ttk.Spinbox*), 1480
set() (*método de tkinter.ttk.Treeview*), 1489
set() (*método de xml.etree.ElementTree.Element*), 1211
SET_ADD (*opcode*), 1946
set_allowed_domains() (*método de http.cookiejar.DefaultCookiePolicy*), 1356
set_alpn_protocols() (*método de ssl.SSLContext*), 1053
set_app() (*método de wsgiref.simple_server.WSGIServer*), 1264
set_asyncgen_hooks() (*en el módulo sys*), 1765
set_authorizer() (*método de sqlite3.Connection*), 498
set_auto_history() (*en el módulo readline*), 163
set_blocked_domains() (*método de http.cookiejar.DefaultCookiePolicy*), 1356
set_blocking() (*en el módulo os*), 616
set_boundary() (*método de email.message.EmailMessage*), 1099
set_boundary() (*método de email.message.Message*), 1138
set_break() (*método de bdb.Bdb*), 1692
set_charset() (*método de email.message.Message*), 1134
set_child_watcher() (*en el módulo asyncio*), 993
set_child_watcher() (*método de asyncio.AbstractEventLoopPolicy*), 993
set_children() (*método de tkinter.ttk.Treeview*), 1486
set_ciphers() (*método de ssl.SSLContext*), 1053
set_completer() (*en el módulo readline*), 164
set_completer_delims() (*en el módulo readline*), 164
set_completion_display_matches_hook() (*en el módulo readline*), 164
set_content() (*en el módulo email.contentmanager*), 1124
set_content() (*método de email.contentmanager.ContentManager*), 1123
set_content() (*método de email.message.EmailMessage*), 1101
set_continue() (*método de bdb.Bdb*), 1692
set_cookie() (*método de http.cookiejar.CookieJar*), 1352
set_cookie_if_ok() (*método de http.cookiejar.CookieJar*), 1352
set_coroutine_origin_tracking_depth() (*en el módulo sys*), 1765
set_current() (*método de msilib.Feature*), 2019
set_data() (*método de importlib.abc.SourceLoader*), 1871
set_data() (*método de importlib.machinery.SourceFileLoader*), 1876
set_date() (*método de mailbox.MaildirMessage*), 1171
set_debug() (*en el módulo gc*), 1824
set_debug() (*método de asyncio.loop*), 969
set_debuglevel() (*método de ftplib.FTP*), 1310
set_debuglevel() (*método de http.client.HTTPConnection*), 1304
set_debuglevel() (*método de nntplib.NNTP*), 2027
set_debuglevel() (*método de poplib.POP3*), 1314
set_debuglevel() (*método de smtplib.SMTP*), 1325
set_debuglevel() (*método de telnetlib.Telnet*), 2070
set_default_executor() (*método de asyncio.loop*), 967
set_default_type() (*método de email.message.EmailMessage*), 1098
set_default_type() (*método de email.message.Message*), 1136
set_default_verify_paths() (*método de ssl.SSLContext*), 1053
set_defaults() (*método de argparse.ArgumentParser*), 708
set_defaults() (*método de optparse.OptionParser*), 2048
set_ecdh_curve() (*método de ssl.SSLContext*), 1054
set_errno() (*en el módulo ctypes*), 814
set_escdelay() (*en el módulo curses*), 761
set_event_loop() (*en el módulo asyncio*), 954
set_event_loop() (*método de asyncio.AbstractEventLoopPolicy*), 992
set_event_loop_policy() (*en el módulo asyncio*), 992
set_exception() (*método de asyncio.Future*), 976
set_exception() (*método de concurrent.futures.Future*), 887
set_exception_handler() (*método de asyncio.loop*), 968
set_executable() (*en el módulo multiprocessing*), 848
set_filter() (*método de tkin-*

ter.filedialog.FileDialog), 1471
set_flags() (método de mailbox.MaildirMessage), 1171
set_flags() (método de mailbox.mboxMessage), 1172
set_flags() (método de mailbox.MMDFMessage), 1176
set_from() (método de mailbox.mboxMessage), 1172
set_from() (método de mailbox.MMDFMessage), 1176
set_handle_inheritable() (en el módulo os), 619
set_history_length() (en el módulo readline), 163
set_info() (método de mailbox.MaildirMessage), 1171
set_inheritable() (en el módulo os), 619
set_inheritable() (método de socket.socket), 1028
set_int_max_str_digits() (en el módulo sys), 1763
set_labels() (método de mailbox.BabylMessage), 1175
set_last_error() (en el módulo ctypes), 814
set_literal (2to3 fixer), 1664
set_loader() (en el módulo importlib.util), 1880
set_match_tests() (en el módulo test.support), 1671
set_memlimit() (en el módulo test.support), 1671
set_name() (método de asyncio.Task), 933
set_next() (método de bdb.Bdb), 1692
set_nonstandard_attr() (método de http.cookiejar.Cookie), 1358
set_npn_protocols() (método de ssl.SSLContext), 1053
set_ok() (método de http.cookiejar.CookiePolicy), 1354
set_option_negotiation_callback() (método de telnetlib.Telnet), 2071
set_output_charset() (método de gettext.NullTranslations), 1395
set_package() (en el módulo importlib.util), 1880
set_param() (método de email.message.EmailMessage), 1098
set_param() (método de email.message.Message), 1137
set_pasv() (método de ftplib.FTP), 1311
set_payload() (método de email.message.Message), 1134
set_policy() (método de http.cookiejar.CookieJar), 1352
set_position() (método de xdrlib.Unpacker), 2073
set_pre_input_hook() (en el módulo readline), 164
set_progress_handler() (método de sqlite3.Connection), 498
set_protocol() (método de asyncio.BaseTransport), 981
set_proxy() (método de urllib.request.Request), 1276
set_quit() (método de bdb.Bdb), 1692
set_recsrc() (método de ossaudiodev.oss_mixer_device), 2059
set_result() (método de asyncio.Future), 976
set_result() (método de concurrent.futures.Future), 887
set_return() (método de bdb.Bdb), 1692
set_running_or_notify_cancel() (método de concurrent.futures.Future), 887
set_selection() (método de tkinter.filedialog.FileDialog), 1471
set_seq1() (método de difflib.SequenceMatcher), 149
set_seq2() (método de difflib.SequenceMatcher), 149
set_seqs() (método de difflib.SequenceMatcher), 149
set_sequences() (método de mailbox.MH), 1167
set_sequences() (método de mailbox.MHMessage), 1174
set_server_documentation() (método de xmlrpc.server.DocCGIXMLRPCRequestHandler), 1372
set_server_documentation() (método de xmlrpc.server.DocXMLRPCServer), 1372
set_server_name() (método de xmlrpc.server.DocCGIXMLRPCRequestHandler), 1372
set_server_name() (método de xmlrpc.server.DocXMLRPCServer), 1371
set_server_title() (método de xmlrpc.server.DocCGIXMLRPCRequestHandler), 1372
set_server_title() (método de xmlrpc.server.DocXMLRPCServer), 1371
set_servername_callback (atributo de ssl.SSLContext), 1054
set_start_method() (en el módulo multiprocessing), 848
set_startup_hook() (en el módulo readline), 164
set_step() (método de bdb.Bdb), 1692
set_subdir() (método de mailbox.MaildirMessage), 1171
set_tabsize() (en el módulo curses), 761
set_task_factory() (método de asyncio.loop), 958
set_terminator() (método de asynchat.async_chat), 1989
set_threshold() (en el módulo gc), 1824
set_trace() (en el módulo bdb), 1694
set_trace() (en el módulo pdb), 1698
set_trace() (método de bdb.Bdb), 1692
set_trace() (método de pdb.Pdb), 1698
set_trace_callback() (método de sqlite3.Connection), 498
set_tunnel() (método de http.client.HTTPConnection), 1304
set_type() (método de email.message.Message), 1138

set_unittest_reportflags() (*en el módulo doctest*), 1562
set_unixfrom() (*método de email.message.EmailMessage*), 1096
set_unixfrom() (*método de email.message.Message*), 1133
set_until() (*método de bdb.Bdb*), 1692
SET_UPDATE (*opcode*), 1949
set_url() (*método de urllib.robotparser.RobotFileParser*), 1298
set_usage() (*método de optparse.OptionParser*), 2048
set_userptr() (*método de curses.panel.Panel*), 778
set_visible() (*método de mailbox.BabylMessage*), 1175
set_wakeup_fd() (*en el módulo signal*), 1083
set_write_buffer_limits() (*método de asyncio.WriteTransport*), 982
setacl() (*método de imaplib.IMAP4*), 1321
setannotation() (*método de imaplib.IMAP4*), 1321
setattr()
    función incorporada, 23
setAttribute() (*método de xml.dom.Element*), 1224
setAttributeNode() (*método de xml.dom.Element*), 1224
setAttributeNodeNS() (*método de xml.dom.Element*), 1224
setAttributeNS() (*método de xml.dom.Element*), 1224
SetBase() (*método de xml.parsers.expat.xmlparser*), 1248
setblocking() (*método de socket.socket*), 1028
setByteStream() (*método de xml.sax.xmlreader.InputSource*), 1246
setcbreak() (*en el módulo tty*), 1975
setCharacterStream() (*método de xml.sax.xmlreader.InputSource*), 1246
SetComp (*clase en ast*), 1899
setcomptype() (*método de aifc.aifc*), 1987
setcomptype() (*método de sunau.AU_write*), 2068
setcomptype() (*método de wave.Wave_write*), 1389
setContentHandler() (*método de xml.sax.xmlreader.XMLReader*), 1244
setcontext() (*en el módulo decimal*), 342
setDaemon() (*método de threading.Thread*), 826
setdefault() (*método de dict*), 85
setdefault() (*método de http.cookies.Morsel*), 1349
setdefaulttimeout() (*en el módulo socket*), 1021
setdlopenflags() (*en el módulo sys*), 1763
setDocumentLocator() (*método de xml.sax.handler.ContentHandler*), 1239
setDTDHandler() (*método de xml.sax.xmlreader.XMLReader*), 1244
setegid() (*en el módulo os*), 607
setEncoding() (*método de xml.sax.xmlreader.InputSource*), 1246
setEntityResolver() (*método de xml.sax.xmlreader.XMLReader*), 1244
setErrorHandler() (*método de xml.sax.xmlreader.XMLReader*), 1244
seteuid() (*en el módulo os*), 607
setFeature() (*método de xml.sax.xmlreader.XMLReader*), 1245
setfirstweekday() (*en el módulo calendar*), 237
setfmt() (*método de ossaudiodev.oss_audio_device*), 2057
setFormatter() (*método de logging.Handler*), 719
setframerate() (*método de aifc.aifc*), 1987
setframerate() (*método de sunau.AU_write*), 2068
setframerate() (*método de wave.Wave_write*), 1389
setgid() (*en el módulo os*), 607
setgroups() (*en el módulo os*), 607
seth() (*en el módulo turtle*), 1415
setheading() (*en el módulo turtle*), 1415
sethostname() (*en el módulo socket*), 1021
setinputsizes() (*método de sqlite3.Cursor*), 503
SetInteger() (*método de msilib.Record*), 2018
setitem() (*en el módulo operator*), 408
setitimer() (*en el módulo signal*), 1083
setLevel() (*método de logging.Handler*), 719
setLevel() (*método de logging.Logger*), 715
setlocale() (*en el módulo locale*), 1400
setLocale() (*método de xml.sax.xmlreader.XMLReader*), 1244
setLoggerClass() (*en el módulo logging*), 729
setlogmask() (*en el módulo syslog*), 1984
setLogRecordFactory() (*en el módulo logging*), 729
setmark() (*método de aifc.aifc*), 1987
setMaxConns() (*método de urllib.request.CacheFTPHandler*), 1282
setmode() (*en el módulo msvcrt*), 1958
setName() (*método de threading.Thread*), 825
setnchannels() (*método de aifc.aifc*), 1986
setnchannels() (*método de sunau.AU_write*), 2068
setnchannels() (*método de wave.Wave_write*), 1389
setnframes() (*método de aifc.aifc*), 1987
setnframes() (*método de sunau.AU_write*), 2068
setnframes() (*método de wave.Wave_write*), 1389
setoutputsize() (*método de sqlite3.Cursor*), 503
SetParamEntityParsing() (*método de xml.parsers.expat.xmlparser*), 1249
setparameters() (*método de ossaudiodev.oss_audio_device*), 2058
setparams() (*método de aifc.aifc*), 1987
setparams() (*método de sunau.AU_write*), 2068
setparams() (*método de wave.Wave_write*), 1389
setpassword() (*método de zipfile.ZipFile*), 533
setpgid() (*en el módulo os*), 607
setpgrp() (*en el módulo os*), 607
setpos() (*en el módulo turtle*), 1415
setpos() (*método de aifc.aifc*), 1986

setpos() (*método de sunau.AU_read*), 2067
setpos() (*método de wave.Wave_read*), 1388
setposition() (*en el módulo turtle*), 1415
setpriority() (*en el módulo os*), 607
setprofile() (*en el módulo sys*), 1763
setprofile() (*en el módulo threading*), 823
SetProperty() (*método de msilib.SummaryInformation*), 2017
setProperty() (*método de xml.sax.xmlreader.XMLReader*), 1245
setPublicId() (*método de xml.sax.xmlreader.InputSource*), 1246
setquota() (*método de imaplib.IMAP4*), 1321
setraw() (*en el módulo tty*), 1975
setrecursionlimit() (*en el módulo sys*), 1764
setregid() (*en el módulo os*), 607
setresgid() (*en el módulo os*), 608
setresuid() (*en el módulo os*), 608
setreuid() (*en el módulo os*), 608
setrlimit() (*en el módulo resource*), 1979
setsampwidth() (*método de aifc.aifc*), 1987
setsampwidth() (*método de sunau.AU_write*), 2068
setsampwidth() (*método de wave.Wave_write*), 1389
setscrreg() (*método de curses.window*), 769
setsid() (*en el módulo os*), 608
setsockopt() (*método de socket.socket*), 1028
setstate() (*en el módulo random*), 361
setstate() (*método de codecs.IncrementalDecoder*), 181
setstate() (*método de codecs.IncrementalEncoder*), 180
setStream() (*método de logging.StreamHandler*), 742
SetStream() (*método de msilib.Record*), 2018
SetString() (*método de msilib.Record*), 2018
setswitchinterval() (*en el módulo sys*), 1764
setswitchinterval() (*en el módulo test.support*), 1671
setSystemId() (*método de xml.sax.xmlreader.InputSource*), 1246
setsyx() (*en el módulo curses*), 762
setTarget() (*método de logging.handlers.MemoryHandler*), 752
settiltangle() (*en el módulo turtle*), 1426
settimeout() (*método de socket.socket*), 1028
setTimeout() (*método de urllib.request.CacheFTPHandler*), 1282
settrace() (*en el módulo sys*), 1764
settrace() (*en el módulo threading*), 823
setuid() (*en el módulo os*), 608
setundobuffer() (*en el módulo turtle*), 1429
setup() (*en el módulo turtle*), 1436
setup() (*método de socketserver.BaseRequestHandler*), 1337
setUp() (*método de unittest.TestCase*), 1580
--setup=S
    timeitopción de línea de comando, 1714
SETUP_ANNOTATIONS (*opcode*), 1947
SETUP_ASYNC_WITH (*opcode*), 1946
setup_environ() (*método de wsgiref.handlers.BaseHandler*), 1267
SETUP_FINALLY (*opcode*), 1950
setup_python() (*método de venv.EnvBuilder*), 1738
setup_scripts() (*método de venv.EnvBuilder*), 1738
setup_testing_defaults() (*en el módulo wsgiref.util*), 1261
SETUP_WITH (*opcode*), 1947
setUpClass() (*método de unittest.TestCase*), 1581
setupterm() (*en el módulo curses*), 762
SetValue() (*en el módulo winreg*), 1963
SetValueEx() (*en el módulo winreg*), 1963
setworldcoordinates() (*en el módulo turtle*), 1431
setx() (*en el módulo turtle*), 1415
setxattr() (*en el módulo os*), 640
sety() (*en el módulo turtle*), 1415
SF_APPEND (*en el módulo stat*), 443
SF_ARCHIVED (*en el módulo stat*), 443
SF_IMMUTABLE (*en el módulo stat*), 443
SF_MNOWAIT (*en el módulo os*), 616
SF_NODISKIO (*en el módulo os*), 616
SF_NOUNLINK (*en el módulo stat*), 443
SF_SNAPSHOT (*en el módulo stat*), 443
SF_SYNC (*en el módulo os*), 616
shape (*atributo de memoryview*), 80
Shape (*clase en turtle*), 1437
shape() (*en el módulo turtle*), 1425
shapesize() (*en el módulo turtle*), 1425
shapetransform() (*en el módulo turtle*), 1427
share() (*método de socket.socket*), 1028
ShareableList (*clase en multiprocessing.shared_memory*), 880
ShareableList() (*método de multiprocessing.managers.SharedMemoryManager*), 880
Shared Memory, 877
shared_ciphers() (*método de ssl.SSLSocket*), 1049
SharedMemory (*clase en multiprocessing.shared_memory*), 877
SharedMemory() (*método de multiprocessing.managers.SharedMemoryManager*), 880
SharedMemoryManager (*clase en multiprocessing.managers*), 879
shearfactor() (*en el módulo turtle*), 1426
Shelf (*clase en shelve*), 483
shelve
    módulo, 482, 485
shield() (*en el módulo asyncio*), 926
shift() (*método de decimal.Context*), 347
shift() (*método de decimal.Decimal*), 342

shift_path_info() (*en el módulo wsgiref.util*), 1261
shifting
    operations, 36
shlex
    módulo, 1448
shlex (*clase en shlex*), 1449
shm (*atributo de multiprocessing.shared_memory.ShareableList*), 881
SHORT_TIMEOUT (*en el módulo test.support*), 1669
shortDescription() (*método de unittest.TestCase*), 1588
shorten() (*en el módulo textwrap*), 155
shouldFlush() (*método de logging.handlers.BufferingHandler*), 752
shouldFlush() (*método de logging.handlers.MemoryHandler*), 752
shouldStop (*atributo de unittest.TestResult*), 1595
show() (*método de curses.panel.Panel*), 778
show() (*método de tkinter.commondialog.Dialog*), 1472
show_code() (*en el módulo dis*), 1941
showerror() (*en el módulo tkinter.messagebox*), 1473
showinfo() (*en el módulo tkinter.messagebox*), 1472
showsyntaxerror() (*método de code.InteractiveInterpreter*), 1850
showtraceback() (*método de code.InteractiveInterpreter*), 1850
showturtle() (*en el módulo turtle*), 1424
showwarning() (*en el módulo tkinter.messagebox*), 1473
showwarning() (*en el módulo warnings*), 1784
shuffle() (*en el módulo random*), 363
shutdown() (*en el módulo logging*), 729
shutdown() (*método de concurrent.futures.Executor*), 883
shutdown() (*método de imaplib.IMAP4*), 1321
shutdown() (*método de multiprocessing.managers.BaseManager*), 856
shutdown() (*método de socketserver.BaseServer*), 1335
shutdown() (*método de socket.socket*), 1028
shutdown_asyncgens() (*método de asyncio.loop*), 955
shutdown_default_executor() (*método de asyncio.loop*), 956
shutil
    módulo, 454
side_effect (*atributo de unittest.mock.Mock*), 1608
SIG_BLOCK (*en el módulo signal*), 1081
SIG_DFL (*en el módulo signal*), 1079
SIG_IGN (*en el módulo signal*), 1079
SIG_SETMASK (*en el módulo signal*), 1081
SIG_UNBLOCK (*en el módulo signal*), 1081
SIGABRT (*en el módulo signal*), 1079
SIGALRM (*en el módulo signal*), 1079
SIGBREAK (*en el módulo signal*), 1079
SIGBUS (*en el módulo signal*), 1079
SIGCHLD (*en el módulo signal*), 1079
SIGCLD (*en el módulo signal*), 1079
SIGCONT (*en el módulo signal*), 1079
SIGFPE (*en el módulo signal*), 1079
SIGHUP (*en el módulo signal*), 1079
SIGILL (*en el módulo signal*), 1080
SIGINT (*en el módulo signal*), 1080
siginterrupt() (*en el módulo signal*), 1083
SIGKILL (*en el módulo signal*), 1080
signal
    módulo, 918, 1078
signal() (*en el módulo signal*), 1083
signature (*atributo de inspect.BoundArguments*), 1836
Signature (*clase en inspect*), 1833
signature() (*en el módulo inspect*), 1832
sigpending() (*en el módulo signal*), 1084
SIGPIPE (*en el módulo signal*), 1080
SIGSEGV (*en el módulo signal*), 1080
SIGTERM (*en el módulo signal*), 1080
sigtimedwait() (*en el módulo signal*), 1084
SIGUSR1 (*en el módulo signal*), 1080
SIGUSR2 (*en el módulo signal*), 1080
sigwait() (*en el módulo signal*), 1084
sigwaitinfo() (*en el módulo signal*), 1084
SIGWINCH (*en el módulo signal*), 1080
Simple Mail Transfer Protocol, 1323
SimpleCookie (*clase en http.cookies*), 1347
simplefilter() (*en el módulo warnings*), 1784
SimpleHandler (*clase en wsgiref.handlers*), 1266
SimpleHTTPRequestHandler (*clase en http.server*), 1344
SimpleNamespace (*clase en types*), 285
SimpleQueue (*clase en multiprocessing*), 846
SimpleQueue (*clase en queue*), 910
SimpleXMLRPCRequestHandler (*clase en xmlrpc.server*), 1367
SimpleXMLRPCServer (*clase en xmlrpc.server*), 1366
sin() (*en el módulo cmath*), 329
sin() (*en el módulo math*), 325
SingleAddressHeader (*clase en email.headerregistry*), 1120
singledispatch() (*en el módulo functools*), 401
singledispatchmethod (*clase en functools*), 403
sinh() (*en el módulo cmath*), 329
sinh() (*en el módulo math*), 326
SIO_KEEPALIVE_VALS (*en el módulo socket*), 1014
SIO_LOOPBACK_FAST_PATH (*en el módulo socket*), 1014
SIO_RCVALL (*en el módulo socket*), 1014
site
    módulo, 1844
site_maps() (*método de urllib.robotparser.RobotFileParser*), 1298
sitecustomize
    módulo, 1845
siteopción de línea de comando

--user-base, 1847
--user-site, 1847
site-packages
directory, 1844
sixtofour (*atributo de ipaddress.IPv6Address*), 1376
size (*atributo de multiprocessing.shared_memory.SharedMemory*), 878
size (*atributo de struct.Struct*), 173
size (*atributo de tarfile.TarInfo*), 546
size (*atributo de tracemalloc.Statistic*), 1727
size (*atributo de tracemalloc.StatisticDiff*), 1728
size (*atributo de tracemalloc.Trace*), 1728
size() (*método de ftplib.FTP*), 1312
size() (*método de mmap.mmap*), 1090
size_diff (*atributo de tracemalloc.StatisticDiff*), 1728
Sized (*clase en collections.abc*), 260
Sized (*clase en typing*), 1538
sizeof() (*en el módulo ctypes*), 814
SKIP (*en el módulo doctest*), 1556
skip() (*en el módulo unittest*), 1579
skip() (*método de chunk.Chunk*), 2006
skip_if_broken_multiprocessing_synchronize() (*en el módulo test.support*), 1676
skip_unless_bind_unix_socket() (*en el módulo test.support.socket_helper*), 1677
skip_unless_symlink() (*en el módulo test.support.os_helper*), 1681
skip_unless_xattr() (*en el módulo test.support.os_helper*), 1681
skipIf() (*en el módulo unittest*), 1579
skipinitialspace (*atributo de csv.Dialect*), 560
skipped (*atributo de unittest.TestResult*), 1595
skippedEntity() (*método de xml.sax.handler.ContentHandler*), 1240
SkipTest, 1579
skipTest() (*método de unittest.TestCase*), 1581
skipUnless() (*en el módulo unittest*), 1579
SLASH (*en el módulo token*), 1925
SLASHEQUAL (*en el módulo token*), 1926
slave() (*método de nntplib.NNTP*), 2027
sleep() (*en el módulo asyncio*), 925
sleep() (*en el módulo time*), 673
slice
assignment, 44
función incorporada, 1952
operation, 42
Slice (*clase en ast*), 1899
slice (*clase incorporada*), 24
SMALLEST (*en el módulo test.support*), 1670
SMTP
protocol, 1323
SMTP (*clase en smtplib*), 1323
SMTP (*en el módulo email.policy*), 1114
smtp_server (*atributo de smtpd.SMTPChannel*), 2063
SMTP_SSL (*clase en smtplib*), 1324
smtp_state (*atributo de smtpd.SMTPChannel*), 2063
SMTPAuthenticationError, 1325
SMTPChannel (*clase en smtpd*), 2063
SMTPConnectError, 1325
smtpd
módulo, 2061
SMTPDataError, 1325
SMTPException, 1324
SMTPHandler (*clase en logging.handlers*), 751
SMTPHeloError, 1325
smtplib
módulo, 1323
SMTPNotSupportedError, 1325
SMTPRecipientsRefused, 1324
SMTPResponseException, 1324
SMTPSenderRefused, 1324
SMTPServer (*clase en smtpd*), 2061
SMTPServerDisconnected, 1324
SMTPUTF8 (*en el módulo email.policy*), 1114
Snapshot (*clase en tracemalloc*), 1726
SND_ALIAS (*en el módulo winsound*), 1968
SND_ASYNC (*en el módulo winsound*), 1969
SND_FILENAME (*en el módulo winsound*), 1968
SND_LOOP (*en el módulo winsound*), 1968
SND_MEMORY (*en el módulo winsound*), 1968
SND_NODEFAULT (*en el módulo winsound*), 1969
SND_NOSTOP (*en el módulo winsound*), 1969
SND_NOWAIT (*en el módulo winsound*), 1969
SND_PURGE (*en el módulo winsound*), 1969
sndhdr
módulo, 2064
sni_callback (*atributo de ssl.SSLContext*), 1053
sniff() (*método de csv.Sniffer*), 558
Sniffer (*clase en csv*), 558
sock_accept() (*método de asyncio.loop*), 965
SOCK_CLOEXEC (*en el módulo socket*), 1012
sock_connect() (*método de asyncio.loop*), 964
SOCK_DGRAM (*en el módulo socket*), 1012
SOCK_MAX_SIZE (*en el módulo test.support*), 1669
SOCK_NONBLOCK (*en el módulo socket*), 1012
SOCK_RAW (*en el módulo socket*), 1012
SOCK_RDM (*en el módulo socket*), 1012
sock_recv() (*método de asyncio.loop*), 964
sock_recv_into() (*método de asyncio.loop*), 964
sock_sendall() (*método de asyncio.loop*), 964
sock_sendfile() (*método de asyncio.loop*), 965
SOCK_SEQPACKET (*en el módulo socket*), 1012
SOCK_STREAM (*en el módulo socket*), 1012
socket
módulo, 1008, 1257
objeto, 1008
socket (*atributo de socketserver.BaseServer*), 1336
socket (*clase en socket*), 1015
socket() (*in module socket*), 1069
socket() (*método de imaplib.IMAP4*), 1321
socket_type (*atributo de socketserver.BaseServer*), 1336
SocketHandler (*clase en logging.handlers*), 747
socketpair() (*en el módulo socket*), 1016

sockets (*atributo de asyncio.Server*), 972
socketserver
    módulo, 1333
SocketType (*en el módulo socket*), 1017
SOFT_KEYWORD (*en el módulo token*), 1926
softkwlist (*en el módulo keyword*), 1927
SOL_ALG (*en el módulo socket*), 1014
SOL_RDS (*en el módulo socket*), 1014
SOMAXCONN (*en el módulo socket*), 1012
sort() (*método de imaplib.IMAP4*), 1321
sort() (*método de list*), 45
sort_stats() (*método de pstats.Stats*), 1707
sortdict() (*en el módulo test.support*), 1670
sorted()
    función incorporada, 24
--sort-keys
    json.toolopción de línea de comando, 1161
sortTestMethodsUsing (*atributo de unittest.TestLoader*), 1594
source (*atributo de doctest.Example*), 1564
source (*atributo de shlex.shlex*), 1451
source (*pdb command*), 1701
SOURCE_DATE_EPOCH, 1935, 1937
source_from_cache() (*en el módulo imp*), 2012
source_from_cache() (*en el módulo importlib.util*), 1879
source_hash() (*en el módulo importlib.util*), 1881
SOURCE_SUFFIXES (*en el módulo importlib.machinery*), 1874
source_to_code() (*método estático de importlib.abc.InspectLoader*), 1869
SourceFileLoader (*clase en importlib.machinery*), 1876
sourcehook() (*método de shlex.shlex*), 1450
SourcelessFileLoader (*clase en importlib.machinery*), 1876
SourceLoader (*clase en importlib.abc*), 1870
space
    in printf-style formatting, 58, 73
    in string formatting, 117
span() (*método de re.Match*), 138
spawn() (*en el módulo pty*), 1975
spawn_python() (*en el módulo test.support.script_helper*), 1678
spawnl() (*en el módulo os*), 646
spawnle() (*en el módulo os*), 646
spawnlp() (*en el módulo os*), 646
spawnlpe() (*en el módulo os*), 646
spawnv() (*en el módulo os*), 646
spawnve() (*en el módulo os*), 646
spawnvp() (*en el módulo os*), 646
spawnvpe() (*en el módulo os*), 646
spec_from_file_location() (*en el módulo importlib.util*), 1880
spec_from_loader() (*en el módulo importlib.util*), 1880
special
    method, 2091
SpecialFileError, 541
specified_attributes (*atributo de xml.parsers.expat.xmlparser*), 1249
speed() (*en el módulo turtle*), 1418
speed() (*método de ossaudiodev.oss_audio_device*), 2057
Spinbox (*clase en tkinter.ttk*), 1480
splice() (*en el módulo os*), 617
SPLICE_F_MORE (*en el módulo os*), 617
SPLICE_F_MOVE (*en el módulo os*), 617
SPLICE_F_NONBLOCK (*en el módulo os*), 617
split() (*en el módulo os.path*), 435
split() (*en el módulo re*), 132
split() (*en el módulo shlex*), 1448
split() (*método de bytearray*), 67
split() (*método de bytes*), 67
split() (*método de re.Pattern*), 136
split() (*método de str*), 54
splitdrive() (*en el módulo os.path*), 436
splitext() (*en el módulo os.path*), 436
splitlines() (*método de bytearray*), 70
splitlines() (*método de bytes*), 70
splitlines() (*método de str*), 55
SplitResult (*clase en urllib.parse*), 1295
SplitResultBytes (*clase en urllib.parse*), 1295
SpooledTemporaryFile (*clase en tempfile*), 447
sprintf-style formatting, 57, 72
spwd
    módulo, 2065
sqlite3
    módulo, 491
SQLITE_DENY (*en el módulo sqlite3*), 495
SQLITE_IGNORE (*en el módulo sqlite3*), 495
SQLITE_OK (*en el módulo sqlite3*), 495
sqlite_version (*en el módulo sqlite3*), 495
sqlite_version_info (*en el módulo sqlite3*), 495
sqrt() (*en el módulo cmath*), 329
sqrt() (*en el módulo math*), 325
sqrt() (*método de decimal.Context*), 348
sqrt() (*método de decimal.Decimal*), 342
ssl
    módulo, 1033
SSL, 1033
ssl_version (*atributo de ftplib.FTP_TLS*), 1313
SSLCertVerificationError, 1036
SSLContext (*clase en ssl*), 1050
SSLEOFError, 1036
SSLError, 1036
SSLErrorNumber (*clase en ssl*), 1046
SSLKEYLOGFILE, 1035
SSLObject (*clase en ssl*), 1063
sslobject_class (*atributo de ssl.SSLContext*), 1056
SSLSession (*clase en ssl*), 1065
SSLSocket (*clase en ssl*), 1046
sslsocket_class (*atributo de ssl.SSLContext*), 1055
SSLSyscallError, 1036
SSLv3 (*atributo de ssl.TLSVersion*), 1046

SSLWantReadError, 1036
SSLWantWriteError, 1036
SSLZeroReturnError, 1036
st() (*en el módulo turtle*), 1424
st_atime (*atributo de os.stat_result*), 631
ST_ATIME (*en el módulo stat*), 441
st_atime_ns (*atributo de os.stat_result*), 631
st_birthtime (*atributo de os.stat_result*), 632
st_blksize (*atributo de os.stat_result*), 632
st_blocks (*atributo de os.stat_result*), 632
st_creator (*atributo de os.stat_result*), 632
st_ctime (*atributo de os.stat_result*), 631
ST_CTIME (*en el módulo stat*), 441
st_ctime_ns (*atributo de os.stat_result*), 631
st_dev (*atributo de os.stat_result*), 631
ST_DEV (*en el módulo stat*), 441
st_file_attributes (*atributo de os.stat_result*), 632
st_flags (*atributo de os.stat_result*), 632
st_fstype (*atributo de os.stat_result*), 632
st_gen (*atributo de os.stat_result*), 632
st_gid (*atributo de os.stat_result*), 631
ST_GID (*en el módulo stat*), 441
st_ino (*atributo de os.stat_result*), 631
ST_INO (*en el módulo stat*), 441
st_mode (*atributo de os.stat_result*), 631
ST_MODE (*en el módulo stat*), 441
st_mtime (*atributo de os.stat_result*), 631
ST_MTIME (*en el módulo stat*), 441
st_mtime_ns (*atributo de os.stat_result*), 631
st_nlink (*atributo de os.stat_result*), 631
ST_NLINK (*en el módulo stat*), 441
st_rdev (*atributo de os.stat_result*), 632
st_reparse_tag (*atributo de os.stat_result*), 632
st_rsize (*atributo de os.stat_result*), 632
st_size (*atributo de os.stat_result*), 631
ST_SIZE (*en el módulo stat*), 441
st_type (*atributo de os.stat_result*), 632
st_uid (*atributo de os.stat_result*), 631
ST_UID (*en el módulo stat*), 441
stack (*atributo de traceback.TracebackException*), 1818
stack viewer, 1500
stack() (*en el módulo inspect*), 1841
stack_effect() (*en el módulo dis*), 1943
stack_size() (*en el módulo _thread*), 917
stack_size() (*en el módulo threading*), 823
stackable streams, 174
StackSummary (*clase en traceback*), 1819
stamp() (*en el módulo turtle*), 1417
standard_b64decode() (*en el módulo base64*), 1182
standard_b64encode() (*en el módulo base64*), 1182
standarderror (*2to3 fixer*), 1665
standend() (*método de curses.window*), 769
standout() (*método de curses.window*), 769
STAR (*en el módulo token*), 1925
STAREQUAL (*en el módulo token*), 1926
starmap() (*en el módulo itertools*), 390
starmap() (*método de multiprocessing.pool.Pool*), 863
starmap_async() (*método de multiprocessing.pool.Pool*), 863
Starred (*clase en ast*), 1895
start (*atributo de range*), 47
start (*atributo de UnicodeError*), 107
start() (*en el módulo tracemalloc*), 1724
start() (*método de logging.handlers.QueueListener*), 755
start() (*método de multiprocessing.managers.BaseManager*), 855
start() (*método de multiprocessing.Process*), 841
start() (*método de re.Match*), 138
start() (*método de threading.Thread*), 825
start() (*método de tkinter.ttk.Progressbar*), 1483
start() (*método de xml.etree.ElementTree.TreeBuilder*), 1215
start_color() (*en el módulo curses*), 762
start_component() (*método de msilib.Directory*), 2019
start_new_thread() (*en el módulo _thread*), 916
start_ns() (*método de xml.etree.ElementTree.TreeBuilder*), 1215
start_server() (*en el módulo asyncio*), 935
start_serving() (*método de asyncio.Server*), 971
start_threads() (*en el módulo test.support.threading_helper*), 1679
start_tls() (*método de asyncio.loop*), 963
start_unix_server() (*en el módulo asyncio*), 936
startCDATA() (*método de xml.sax.handler.LexicalHandler*), 1242
StartCdataSectionHandler() (*método de xml.parsers.expat.xmlparser*), 1251
--start-directory directory unittest-discoveropción de línea de comando, 1575
StartDoctypeDeclHandler() (*método de xml.parsers.expat.xmlparser*), 1250
startDocument() (*método de xml.sax.handler.ContentHandler*), 1239
startDTD() (*método de xml.sax.handler.LexicalHandler*), 1242
startElement() (*método de xml.sax.handler.ContentHandler*), 1239
StartElementHandler() (*método de xml.parsers.expat.xmlparser*), 1250
startElementNS() (*método de xml.sax.handler.ContentHandler*), 1240
STARTF_USESHOWWINDOW (*en el módulo subprocess*), 901
STARTF_USESTDHANDLES (*en el módulo subprocess*), 901
startfile() (*en el módulo os*), 648
StartNamespaceDeclHandler() (*método de xml.parsers.expat.xmlparser*), 1251

startPrefixMapping() (*método de xml.sax.handler.ContentHandler*), 1239
startswith() (*método de bytearray*), 65
startswith() (*método de bytes*), 65
startswith() (*método de str*), 56
startTest() (*método de unittest.TestResult*), 1595
startTestRun() (*método de unittest.TestResult*), 1596
starttls() (*método de imaplib.IMAP4*), 1321
starttls() (*método de nntplib.NNTP*), 2024
starttls() (*método de smtplib.SMTP*), 1327
STARTUPINFO (*clase en subprocess*), 900
stat
    módulo, 439, 630
stat() (*en el módulo os*), 630
stat() (*método de nntplib.NNTP*), 2026
stat() (*método de os.DirEntry*), 630
stat() (*método de pathlib.Path*), 424
stat() (*método de poplib.POP3*), 1315
stat_result (*clase en os*), 631
state() (*método de tkinter.ttk.Widget*), 1478
static_order() (*método de graphlib.TopologicalSorter*), 314
staticmethod()
    función incorporada, 24
Statistic (*clase en tracemalloc*), 1727
StatisticDiff (*clase en tracemalloc*), 1728
statistics
    módulo, 368
statistics() (*método de tracemalloc.Snapshot*), 1727
StatisticsError, 377
Stats (*clase en pstats*), 1707
status (*atributo de http.client.HTTPResponse*), 1306
status (*atributo de urllib.response.addinfourl*), 1288
status() (*método de imaplib.IMAP4*), 1321
statvfs() (*en el módulo os*), 633
STD_ERROR_HANDLE (*en el módulo subprocess*), 901
STD_INPUT_HANDLE (*en el módulo subprocess*), 901
STD_OUTPUT_HANDLE (*en el módulo subprocess*), 901
StdButtonBox (*clase en tkinter.tix*), 1494
stderr (*atributo de asyncio.subprocess.Process*), 948
stderr (*atributo de subprocess.CalledProcessError*), 891
stderr (*atributo de subprocess.CompletedProcess*), 890
stderr (*atributo de subprocess.Popen*), 899
stderr (*atributo de subprocess.TimeoutExpired*), 891
stderr (*en el módulo sys*), 1766
stdev (*atributo de statistics.NormalDist*), 377
stdev() (*en el módulo statistics*), 374
stdin (*atributo de asyncio.subprocess.Process*), 948
stdin (*atributo de subprocess.Popen*), 899
stdin (*en el módulo sys*), 1766
stdlib_module_names (*en el módulo sys*), 1767
stdout (*atributo de asyncio.subprocess.Process*), 948
stdout (*atributo de subprocess.CalledProcessError*), 891
stdout (*atributo de subprocess.CompletedProcess*), 890
stdout (*atributo de subprocess.Popen*), 899
stdout (*atributo de subprocess.TimeoutExpired*), 891
STDOUT (*en el módulo subprocess*), 890
stdout (*en el módulo sys*), 1766
stem (*atributo de pathlib.PurePath*), 420
step (*atributo de range*), 47
step (*pdb command*), 1700
step() (*método de tkinter.ttk.Progressbar*), 1483
stereocontrols() (*método de ossaudiodev.oss_mixer_device*), 2059
stls() (*método de poplib.POP3*), 1315
stop (*atributo de range*), 47
stop() (*en el módulo tracemalloc*), 1725
stop() (*método de asyncio.loop*), 955
stop() (*método de logging.handlers.QueueListener*), 755
stop() (*método de tkinter.ttk.Progressbar*), 1483
stop() (*método de unittest.TestResult*), 1595
stop_here() (*método de bdb.Bdb*), 1692
StopAsyncIteration, 105
StopIteration, 105
stopListening() (*en el módulo logging.config*), 732
stopTest() (*método de unittest.TestResult*), 1596
stopTestRun() (*método de unittest.TestResult*), 1596
storbinary() (*método de ftplib.FTP*), 1311
Store (*clase en ast*), 1895
store() (*método de imaplib.IMAP4*), 1321
STORE_ACTIONS (*atributo de optparse.Option*), 2054
STORE_ATTR (*opcode*), 1948
STORE_DEREF (*opcode*), 1951
STORE_FAST (*opcode*), 1950
STORE_GLOBAL (*opcode*), 1948
STORE_NAME (*opcode*), 1948
STORE_SUBSCR (*opcode*), 1946
storlines() (*método de ftplib.FTP*), 1311
str (*built-in class*)
    (see also string), 48
str (*clase incorporada*), 48
str() (*en el módulo locale*), 1405
strcoll() (*en el módulo locale*), 1404
StreamError, 541
StreamHandler (*clase en logging*), 742
streamreader (*atributo de codecs.CodecInfo*), 174
StreamReader (*clase en asyncio*), 936
StreamReader (*clase en codecs*), 182
StreamReaderWriter (*clase en codecs*), 183
StreamRecoder (*clase en codecs*), 183
StreamRequestHandler (*clase en socketserver*), 1337
streams, 174
    stackable, 174
streamwriter (*atributo de codecs.CodecInfo*), 174
StreamWriter (*clase en asyncio*), 937
StreamWriter (*clase en codecs*), 181
strerror (*atributo de OSError*), 104
strerror() (*en el módulo os*), 608
strftime() (*en el módulo time*), 674
strftime() (*método de datetime.date*), 202

strftime() (*método de datetime.datetime*), 211
strftime() (*método de datetime.time*), 216
strict
    error handler's name, 177
strict (*atributo de csv.Dialect*), 560
strict (*en el módulo email.policy*), 1115
strict_domain (*atributo de http.cookiejar.DefaultCookiePolicy*), 1356
strict_errors() (*en el módulo codecs*), 178
strict_ns_domain (*atributo de http.cookiejar.DefaultCookiePolicy*), 1356
strict_ns_set_initial_dollar (*atributo de http.cookiejar.DefaultCookiePolicy*), 1356
strict_ns_set_path (*atributo de http.cookiejar.DefaultCookiePolicy*), 1356
strict_ns_unverifiable (*atributo de http.cookiejar.DefaultCookiePolicy*), 1356
strict_rfc2965_unverifiable (*atributo de http.cookiejar.DefaultCookiePolicy*), 1356
strides (*atributo de memoryview*), 80
string
    format() (*built-in function*), 13
    formatting, printf, 57
    interpolation, printf, 57
    methods, 49
    módulo, 113, 1405
    objeto, 48
    str (*built-in class*), 48
    str() (*built-in function*), 25
    text sequence type, 48
string (*atributo de re.Match*), 139
STRING (*en el módulo token*), 1924
string_at() (*en el módulo ctypes*), 814
StringIO (*clase en io*), 668
stringprep
    módulo, 160
strip() (*método de bytearray*), 67
strip() (*método de bytes*), 67
strip() (*método de str*), 56
strip_dirs() (*método de pstats.Stats*), 1707
stripspaces (*atributo de curses.textpad.Textbox*), 775
strptime() (*en el módulo time*), 675
strptime() (*método de clase de datetime.datetime*), 206
strsignal() (*en el módulo signal*), 1081
struct
    módulo, 167, 1028
Struct (*clase en struct*), 173
struct_time (*clase en time*), 675
Structure (*clase en ctypes*), 818
structures
    C, 167
strxfrm() (*en el módulo locale*), 1404
Style (*clase en tkinter.ttk*), 1490
Sub (*clase en ast*), 1896
sub() (*en el módulo operator*), 407
sub() (*en el módulo re*), 133
sub() (*método de re.Pattern*), 136
subdirs (*atributo de filecmp.dircmp*), 445
SubElement() (*en el módulo xml.etree.ElementTree*), 1208
submit() (*método de concurrent.futures.Executor*), 882
submodule_search_locations (*atributo de importlib.machinery.ModuleSpec*), 1878
subn() (*en el módulo re*), 134
subn() (*método de re.Pattern*), 136
subnet_of() (*método de ipaddress.IPv4Network*), 1380
subnet_of() (*método de ipaddress.IPv6Network*), 1382
subnets() (*método de ipaddress.IPv4Network*), 1379
subnets() (*método de ipaddress.IPv6Network*), 1382
Subnormal (*clase en decimal*), 350
suboffsets (*atributo de memoryview*), 80
subpad() (*método de curses.window*), 769
subprocess
    módulo, 889
subprocess_exec() (*método de asyncio.loop*), 969
subprocess_shell() (*método de asyncio.loop*), 970
SubprocessError, 891
SubprocessProtocol (*clase en asyncio*), 984
SubprocessTransport (*clase en asyncio*), 980
subscribe() (*método de imaplib.IMAP4*), 1322
subscript
    assignment, 44
    operation, 42
Subscript (*clase en ast*), 1899
subsequent_indent (*atributo de textwrap.TextWrapper*), 157
substitute() (*método de string.Template*), 122
subTest() (*método de unittest.TestCase*), 1581
subtract() (*método de collections.Counter*), 243
subtract() (*método de decimal.Context*), 348
subtype (*atributo de email.headerregistry.ContentTypeHeader*), 1120
subwin() (*método de curses.window*), 769
successful() (*método de multiprocessing.pool.AsyncResult*), 863
suffix (*atributo de pathlib.PurePath*), 419
suffix_map (*atributo de mimetypes.MimeTypes*), 1181
suffix_map (*en el módulo mimetypes*), 1180
suffixes (*atributo de pathlib.PurePath*), 420
suiteClass (*atributo de unittest.TestLoader*), 1594
sum()
    función incorporada, 25
summarize() (*método de doctest.DocTestRunner*), 1567
summarize_address_range() (*en el módulo ipaddress*), 1384
--summary

traceopción de línea de comando, 1717
sunau
módulo, 2066
SUNDAY (*en el módulo calendar*), 238
super (*atributo de pyclbr.Class*), 1934
super (*clase incorporada*), 25
supernet() (*método de ipaddress.IPv4Network*), 1380
supernet() (*método de ipaddress.IPv6Network*), 1382
supernet_of() (*método de ipaddress.IPv4Network*), 1380
supernet_of() (*método de ipaddress.IPv6Network*), 1382
supports_bytes_environ (*en el módulo os*), 608
supports_dir_fd (*en el módulo os*), 633
supports_effective_ids (*en el módulo os*), 634
supports_fd (*en el módulo os*), 634
supports_follow_symlinks (*en el módulo os*), 634
supports_unicode_filenames (*en el módulo os.path*), 436
SupportsAbs (*clase en typing*), 1540
SupportsBytes (*clase en typing*), 1540
SupportsComplex (*clase en typing*), 1540
SupportsFloat (*clase en typing*), 1540
SupportsIndex (*clase en typing*), 1540
SupportsInt (*clase en typing*), 1540
SupportsRound (*clase en typing*), 1540
suppress() (*en el módulo contextlib*), 1799
SuppressCrashReport (*clase en test.support*), 1676
surrogateescape
error handler's name, 177
surrogatepass
error handler's name, 177
SW_HIDE (*en el módulo subprocess*), 901
swap_attr() (*en el módulo test.support*), 1672
swap_item() (*en el módulo test.support*), 1672
swapcase() (*método de bytearray*), 70
swapcase() (*método de bytes*), 70
swapcase() (*método de str*), 56
Symbol (*clase en symtable*), 1923
SymbolTable (*clase en symtable*), 1922
symlink() (*en el módulo os*), 634
symlink_to() (*método de pathlib.Path*), 429
symmetric_difference() (*método de frozenset*), 82
symmetric_difference_update() (*método de frozenset*), 83
symtable
módulo, 1922
symtable() (*en el módulo symtable*), 1922
sync() (*en el módulo os*), 635
sync() (*método de dbm.dumb.dumbdbm*), 490
sync() (*método de dbm.gnu.gdbm*), 489
sync() (*método de ossaudiodev.oss_audio_device*), 2058
sync() (*método de shelve.Shelf*), 483
syncdown() (*método de curses.window*), 769
synchronized() (*en el módulo multiprocessing.sharedctypes*), 854
SyncManager (*clase en multiprocessing.managers*), 856
syncok() (*método de curses.window*), 769
syncup() (*método de curses.window*), 769
SyntaxErr, 1227
SyntaxError, 105
SyntaxWarning, 109
sys
módulo, 20, 1749
sys_exc (*2to3 fixer*), 1665
sys_version (*atributo de http.server.BaseHTTPRequestHandler*), 1342
sysconf() (*en el módulo os*), 654
sysconf_names (*en el módulo os*), 654
sysconfig
módulo, 1769
syslog
módulo, 1983
syslog() (*en el módulo syslog*), 1983
SysLogHandler (*clase en logging.handlers*), 749
system() (*en el módulo os*), 648
system() (*en el módulo platform*), 780
system_alias() (*en el módulo platform*), 780
system_must_validate_cert() (*en el módulo test.support*), 1673
SystemError, 106
SystemExit, 106
systemId (*atributo de xml.dom.DocumentType*), 1222
SystemRandom (*clase en random*), 365
SystemRandom (*clase en secrets*), 597
SystemRoot, 896

T

-t
traceopción de línea de comando, 1717
unittest-discoveropción de línea de comando, 1575
-T
traceopción de línea de comando, 1717
-t <tarfile>
tarfileopción de línea de comando, 551
-t <zipfile>
zipfileopción de línea de comando, 538
T_FMT (*en el módulo locale*), 1402
T_FMT_AMPM (*en el módulo locale*), 1402
--tab
json.toolopción de línea de comando, 1161

tab() (*método de tkinter.ttk.Notebook*), 1482
TabError, 106
tabnanny
    módulo, 1932
tabs() (*método de tkinter.ttk.Notebook*), 1482
tabsize (*atributo de textwrap.TextWrapper*), 157
tabular
    data, 555
tag (*atributo de xml.etree.ElementTree.Element*), 1210
tag_bind() (*método de tkinter.ttk.Treeview*), 1489
tag_configure() (*método de tkinter.ttk.Treeview*), 1489
tag_has() (*método de tkinter.ttk.Treeview*), 1489
tagName (*atributo de xml.dom.Element*), 1224
tail (*atributo de xml.etree.ElementTree.Element*), 1210
take_snapshot() (*en el módulo tracemalloc*), 1725
takewhile() (*en el módulo itertools*), 390
tan() (*en el módulo cmath*), 329
tan() (*en el módulo math*), 326
tanh() (*en el módulo cmath*), 330
tanh() (*en el módulo math*), 326
tar_filter() (*en el módulo tarfile*), 548
TarError, 541
tarfile
    módulo, 539
TarFile (*clase en tarfile*), 542
tarfileopción de línea de comando
    -c <tarfile> <source1> ... <sourceN>, 551
    --create <tarfile> <source1> ... <sourceN>, 551
    -e <tarfile> [<output_dir>], 551
    --extract <tarfile> [<output_dir>], 551
    --filter <filtername>, 551
    -l <tarfile>, 551
    --list <tarfile>, 551
    -t <tarfile>, 551
    --test <tarfile>, 551
    -v, 551
    --verbose, 551
target (*atributo de xml.dom.ProcessingInstruction*), 1226
tarinfo (*atributo de tarfile.FilterError*), 541
TarInfo (*clase en tarfile*), 546
Task (*clase en asyncio*), 931
task_done() (*método de asyncio.Queue*), 950
task_done() (*método de multiprocessing.JoinableQueue*), 846
task_done() (*método de queue.Queue*), 911
tau (*en el módulo cmath*), 330
tau (*en el módulo math*), 327
tb_locals (*atributo de unittest.TestResult*), 1595
tbreak (*pdb command*), 1700
tcdrain() (*en el módulo termios*), 1974
tcflow() (*en el módulo termios*), 1974
tcflush() (*en el módulo termios*), 1974
tcgetattr() (*en el módulo termios*), 1974
tcgetpgrp() (*en el módulo os*), 617
Tcl() (*en el módulo tkinter*), 1457
TCPServer (*clase en socketserver*), 1333
tcsendbreak() (*en el módulo termios*), 1974
tcsetattr() (*en el módulo termios*), 1974
tcsetpgrp() (*en el módulo os*), 617
tearDown() (*método de unittest.TestCase*), 1580
tearDownClass() (*método de unittest.TestCase*), 1581
tee() (*en el módulo itertools*), 390
tell() (*método de aifc.aifc*), 1986
tell() (*método de chunk.Chunk*), 2006
tell() (*método de io.IOBase*), 661
tell() (*método de io.TextIOBase*), 667
tell() (*método de mmap.mmap*), 1090
tell() (*método de sunau.AU_read*), 2067
tell() (*método de sunau.AU_write*), 2068
tell() (*método de wave.Wave_read*), 1388
tell() (*método de wave.Wave_write*), 1389
Telnet (*clase en telnetlib*), 2068
telnetlib
    módulo, 2068
TEMP, 449
temp_cwd() (*en el módulo test.support.os_helper*), 1681
temp_dir() (*en el módulo test.support.os_helper*), 1681
temp_umask() (*en el módulo test.support.os_helper*), 1681
tempdir (*en el módulo tempfile*), 449
tempfile
    módulo, 446
template (*atributo de string.Template*), 122
Template (*clase en pipes*), 2060
Template (*clase en string*), 122
temporary
    file, 446
    file name, 446
temporary (*atributo de bdb.Breakpoint*), 1690
TemporaryDirectory (*clase en tempfile*), 447
TemporaryFile() (*en el módulo tempfile*), 446
teredo (*atributo de ipaddress.IPv6Address*), 1376
TERM, 762
termattrs() (*en el módulo curses*), 762
terminal_size (*clase en os*), 618
terminate() (*método de asyncio.subprocess.Process*), 948
terminate() (*método de asyncio.SubprocessTransport*), 983
terminate() (*método de multiprocessing.pool.Pool*), 863
terminate() (*método de multiprocessing.Process*), 842
terminate() (*método de subprocess.Popen*), 899
terminator (*atributo de logging.StreamHandler*), 742
termios
    módulo, 1974
termname() (*en el módulo curses*), 762

test
    módulo, 1666
test (*atributo de doctest.DocTestFailure*), 1570
test (*atributo de doctest.UnexpectedException*), 1570
--test <tarfile>
    tarfileopción de línea de comando, 551
--test <zipfile>
    zipfileopción de línea de comando, 538
test() (*en el módulo cgi*), 2001
TEST_DATA_DIR (*en el módulo test.support*), 1669
TEST_HOME_DIR (*en el módulo test.support*), 1669
TEST_HTTP_URL (*en el módulo test.support*), 1670
TEST_SUPPORT_DIR (*en el módulo test.support*), 1669
TestCase (*clase en unittest*), 1580
TestFailed, 1668
testfile() (*en el módulo doctest*), 1559
TESTFN (*en el módulo test.support.os_helper*), 1680
TESTFN_NONASCII (*en el módulo test.support.os_helper*), 1680
TESTFN_UNDECODABLE (*en el módulo test.support.os_helper*), 1680
TESTFN_UNENCODABLE (*en el módulo test.support.os_helper*), 1680
TESTFN_UNICODE (*en el módulo test.support.os_helper*), 1680
TestLoader (*clase en unittest*), 1592
testMethodPrefix (*atributo de unittest.TestLoader*), 1594
testmod() (*en el módulo doctest*), 1559
testNamePatterns (*atributo de unittest.TestLoader*), 1594
TestResult (*clase en unittest*), 1594
tests (*en el módulo imghdr*), 2009
testsource() (*en el módulo doctest*), 1569
testsRun (*atributo de unittest.TestResult*), 1595
TestSuite (*clase en unittest*), 1591
test.support
    módulo, 1668
test.support.bytecode_helper
    módulo, 1679
test.support.import_helper
    módulo, 1682
test.support.os_helper
    módulo, 1680
test.support.script_helper
    módulo, 1677
test.support.socket_helper
    módulo, 1677
test.support.threading_helper
    módulo, 1679
test.support.warnings_helper
    módulo, 1683
testzip() (*método de zipfile.ZipFile*), 533
text (*atributo de SyntaxError*), 106
text (*atributo de traceback.TracebackException*), 1818
text (*atributo de xml.etree.ElementTree.Element*), 1210
Text (*clase en typing*), 1536
text (*en el módulo msilib*), 2021
text mode, 20
text() (*en el módulo cgitb*), 2005
text() (*método de msilib.Dialog*), 2020
text_encoding() (*en el módulo io*), 659
text_factory (*atributo de sqlite3.Connection*), 501
Textbox (*clase en curses.textpad*), 774
TextCalendar (*clase en calendar*), 235
textdomain() (*en el módulo gettext*), 1392
textdomain() (*en el módulo locale*), 1407
textinput() (*en el módulo turtle*), 1434
TextIO (*clase en typing*), 1536
TextIOBase (*clase en io*), 666
TextIOWrapper (*clase en io*), 667
TextTestResult (*clase en unittest*), 1596
TextTestRunner (*clase en unittest*), 1597
textwrap
    módulo, 155
TextWrapper (*clase en textwrap*), 156
theme_create() (*método de tkinter.ttk.Style*), 1492
theme_names() (*método de tkinter.ttk.Style*), 1492
theme_settings() (*método de tkinter.ttk.Style*), 1492
theme_use() (*método de tkinter.ttk.Style*), 1492
THOUSEP (*en el módulo locale*), 1402
Thread (*clase en threading*), 824
thread() (*método de imaplib.IMAP4*), 1322
thread_info (*en el módulo sys*), 1767
thread_time() (*en el módulo time*), 676
thread_time_ns() (*en el módulo time*), 676
ThreadedChildWatcher (*clase en asyncio*), 994
threading
    módulo, 821
threading_cleanup() (*en el módulo test.support.threading_helper*), 1679
threading_setup() (*en el módulo test.support.threading_helper*), 1679
ThreadingHTTPServer (*clase en http.server*), 1341
ThreadingMixIn (*clase en socketserver*), 1334
ThreadingTCPServer (*clase en socketserver*), 1334
ThreadingUDPServer (*clase en socketserver*), 1334
ThreadPool (*clase en multiprocessing.pool*), 868
ThreadPoolExecutor (*clase en concurrent.futures*), 884
threads
    POSIX, 916
threadsafety (*en el módulo sqlite3*), 495
throw (*2to3 fixer*), 1665
THURSDAY (*en el módulo calendar*), 238
ticket_lifetime_hint (*atributo de ssl.SSLSession*), 1065
tigetflag() (*en el módulo curses*), 762
tigetnum() (*en el módulo curses*), 762
tigetstr() (*en el módulo curses*), 762
TILDE (*en el módulo token*), 1925
tilt() (*en el módulo turtle*), 1426

tiltangle() (*en el módulo turtle*), 1426
time
    módulo, 670
time (*atributo de ssl.SSLSession*), 1065
time (*clase en datetime*), 213
time() (*en el módulo time*), 676
time() (*método de asyncio.loop*), 957
time() (*método de datetime.datetime*), 207
Time2Internaldate() (*en el módulo imaplib*), 1317
time_ns() (*en el módulo time*), 676
timedelta (*clase en datetime*), 195
TimedRotatingFileHandler (*clase en logging.handlers*), 746
timegm() (*en el módulo calendar*), 238
timeit
    módulo, 1711
timeit() (*en el módulo timeit*), 1712
timeit() (*método de timeit.Timer*), 1713
timeitopción de línea de comando
    -h, 1714
    --help, 1714
    -n N, 1714
    --number=N, 1714
    -p, 1714
    --process, 1714
    -r N, 1714
    --repeat=N, 1714
    -s S, 1714
    --setup=S, 1714
    -u, 1714
    --unit=U, 1714
    -v, 1714
    --verbose, 1714
timeout, 1012
timeout (*atributo de socketserver.BaseServer*), 1336
timeout (*atributo de ssl.SSLSession*), 1065
timeout (*atributo de subprocess.TimeoutExpired*), 891
timeout() (*método de curses.window*), 769
TIMEOUT_MAX (*en el módulo _thread*), 917
TIMEOUT_MAX (*en el módulo threading*), 823
TimeoutError, 109, 843, 888, 952
TimeoutExpired, 891
Timer (*clase en threading*), 832
Timer (*clase en timeit*), 1712
TimerHandle (*clase en asyncio*), 970
times() (*en el módulo os*), 649
TIMESTAMP (*atributo de py_compile.PycInvalidationMode*), 1935
timestamp() (*método de datetime.datetime*), 209
timetuple() (*método de datetime.date*), 200
timetuple() (*método de datetime.datetime*), 208
timetz() (*método de datetime.datetime*), 207
timezone (*clase en datetime*), 223
timezone (*en el módulo time*), 679
--timing
    traceopción de línea de comando, 1717
tipado de pato, **2083**
tipo, **2091**
tipos genéricos, **2084**
title() (*en el módulo turtle*), 1437
title() (*método de bytearray*), 71
title() (*método de bytes*), 71
title() (*método de str*), 56
Tix, 1493
tix_addbitmapdir() (*método de tkinter.tix.tixCommand*), 1497
tix_cget() (*método de tkinter.tix.tixCommand*), 1497
tix_configure() (*método de tkinter.tix.tixCommand*), 1496
tix_filedialog() (*método de tkinter.tix.tixCommand*), 1497
tix_getbitmap() (*método de tkinter.tix.tixCommand*), 1497
tix_getimage() (*método de tkinter.tix.tixCommand*), 1497
tix_option_get() (*método de tkinter.tix.tixCommand*), 1497
tix_resetoptions() (*método de tkinter.tix.tixCommand*), 1497
tixCommand (*clase en tkinter.tix*), 1496
Tk, 1455
tk (*atributo de tkinter.Tk*), 1457
Tk (*clase en tkinter*), 1457
Tk (*clase en tkinter.tix*), 1493
Tk Option Data Types, 1465
tkinter
    módulo, 1455
Tkinter, 1455
tkinter.colorchooser
    módulo, 1468
tkinter.commondialog
    módulo, 1472
tkinter.dnd
    módulo, 1473
tkinter.filedialog
    módulo, 1470
tkinter.font
    módulo, 1468
tkinter.messagebox
    módulo, 1472
tkinter.scrolledtext
    módulo, 1473
tkinter.simpledialog
    módulo, 1469
tkinter.tix
    módulo, 1493
tkinter.ttk
    módulo, 1474
TList (*clase en tkinter.tix*), 1495
TLS, 1033
TLSv1 (*atributo de ssl.TLSVersion*), 1046
TLSv1_1 (*atributo de ssl.TLSVersion*), 1046
TLSv1_2 (*atributo de ssl.TLSVersion*), 1046
TLSv1_3 (*atributo de ssl.TLSVersion*), 1046

TLSVersion (*clase en ssl*), 1046
TMP, 449
TMPDIR, 449
to_bytes() (*método de int*), 37
to_eng_string() (*método de decimal.Context*), 348
to_eng_string() (*método de decimal.Decimal*), 342
to_integral() (*método de decimal.Decimal*), 342
to_integral_exact() (*método de decimal.Context*), 348
to_integral_exact() (*método de decimal.Decimal*), 342
to_integral_value() (*método de decimal.Decimal*), 342
to_sci_string() (*método de decimal.Context*), 348
to_thread() (*en el módulo asyncio*), 929
ToASCII() (*en el módulo encodings.idna*), 190
tobuf() (*método de tarfile.TarInfo*), 546
tobytes() (*método de array.array*), 272
tobytes() (*método de memoryview*), 76
today() (*método de clase de datetime.date*), 199
today() (*método de clase de datetime.datetime*), 203
tofile() (*método de array.array*), 272
tok_name (*en el módulo token*), 1924
token
    módulo, 1924
token (*atributo de shlex.shlex*), 1451
Token (*clase en contextvars*), 913
token_bytes() (*en el módulo secrets*), 598
token_hex() (*en el módulo secrets*), 598
token_urlsafe() (*en el módulo secrets*), 598
TokenError, 1929
tokenize
    módulo, 1928
tokenize() (*en el módulo tokenize*), 1928
tokenizeopción de línea de comando
    -e, 1929
    --exact, 1929
    -h, 1929
    --help, 1929
tolist() (*método de array.array*), 272
tolist() (*método de memoryview*), 77
tomono() (*en el módulo audioop*), 1996
toordinal() (*método de datetime.date*), 201
toordinal() (*método de datetime.datetime*), 209
top() (*método de curses.panel.Panel*), 779
top() (*método de poplib.POP3*), 1315
top_panel() (*en el módulo curses.panel*), 778
--top-level-directory directory
    unittest-discoveropción de línea de comando, 1575
TopologicalSorter (*clase en graphlib*), 313
toprettyxml() (*método de xml.dom.minidom.Node*), 1230
toreadonly() (*método de memoryview*), 77
tostereo() (*en el módulo audioop*), 1996
tostring() (*en el módulo xml.etree.ElementTree*), 1208
tostringlist() (*en el módulo xml.etree.ElementTree*), 1208
total() (*método de collections.Counter*), 243
total_changes (*atributo de sqlite3.Connection*), 501
total_nframe (*atributo de tracemalloc.Traceback*), 1729
total_ordering() (*en el módulo functools*), 399
total_seconds() (*método de datetime.timedelta*), 198
touch() (*método de pathlib.Path*), 429
touchline() (*método de curses.window*), 769
touchwin() (*método de curses.window*), 769
ToUnicode() (*en el módulo encodings.idna*), 190
tounicode() (*método de array.array*), 272
towards() (*en el módulo turtle*), 1418
toxml() (*método de xml.dom.minidom.Node*), 1230
tparm() (*en el módulo curses*), 762
trace
    módulo, 1716
--trace
    traceopción de línea de comando, 1717
Trace (*clase en trace*), 1718
Trace (*clase en tracemalloc*), 1728
trace function, 823, 1757, 1764
trace() (*en el módulo inspect*), 1841
trace_dispatch() (*método de bdb.Bdb*), 1691
traceback
    módulo, 1816
    objeto, 1753, 1816
traceback (*atributo de tracemalloc.Statistic*), 1727
traceback (*atributo de tracemalloc.StatisticDiff*), 1728
traceback (*atributo de tracemalloc.Trace*), 1728
Traceback (*clase en tracemalloc*), 1728
traceback_limit (*atributo de tracemalloc.Snapshot*), 1727
traceback_limit (*atributo de wsgiref.handlers.BaseHandler*), 1267
TracebackException (*clase en traceback*), 1818
tracebacklimit (*en el módulo sys*), 1767
tracebacks
    in CGI scripts, 2004
TracebackType (*clase en types*), 283
tracemalloc
    módulo, 1719
traceopción de línea de comando
    -c, 1717
    -C, 1717
    --count, 1717
    --coverdir=<dir>, 1717
    -f, 1717
    --file=<file>, 1717
    -g, 1717
    --help, 1716
    --ignore-dir=<dir>, 1718
    --ignore-module=<mod>, 1718
    -l, 1717

--listfuncs, 1717
-m, 1717
--missing, 1717
--no-report, 1717
-r, 1717
-R, 1717
--report, 1717
-s, 1717
--summary, 1717
-t, 1717
-T, 1717
--timing, 1717
--trace, 1717
--trackcalls, 1717
--version, 1717
tracer() (*en el módulo turtle*), 1432
traces (*atributo de tracemalloc.Snapshot*), 1727
--trackcalls
traceopción de línea de comando, 1717
transfercmd() (*método de ftplib.FTP*), 1311
transient_internet() (*en el módulo test.support.socket_helper*), 1677
translate() (*en el módulo fnmatch*), 453
translate() (*método de bytearray*), 65
translate() (*método de bytes*), 65
translate() (*método de str*), 56
translation() (*en el módulo gettext*), 1393
transport (*atributo de asyncio.StreamWriter*), 937
Transport (*clase en asyncio*), 980
Transport Layer Security, 1033
Traversable (*clase en importlib.abc*), 1871
TraversableResources (*clase en importlib.abc*), 1872
Tree (*clase en tkinter.tix*), 1495
TreeBuilder (*clase en xml.etree.ElementTree*), 1214
Treeview (*clase en tkinter.ttk*), 1486
triangular() (*en el módulo random*), 364
true, 33
True, 33, 97
True (*variable incorporada*), 31
truediv() (*en el módulo operator*), 407
trunc() (*en el módulo math*), 323
trunc() (*in module math*), 36
truncate() (*en el módulo os*), 635
truncate() (*método de io.IOBase*), 661
truth
value, 33
truth() (*en el módulo operator*), 406
try
sentencia, 101
Try (*clase en ast*), 1906
ttk, 1474
tty
I/O control, 1974
módulo, 1975
ttyname() (*en el módulo os*), 617
TUESDAY (*en el módulo calendar*), 238
tupla nombrada, **2087**
tuple
objeto, 44, 46
Tuple (*clase en ast*), 1894
tuple (*clase incorporada*), 46
Tuple (*en el módulo typing*), 1520
tuple_params (*2to3 fixer*), 1665
turtle
módulo, 1409
Turtle (*clase en turtle*), 1437
turtledemo
módulo, 1441
turtles() (*en el módulo turtle*), 1436
TurtleScreen (*clase en turtle*), 1437
turtlesize() (*en el módulo turtle*), 1425
type
Boolean, 7
función incorporada, 96
objeto, 26
operations on dictionary, 83
operations on list, 44
union, 93
type (*atributo de optparse.Option*), 2042
type (*atributo de socket.socket*), 1029
type (*atributo de tarfile.TarInfo*), 546
type (*atributo de urllib.request.Request*), 1275
Type (*clase en typing*), 1522
type (*clase incorporada*), 26
type_check_only() (*en el módulo typing*), 1542
TYPE_CHECKER (*atributo de optparse.Option*), 2053
TYPE_CHECKING (*en el módulo typing*), 1543
type_comment (*atributo de ast.arg*), 1914
type_comment (*atributo de ast.Assign*), 1901
type_comment (*atributo de ast.For*), 1905
type_comment (*atributo de ast.FunctionDef*), 1914
type_comment (*atributo de ast.With*), 1907
TYPE_COMMENT (*en el módulo token*), 1926
TYPE_IGNORE (*en el módulo token*), 1926
typeahead() (*en el módulo curses*), 762
TypeAlias (*en el módulo typing*), 1520
typecode (*atributo de array.array*), 271
typecodes (*en el módulo array*), 270
TYPED_ACTIONS (*atributo de optparse.Option*), 2054
typed_subpart_iterator() (*en el módulo email.iterators*), 1151
TypedDict (*clase en typing*), 1532
TypeError, 107
TypeGuard (*en el módulo typing*), 1525
types
built-in, 33
immutable sequence, 44
módulo, 96, 280
mutable sequence, 44
operations on integer, 36
operations on mapping, 83
operations on numeric, 35
operations on sequence, 42, 44
types (*2to3 fixer*), 1665

TYPES (*atributo de optparse.Option*), 2053
types_map (*atributo de mimetypes.MimeTypes*), 1181
types_map (*en el módulo mimetypes*), 1180
types_map_inv (*atributo de mimetypes.MimeTypes*), 1181
TypeVar (*clase en typing*), 1526
typing
    módulo, 1511
TZ, 676, 677
tzinfo (*atributo de datetime.datetime*), 206
tzinfo (*atributo de datetime.time*), 214
tzinfo (*clase en datetime*), 217
tzname (*en el módulo time*), 679
tzname() (*método de datetime.datetime*), 208
tzname() (*método de datetime.time*), 216
tzname() (*método de datetime.timezone*), 223
tzname() (*método de datetime.tzinfo*), 218
TZPATH (*en el módulo zoneinfo*), 234
tzset() (*en el módulo time*), 676

U

-u
    timeitopción de línea de comando, 1714
UAdd (*clase en ast*), 1896
ucd_3_2_0 (*en el módulo unicodedata*), 160
udata (*atributo de select.kevent*), 1074
UDPServer (*clase en socketserver*), 1333
UF_APPEND (*en el módulo stat*), 443
UF_COMPRESSED (*en el módulo stat*), 443
UF_HIDDEN (*en el módulo stat*), 443
UF_IMMUTABLE (*en el módulo stat*), 443
UF_NODUMP (*en el módulo stat*), 443
UF_NOUNLINK (*en el módulo stat*), 443
UF_OPAQUE (*en el módulo stat*), 443
uid (*atributo de tarfile.TarInfo*), 546
UID (*clase en plistlib*), 582
uid() (*método de imaplib.IMAP4*), 1322
uidl() (*método de poplib.POP3*), 1315
u-LAW, 1987, 1994, 2064
ulaw2lin() (*en el módulo audioop*), 1996
ulp() (*en el módulo math*), 323
umask() (*en el módulo os*), 608
unalias (*pdb command*), 1702
uname (*atributo de tarfile.TarInfo*), 547
uname() (*en el módulo os*), 608
uname() (*en el módulo platform*), 780
UNARY_INVERT (*opcode*), 1944
UNARY_NEGATIVE (*opcode*), 1944
UNARY_NOT (*opcode*), 1944
UNARY_POSITIVE (*opcode*), 1944
UnaryOp (*clase en ast*), 1896
UnboundLocalError, 107
unbuffered I/O, 20
UNC paths
    and os.makedirs(), 624
UNCHECKED_HASH (*atributo de py_compile.PycInvalidationMode*), 1935
unconsumed_tail (*atributo de zlib.Decompress*), 516
unctrl() (*en el módulo curses*), 762
unctrl() (*en el módulo curses.ascii*), 777
Underflow (*clase en decimal*), 350
undisplay (*pdb command*), 1702
undo() (*en el módulo turtle*), 1417
undobufferentries() (*en el módulo turtle*), 1429
undoc_header (*atributo de cmd.Cmd*), 1445
unescape() (*en el módulo html*), 1191
unescape() (*en el módulo xml.sax.saxutils*), 1242
UnexpectedException, 1570
unexpectedSuccesses (*atributo de unittest.TestResult*), 1595
unfreeze() (*en el módulo gc*), 1826
unget_wch() (*en el módulo curses*), 763
ungetch() (*en el módulo curses*), 763
ungetch() (*en el módulo msvcrt*), 1958
ungetmouse() (*en el módulo curses*), 763
ungetwch() (*en el módulo msvcrt*), 1958
unhexlify() (*en el módulo binascii*), 1188
Unicode, 158, 174
    database, 158
unicode (*2to3 fixer*), 1665
unicodedata
    módulo, 158
UnicodeDecodeError, 107
UnicodeEncodeError, 107
UnicodeError, 107
UnicodeTranslateError, 107
UnicodeWarning, 109
unidata_version (*en el módulo unicodedata*), 160
unified_diff() (*en el módulo difflib*), 147
uniform() (*en el módulo random*), 364
UnimplementedFileMode, 1303
union
    type, 93
Union
    objeto, 93
Union (*clase en ctypes*), 818
Union (*en el módulo typing*), 1520
union() (*método de frozenset*), 82
UnionType (*clase en types*), 283
unique() (*en el módulo enum*), 297
--unit=U
    timeitopción de línea de comando, 1714
unittest
    módulo, 1571
unittest-discoveropción de línea de comando
    -p, 1575
    --pattern pattern, 1575
    -s, 1575
    --start-directory directory, 1575
    -t, 1575
    --top-level-directory directory, 1575

-v, 1574
--verbose, 1574
unittest.mock
módulo, 1601
unittestopción de línea de comando
-b, 1574
--buffer, 1574
-c, 1574
--catch, 1574
-f, 1574
--failfast, 1574
-k, 1574
--locals, 1574
universal newlines
bytearray.splitlines method, 70
bytes.splitlines method, 70
csv.reader function, 556
importlib.abc.InspectLoader.get_source method, 1869
io.IncrementalNewlineDecoder class, 669
io.TextIOWrapper class, 667
open() built-in function, 20
str.splitlines method, 55
subprocess module, 892
UNIX
file control, 1977
I/O control, 1977
unix_dialect (*clase en csv*), 558
unix_shell (*en el módulo test.support*), 1668
UnixDatagramServer (*clase en socketserver*), 1333
UnixStreamServer (*clase en socketserver*), 1333
unknown (*atributo de uuid.SafeUUID*), 1330
unknown_decl() (*método de html.parser.HTMLParser*), 1194
unknown_open() (*método de urllib.request.BaseHandler*), 1278
unknown_open() (*método de urllib.request.UnknownHandler*), 1282
UnknownHandler (*clase en urllib.request*), 1275
UnknownProtocol, 1303
UnknownTransferEncoding, 1303
unlink() (*en el módulo os*), 635
unlink() (*en el módulo test.support.os_helper*), 1681
unlink() (*método de multiprocessing.shared_memory.SharedMemory*), 878
unlink() (*método de pathlib.Path*), 430
unlink() (*método de xml.dom.minidom.Node*), 1229
unload() (*en el módulo test.support.import_helper*), 1682
unlock() (*método de mailbox.Babyl*), 1169
unlock() (*método de mailbox.Mailbox*), 1164
unlock() (*método de mailbox.Maildir*), 1166
unlock() (*método de mailbox.mbox*), 1166
unlock() (*método de mailbox.MH*), 1168
unlock() (*método de mailbox.MMDF*), 1169
unpack() (*en el módulo struct*), 168
unpack() (*método de struct.Struct*), 173
unpack_archive() (*en el módulo shutil*), 461
unpack_array() (*método de xdrlib.Unpacker*), 2074
unpack_bytes() (*método de xdrlib.Unpacker*), 2074
unpack_double() (*método de xdrlib.Unpacker*), 2074
UNPACK_EX (*opcode*), 1948
unpack_farray() (*método de xdrlib.Unpacker*), 2074
unpack_float() (*método de xdrlib.Unpacker*), 2074
unpack_fopaque() (*método de xdrlib.Unpacker*), 2074
unpack_from() (*en el módulo struct*), 168
unpack_from() (*método de struct.Struct*), 173
unpack_fstring() (*método de xdrlib.Unpacker*), 2074
unpack_list() (*método de xdrlib.Unpacker*), 2074
unpack_opaque() (*método de xdrlib.Unpacker*), 2074
UNPACK_SEQUENCE (*opcode*), 1948
unpack_string() (*método de xdrlib.Unpacker*), 2074
Unpacker (*clase en xdrlib*), 2072
unparse() (*en el módulo ast*), 1918
unparsedEntityDecl() (*método de xml.sax.handler.DTDHandler*), 1241
UnparsedEntityDeclHandler() (*método de xml.parsers.expat.xmlparser*), 1251
Unpickler (*clase en pickle*), 470
UnpicklingError, 468
unquote() (*en el módulo email.utils*), 1148
unquote() (*en el módulo urllib.parse*), 1296
unquote_plus() (*en el módulo urllib.parse*), 1296
unquote_to_bytes() (*en el módulo urllib.parse*), 1296
unraisablehook() (*en el módulo sys*), 1767
unregister() (*en el módulo atexit*), 1815
unregister() (*en el módulo codecs*), 175
unregister() (*en el módulo faulthandler*), 1696
unregister() (*método de select.devpoll*), 1070
unregister() (*método de select.epoll*), 1071
unregister() (*método de selectors.BaseSelector*), 1076
unregister() (*método de select.poll*), 1072
unregister_archive_format() (*en el módulo shutil*), 461
unregister_dialect() (*en el módulo csv*), 556
unregister_unpack_format() (*en el módulo shutil*), 462
unsafe (*atributo de uuid.SafeUUID*), 1330
unselect() (*método de imaplib.IMAP4*), 1322
unset() (*método de test.support.os_helper.EnvironmentVarGuard*), 1681
unsetenv() (*en el módulo os*), 609
UnstructuredHeader (*clase en email.headerregistry*), 1118
unsubscribe() (*método de imaplib.IMAP4*), 1322
UnsupportedOperation, 659

until (*pdb command*), 1701
untokenize() (*en el módulo tokenize*), 1928
untouchwin() (*método de curses.window*), 769
unused_data (*atributo de bz2.BZ2Decompressor*), 522
unused_data (*atributo de lzma.LZMADecompressor*), 527
unused_data (*atributo de zlib.Decompress*), 516
unverifiable (*atributo de urllib.request.Request*), 1275
unwrap() (*en el módulo inspect*), 1839
unwrap() (*en el módulo urllib.parse*), 1293
unwrap() (*método de ssl.SSLSocket*), 1049
up (*pdb command*), 1699
up() (*en el módulo turtle*), 1420
update() (*en el módulo turtle*), 1432
update() (*método de collections.Counter*), 243
update() (*método de dict*), 86
update() (*método de frozenset*), 83
update() (*método de hashlib.hash*), 587
update() (*método de hmac.HMAC*), 596
update() (*método de http.cookies.Morsel*), 1349
update() (*método de mailbox.Mailbox*), 1164
update() (*método de mailbox.Maildir*), 1165
update() (*método de trace.CoverageResults*), 1718
update_abstractmethods() (*en el módulo abc*), 1814
update_authenticated() (*método de urllib.request.HTTPPasswordMgrWithPriorAuth*), 1280
update_lines_cols() (*en el módulo curses*), 763
update_panels() (*en el módulo curses.panel*), 778
update_visible() (*método de mailbox.BabylMessage*), 1175
update_wrapper() (*en el módulo functools*), 404
upgrade_dependencies() (*método de venv.EnvBuilder*), 1738
upper() (*método de bytearray*), 71
upper() (*método de bytes*), 71
upper() (*método de str*), 57
urandom() (*en el módulo os*), 656
URL, 1288, 1298, 1341, 1997
    parsing, 1288
    relative, 1288
url (*atributo de http.client.HTTPResponse*), 1306
url (*atributo de urllib.response.addinfourl*), 1288
url (*atributo de xmlrpc.client.ProtocolError*), 1364
url2pathname() (*en el módulo urllib.request*), 1272
urlcleanup() (*en el módulo urllib.request*), 1286
urldefrag() (*en el módulo urllib.parse*), 1293
urlencode() (*en el módulo urllib.parse*), 1296
URLError, 1297
urljoin() (*en el módulo urllib.parse*), 1292
urllib
    módulo, 1270
urllib (*2to3 fixer*), 1665
urllib.error
    módulo, 1297
urllib.parse
    módulo, 1288
urllib.request
    módulo, 1270, 1301
urllib.response
    módulo, 1288
urllib.robotparser
    módulo, 1298
urlopen() (*en el módulo urllib.request*), 1270
URLopener (*clase en urllib.request*), 1286
urlparse() (*en el módulo urllib.parse*), 1289
urlretrieve() (*en el módulo urllib.request*), 1285
urlsafe_b64decode() (*en el módulo base64*), 1183
urlsafe_b64encode() (*en el módulo base64*), 1182
urlsplit() (*en el módulo urllib.parse*), 1291
urlunparse() (*en el módulo urllib.parse*), 1291
urlunsplit() (*en el módulo urllib.parse*), 1292
urn (*atributo de uuid.UUID*), 1331
use_default_colors() (*en el módulo curses*), 763
use_env() (*en el módulo curses*), 763
use_rawinput (*atributo de cmd.Cmd*), 1445
UseForeignDTD() (*método de xml.parsers.expat.xmlparser*), 1249
user
    effective id, 605
    id, 606
    id, setting, 608
USER, 756
user() (*método de poplib.POP3*), 1315
USER_BASE (*en el módulo site*), 1846
user_call() (*método de bdb.Bdb*), 1692
user_exception() (*método de bdb.Bdb*), 1692
user_line() (*método de bdb.Bdb*), 1692
user_return() (*método de bdb.Bdb*), 1692
USER_SITE (*en el módulo site*), 1846
--user-base
    siteopción de línea de comando, 1847
usercustomize
    módulo, 1845
UserDict (*clase en collections*), 255
UserList (*clase en collections*), 256
USERNAME, 433, 605, 756
username (*atributo de email.headerregistry.Address*), 1122
USERPROFILE, 433
userptr() (*método de curses.panel.Panel*), 779
--user-site
    siteopción de línea de comando, 1847
UserString (*clase en collections*), 256
UserWarning, 109
USTAR_FORMAT (*en el módulo tarfile*), 541
USub (*clase en ast*), 1896
UTC, 670
utc (*atributo de datetime.timezone*), 224

utcfromtimestamp() (*método de clase de datetime.datetime*), 204
utcnow() (*método de clase de datetime.datetime*), 204
utcoffset() (*método de datetime.datetime*), 208
utcoffset() (*método de datetime.time*), 216
utcoffset() (*método de datetime.timezone*), 223
utcoffset() (*método de datetime.tzinfo*), 217
utctimetuple() (*método de datetime.datetime*), 209
utf8 (*atributo de email.policy.EmailPolicy*), 1113
utf8() (*método de poplib.POP3*), 1315
utf8_enabled (*atributo de imaplib.IMAP4*), 1322
utime() (*en el módulo os*), 635
uu
    módulo, 1186, 2071
uuid
    módulo, 1329
UUID (*clase en uuid*), 1330
uuid1, 1331
uuid1() (*en el módulo uuid*), 1331
uuid3, 1331
uuid3() (*en el módulo uuid*), 1331
uuid4, 1331
uuid4() (*en el módulo uuid*), 1331
uuid5, 1332
uuid5() (*en el módulo uuid*), 1331
UuidCreate() (*en el módulo msilib*), 2015

V

-v
    tarfileopción de línea de comando, 551
    timeitopción de línea de comando, 1714
    unittest-discoveropción de línea de comando, 1574
v4_int_to_packed() (*en el módulo ipaddress*), 1384
v6_int_to_packed() (*en el módulo ipaddress*), 1384
valid_signals() (*en el módulo signal*), 1081
validator() (*en el módulo wsgiref.validate*), 1264
value
    truth, 33
value (*atributo de ctypes._SimpleCData*), 816
value (*atributo de http.cookiejar.Cookie*), 1357
value (*atributo de http.cookies.Morsel*), 1348
value (*atributo de xml.dom.Attr*), 1225
Value() (*en el módulo multiprocessing*), 852
Value() (*en el módulo multiprocessing.sharedctypes*), 854
Value() (*método de multiprocessing.managers.SyncManager*), 857
value_decode() (*método de http.cookies.BaseCookie*), 1347
value_encode() (*método de http.cookies.BaseCookie*), 1347
ValueError, 107
valuerefs() (*método de weakref.WeakValueDictionary*), 275
values
    Boolean, 97
values() (*método de contextvars.Context*), 915
values() (*método de dict*), 86
values() (*método de email.message.EmailMessage*), 1097
values() (*método de email.message.Message*), 1135
values() (*método de mailbox.Mailbox*), 1163
values() (*método de types.MappingProxyType*), 284
ValuesView (*clase en collections.abc*), 260
ValuesView (*clase en typing*), 1537
var (*atributo de contextvars.Token*), 913
variable de clase, **2081**
variable de contexto, **2082**
variables de entorno
    AUDIODEV, 2056
    BROWSER, 1257, 1258
    COLS, 763
    COLUMNS, 763
    COMSPEC, 649, 894
    DISPLAY, 1457
    HOME, 433, 1457
    HOMEDRIVE, 433
    HOMEPATH, 433
    http_proxy, 1271, 1284
    IDLESTARTUP, 1505
    KDEDIR, 1259
    LANG, 1391, 1393, 1401, 1403
    LANGUAGE, 1391, 1393
    LC_ALL, 1391, 1393
    LC_MESSAGES, 1391, 1393
    LINES, 758, 763
    LNAME, 756
    LOGNAME, 605, 756
    MIXERDEV, 2056
    no_proxy, 1273
    PAGER, 1544
    PATH, 641, 642, 646, 647, 655, 893, 1257, 1844, 2002, 2004
    POSIXLY_CORRECT, 712
    PYTHON_DOM, 1219
    PYTHONASYNCIODEBUG, 969, 1005, 1546
    PYTHONBREAKPOINT, 1752
    PYTHONCASEOK, 29
    PYTHONCOERCECLOCALE, 603
    PYTHONDEVMODE, 1545
    PYTHONDOCS, 1545
    PYTHONDONTWRITEBYTECODE, 1753
    PYTHONFAULTHANDLER, 1546, 1694
    PYTHONHOME, 1678
    PYTHONINTMAXSTRDIGITS, 99, 1760
    PYTHONIOENCODING, 602, 1766
    PYTHONLEGACYWINDOWSFSENCODING, 1766
    PYTHONLEGACYWINDOWSSTDIO, 1766
    PYTHONMALLOC, 1546
    PYTHONNOUSERSITE, 1846

PYTHONPATH, 1678, 1761, 2002
PYTHONPYCACHEPREFIX, 1753
PYTHONSTARTUP, 165, 1505, 1760, 1845
PYTHONTRACEMALLOC, 1719, 1724
PYTHONTRACEMALLOC`comienzo, configura la variable del entorno a ``25`, 1719
PYTHONTZPATH, 230, 234
PYTHONUNBUFFERED, 1766
PYTHONUSERBASE, 1846
PYTHONUSERSITE, 1678
PYTHONUTF8, 603, 1766
PYTHONWARNDEFAULTENCODING, 658
PYTHONWARNINGS, 1545, 1780, 1781
SOURCE_DATE_EPOCH, 1935, 1937
SSLKEYLOGFILE, 1035
SystemRoot, 896
TEMP, 449
TERM, 762
TMP, 449
TMPDIR, 449
TZ, 676, 677
USER, 756
USERNAME, 433, 605, 756
USERPROFILE, 433
variance (*atributo de statistics.NormalDist*), 377
variance() (*en el módulo statistics*), 374
variant (*atributo de uuid.UUID*), 1331
vars()
función incorporada, 26
vbar (*atributo de tkinter.scrolledtext.ScrolledText*), 1473
VBAR (*en el módulo token*), 1925
VBAREQUAL (*en el módulo token*), 1926
Vec2D (*clase en turtle*), 1438
venv
módulo, 1733
--verbose
tarfileopción de línea de comando, 551
timeitopción de línea de comando, 1714
unittest-discoveropción de línea de comando, 1574
VERBOSE (*en el módulo re*), 131
verbose (*en el módulo tabnanny*), 1932
verbose (*en el módulo test.support*), 1668
verify() (*método de smtplib.SMTP*), 1326
VERIFY_ALLOW_PROXY_CERTS (*en el módulo ssl*), 1041
verify_client_post_handshake() (*método de ssl.SSLSocket*), 1049
verify_code (*atributo de ssl.SSLCertVerificationError*), 1036
VERIFY_CRL_CHECK_CHAIN (*en el módulo ssl*), 1041
VERIFY_CRL_CHECK_LEAF (*en el módulo ssl*), 1040
VERIFY_DEFAULT (*en el módulo ssl*), 1040
verify_flags (*atributo de ssl.SSLContext*), 1057
verify_message (*atributo de ssl.SSLCertVerificationError*), 1036
verify_mode (*atributo de ssl.SSLContext*), 1057
verify_request() (*método de socketserver.BaseServer*), 1337
VERIFY_X509_PARTIAL_CHAIN (*en el módulo ssl*), 1041
VERIFY_X509_STRICT (*en el módulo ssl*), 1041
VERIFY_X509_TRUSTED_FIRST (*en el módulo ssl*), 1041
VerifyFlags (*clase en ssl*), 1041
VerifyMode (*clase en ssl*), 1040
--version
traceopción de línea de comando, 1717
version (*atributo de email.headerregistry.MIMEVersionHeader*), 1120
version (*atributo de http.client.HTTPResponse*), 1306
version (*atributo de http.cookiejar.Cookie*), 1357
version (*atributo de ipaddress.IPv4Address*), 1373
version (*atributo de ipaddress.IPv4Network*), 1378
version (*atributo de ipaddress.IPv6Address*), 1375
version (*atributo de ipaddress.IPv6Network*), 1381
version (*atributo de urllib.request.URLopener*), 1286
version (*atributo de uuid.UUID*), 1331
version (*en el módulo curses*), 770
version (*en el módulo marshal*), 486
version (*en el módulo sqlite3*), 496
version (*en el módulo sys*), 1768
version() (*en el módulo ensurepip*), 1733
version() (*en el módulo platform*), 780
version() (*método de ssl.SSLSocket*), 1050
version_info (*en el módulo sqlite3*), 496
version_info (*en el módulo sys*), 1768
version_string() (*método de http.server.BaseHTTPRequestHandler*), 1344
vformat() (*método de string.Formatter*), 114
virtual
Environments, 1733
visit() (*método de ast.NodeVisitor*), 1919
vista de diccionario, **2082**
vline() (*método de curses.window*), 769
voidcmd() (*método de ftplib.FTP*), 1311
volume (*atributo de zipfile.ZipInfo*), 537
vonmisesvariate() (*en el módulo random*), 364

W

W_OK (*en el módulo os*), 620
wait() (*en el módulo asyncio*), 928
wait() (*en el módulo concurrent.futures*), 888
wait() (*en el módulo multiprocessing.connection*), 865
wait() (*en el módulo os*), 649
wait() (*método de asyncio.Condition*), 944
wait() (*método de asyncio.Event*), 942
wait() (*método de asyncio.subprocess.Process*), 947
wait() (*método de multiprocessing.pool.AsyncResult*), 863

wait() (*método de subprocess.Popen*), 898
wait() (*método de threading.Barrier*), 833
wait() (*método de threading.Condition*), 829
wait() (*método de threading.Event*), 832
wait3() (*en el módulo os*), 650
wait4() (*en el módulo os*), 651
wait_closed() (*método de asyncio.Server*), 972
wait_closed() (*método de asyncio.StreamWriter*), 938
wait_for() (*en el módulo asyncio*), 927
wait_for() (*método de asyncio.Condition*), 944
wait_for() (*método de threading.Condition*), 829
wait_process() (*en el módulo test.support*), 1672
wait_threads_exit() (*en el módulo test.support.threading_helper*), 1679
waitid() (*en el módulo os*), 649
waitpid() (*en el módulo os*), 650
waitstatus_to_exitcode() (*en el módulo os*), 651
walk() (*en el módulo ast*), 1919
walk() (*en el módulo os*), 636
walk() (*método de email.message.EmailMessage*), 1099
walk() (*método de email.message.Message*), 1138
walk_packages() (*en el módulo pkgutil*), 1857
walk_stack() (*en el módulo traceback*), 1818
walk_tb() (*en el módulo traceback*), 1818
want (*atributo de doctest.Example*), 1564
warn() (*en el módulo warnings*), 1784
warn_explicit() (*en el módulo warnings*), 1784
Warning, 109, 504
warning() (*en el módulo logging*), 727
warning() (*método de logging.Logger*), 717
warning() (*método de xml.sax.handler.ErrorHandler*), 1241
warnings, 1779
    módulo, 1779
WarningsRecorder (*clase en test.support.warnings_helper*), 1684
warnoptions (*en el módulo sys*), 1768
wasSuccessful() (*método de unittest.TestResult*), 1595
WatchedFileHandler (*clase en logging.handlers*), 744
wave
    módulo, 1387
WCONTINUED (*en el módulo os*), 651
WCOREDUMP() (*en el módulo os*), 651
WeakKeyDictionary (*clase en weakref*), 274
WeakMethod (*clase en weakref*), 275
weakref
    módulo, 272
WeakSet (*clase en weakref*), 275
WeakValueDictionary (*clase en weakref*), 275
webbrowser
    módulo, 1257
WEDNESDAY (*en el módulo calendar*), 238
weekday() (*en el módulo calendar*), 237
weekday() (*método de datetime.date*), 201
weekday() (*método de datetime.datetime*), 209
weekheader() (*en el módulo calendar*), 238
weibullvariate() (*en el módulo random*), 364
WEXITED (*en el módulo os*), 650
WEXITSTATUS() (*en el módulo os*), 652
wfile (*atributo de http.server.BaseHTTPRequestHandler*), 1342
what() (*en el módulo imghdr*), 2008
what() (*en el módulo sndhdr*), 2064
whathdr() (*en el módulo sndhdr*), 2065
whatis (*pdb command*), 1701
when() (*método de asyncio.TimerHandle*), 971
where (*pdb command*), 1699
which() (*en el módulo shutil*), 458
whichdb() (*en el módulo dbm*), 486
while
    sentencia, 33
While (*clase en ast*), 1905
whitespace (*atributo de shlex.shlex*), 1451
whitespace (*en el módulo string*), 114
whitespace_split (*atributo de shlex.shlex*), 1451
Widget (*clase en tkinter.ttk*), 1478
width (*atributo de textwrap.TextWrapper*), 157
width() (*en el módulo turtle*), 1420
WIFCONTINUED() (*en el módulo os*), 652
WIFEXITED() (*en el módulo os*), 652
WIFSIGNALED() (*en el módulo os*), 652
WIFSTOPPED() (*en el módulo os*), 652
win32_edition() (*en el módulo platform*), 781
win32_is_iot() (*en el módulo platform*), 781
win32_ver() (*en el módulo platform*), 781
WinDLL (*clase en ctypes*), 807
window manager (*widgets*), 1464
window() (*método de curses.panel.Panel*), 779
window_height() (*en el módulo turtle*), 1436
window_width() (*en el módulo turtle*), 1436
Windows ini file, 562
WindowsError, 107
WindowsPath (*clase en pathlib*), 423
WindowsProactorEventLoopPolicy (*clase en asyncio*), 993
WindowsRegistryFinder (*clase en importlib.machinery*), 1875
WindowsSelectorEventLoopPolicy (*clase en asyncio*), 993
winerror (*atributo de OSError*), 104
WinError() (*en el módulo ctypes*), 814
WINFUNCTYPE() (*en el módulo ctypes*), 810
winreg
    módulo, 1959
WinSock, 1069
winsound
    módulo, 1968
winver (*en el módulo sys*), 1768
With (*clase en ast*), 1907
WITH_EXCEPT_START (*opcode*), 1947

with_hostmask (*atributo de ipaddress.IPv4Interface*), 1383
with_hostmask (*atributo de ipaddress.IPv4Network*), 1379
with_hostmask (*atributo de ipaddress.IPv6Interface*), 1384
with_hostmask (*atributo de ipaddress.IPv6Network*), 1381
with_name() (*método de pathlib.PurePath*), 422
with_netmask (*atributo de ipaddress.IPv4Interface*), 1383
with_netmask (*atributo de ipaddress.IPv4Network*), 1379
with_netmask (*atributo de ipaddress.IPv6Interface*), 1384
with_netmask (*atributo de ipaddress.IPv6Network*), 1381
with_prefixlen (*atributo de ipaddress.IPv4Interface*), 1383
with_prefixlen (*atributo de ipaddress.IPv4Network*), 1379
with_prefixlen (*atributo de ipaddress.IPv6Interface*), 1384
with_prefixlen (*atributo de ipaddress.IPv6Network*), 1381
with_pymalloc() (*en el módulo test.support*), 1670
with_stem() (*método de pathlib.PurePath*), 422
with_suffix() (*método de pathlib.PurePath*), 422
with_traceback() (*método de BaseException*), 102
withitem (*clase en ast*), 1907
WNOHANG (*en el módulo os*), 651
WNOWAIT (*en el módulo os*), 650
wordchars (*atributo de shlex.shlex*), 1450
World Wide Web, 1257, 1288, 1298
wrap() (*en el módulo textwrap*), 155
wrap() (*método de textwrap.TextWrapper*), 158
wrap_bio() (*método de ssl.SSLContext*), 1055
wrap_future() (*en el módulo asyncio*), 976
wrap_socket() (*en el módulo ssl*), 1039
wrap_socket() (*método de ssl.SSLContext*), 1055
wrapper() (*en el módulo curses*), 763
WrapperDescriptorType (*en el módulo types*), 282
wraps() (*en el módulo functools*), 405
WRITABLE (*en el módulo tkinter*), 1467
writable() (*método de asyncore.dispatcher*), 1992
writable() (*método de io.IOBase*), 662
write() (*en el módulo os*), 617
write() (*en el módulo turtle*), 1424
write() (*método de asyncio.StreamWriter*), 937
write() (*método de asyncio.WriteTransport*), 982
write() (*método de codecs.StreamWriter*), 181
write() (*método de code.InteractiveInterpreter*), 1850
write() (*método de configparser.ConfigParser*), 577
write() (*método de email.generator.BytesGenerator*), 1107
write() (*método de email.generator.Generator*), 1108
write() (*método de io.BufferedIOBase*), 663
write() (*método de io.BufferedWriter*), 666
write() (*método de io.RawIOBase*), 662
write() (*método de io.TextIOBase*), 667
write() (*método de mmap.mmap*), 1090
write() (*método de ossaudiodev.oss_audio_device*), 2056
write() (*método de ssl.MemoryBIO*), 1065
write() (*método de ssl.SSLSocket*), 1047
write() (*método de telnetlib.Telnet*), 2070
write() (*método de xml.etree.ElementTree.ElementTree*), 1213
write() (*método de zipfile.ZipFile*), 533
write_byte() (*método de mmap.mmap*), 1090
write_bytes() (*método de pathlib.Path*), 430
write_docstringdict() (*en el módulo turtle*), 1439
write_eof() (*método de asyncio.StreamWriter*), 937
write_eof() (*método de asyncio.WriteTransport*), 982
write_eof() (*método de ssl.MemoryBIO*), 1065
write_history_file() (*en el módulo readline*), 163
write_results() (*método de trace.CoverageResults*), 1718
write_text() (*método de pathlib.Path*), 430
write_through (*atributo de io.TextIOWrapper*), 668
writeall() (*método de ossaudiodev.oss_audio_device*), 2057
writeframes() (*método de aifc.aifc*), 1987
writeframes() (*método de sunau.AU_write*), 2068
writeframes() (*método de wave.Wave_write*), 1389
writeframesraw() (*método de aifc.aifc*), 1987
writeframesraw() (*método de sunau.AU_write*), 2068
writeframesraw() (*método de wave.Wave_write*), 1389
writeheader() (*método de csv.DictWriter*), 561
writelines() (*método de asyncio.StreamWriter*), 937
writelines() (*método de asyncio.WriteTransport*), 982
writelines() (*método de codecs.StreamWriter*), 181
writelines() (*método de io.IOBase*), 662
writepy() (*método de zipfile.PyZipFile*), 535
writer() (*en el módulo csv*), 556
writerow() (*método de csv.csvwriter*), 560
writerows() (*método de csv.csvwriter*), 560
writestr() (*método de zipfile.ZipFile*), 533
WriteTransport (*clase en asyncio*), 979
writev() (*en el módulo os*), 618
writexml() (*método de xml.dom.minidom.Node*), 1230
WrongDocumentErr, 1227
ws_comma (*2to3 fixer*), 1665
wsgi_file_wrapper (*atributo de wsgiref.handlers.BaseHandler*), 1268
wsgi_multiprocess (*atributo de wsgiref.handlers.BaseHandler*), 1266

wsgi_multithread (*atributo de wsgiref.handlers.BaseHandler*), 1266
wsgi_run_once (*atributo de wsgiref.handlers.BaseHandler*), 1267
wsgiref
    módulo, 1260
wsgiref.handlers
    módulo, 1265
wsgiref.headers
    módulo, 1262
wsgiref.simple_server
    módulo, 1263
wsgiref.util
    módulo, 1260
wsgiref.validate
    módulo, 1264
WSGIRequestHandler (*clase en wsgiref.simple_server*), 1264
WSGIServer (*clase en wsgiref.simple_server*), 1263
wShowWindow (*atributo de subprocess.STARTUPINFO*), 900
WSTOPPED (*en el módulo os*), 650
WSTOPSIG() (*en el módulo os*), 652
wstring_at() (*en el módulo ctypes*), 814
WTERMSIG() (*en el módulo os*), 652
WUNTRACED (*en el módulo os*), 651
WWW, 1257, 1288, 1298
    server, 1341, 1997

X

X (*en el módulo re*), 131
-x regex
    compileallopción de línea de comando, 1936
X509 certificate, 1058
X_OK (*en el módulo os*), 620
xatom() (*método de imaplib.IMAP4*), 1322
XATTR_CREATE (*en el módulo os*), 640
XATTR_REPLACE (*en el módulo os*), 640
XATTR_SIZE_MAX (*en el módulo os*), 640
xcor() (*en el módulo turtle*), 1419
XDR, 2072
xdrlib
    módulo, 2072
xhdr() (*método de nntplib.NNTP*), 2028
XHTML, 1192
XHTML_NAMESPACE (*en el módulo xml.dom*), 1219
xml
    módulo, 1197
XML() (*en el módulo xml.etree.ElementTree*), 1209
XML_ERROR_ABORTED (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_ASYNC_ENTITY (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_BAD_CHAR_REF (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_BINARY_ENTITY_REF (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_DUPLICATE_ATTRIBUTE (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_ENTITY_DECLARED_IN_PE (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_EXTERNAL_ENTITY_HANDLING (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_FEATURE_REQUIRES_XML_DTD (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_FINISHED (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_INCOMPLETE_PE (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_INCORRECT_ENCODING (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_INVALID_TOKEN (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_JUNK_AFTER_DOC_ELEMENT (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_MISPLACED_XML_PI (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_NO_ELEMENTS (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_NO_MEMORY (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_NOT_STANDALONE (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_NOT_SUSPENDED (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_PARAM_ENTITY_REF (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_PARTIAL_CHAR (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_PUBLICID (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_RECURSIVE_ENTITY_REF (*en el módulo xml.parsers.expat.errors*), 1254
XML_ERROR_SUSPEND_PE (*en el módulo xml.parsers.expat.errors*), 1256
XML_ERROR_SUSPENDED (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_SYNTAX (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_TAG_MISMATCH (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_TEXT_DECL (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_UNBOUND_PREFIX (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_UNCLOSED_CDATA_SECTION (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_UNCLOSED_TOKEN (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_UNDECLARING_PREFIX (*en el módulo xml.parsers.expat.errors*), 1255

XML_ERROR_UNDEFINED_ENTITY (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_UNEXPECTED_STATE (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_UNKNOWN_ENCODING (*en el módulo xml.parsers.expat.errors*), 1255
XML_ERROR_XML_DECL (*en el módulo xml.parsers.expat.errors*), 1255
XML_NAMESPACE (*en el módulo xml.dom*), 1219
xmlcharrefreplace
    error handler's name, 177
xmlcharrefreplace_errors() (*en el módulo codecs*), 178
XmlDeclHandler() (*método de xml.parsers.expat.xmlparser*), 1250
xml.dom
    módulo, 1218
xml.dom.minidom
    módulo, 1228
xml.dom.pulldom
    módulo, 1233
xml.etree.ElementInclude.default_loader()
    función incorporada, 1210
xml.etree.ElementInclude.include()
    función incorporada, 1210
xml.etree.ElementTree
    módulo, 1198
XMLFilterBase (*clase en xml.sax.saxutils*), 1243
XMLGenerator (*clase en xml.sax.saxutils*), 1242
XMLID() (*en el módulo xml.etree.ElementTree*), 1209
XMLNS_NAMESPACE (*en el módulo xml.dom*), 1219
XMLParser (*clase en xml.etree.ElementTree*), 1216
xml.parsers.expat
    módulo, 1247
xml.parsers.expat.errors
    módulo, 1254
xml.parsers.expat.model
    módulo, 1253
XMLParserType (*en el módulo xml.parsers.expat*), 1247
XMLPullParser (*clase en xml.etree.ElementTree*), 1217
XMLReader (*clase en xml.sax.xmlreader*), 1243
xmlrpc.client
    módulo, 1359
xmlrpc.server
    módulo, 1366
xml.sax
    módulo, 1235
xml.sax.handler
    módulo, 1236
xml.sax.saxutils
    módulo, 1242
xml.sax.xmlreader
    módulo, 1243
xor() (*en el módulo operator*), 407
xover() (*método de nntplib.NNTP*), 2028
xrange (*2to3 fixer*), 1665
xreadlines (*2to3 fixer*), 1665
xview() (*método de tkinter.ttk.Treeview*), 1489

Y

ycor() (*en el módulo turtle*), 1419
year (*atributo de datetime.date*), 200
year (*atributo de datetime.datetime*), 206
Year 2038, 670
yeardatescalendar() (*método de calendar.Calendar*), 235
yeardays2calendar() (*método de calendar.Calendar*), 235
yeardayscalendar() (*método de calendar.Calendar*), 235
YESEXPR (*en el módulo locale*), 1403
Yield (*clase en ast*), 1915
YIELD_FROM (*opcode*), 1947
YIELD_VALUE (*opcode*), 1947
YieldFrom (*clase en ast*), 1915
yiq_to_rgb() (*en el módulo colorsys*), 1390
yview() (*método de tkinter.ttk.Treeview*), 1490

Z

Zen de Python, **2092**
ZeroDivisionError, 107
zfill() (*método de bytearray*), 72
zfill() (*método de bytes*), 72
zfill() (*método de str*), 57
zip (*2to3 fixer*), 1665
zip()
    función incorporada, 27
ZIP_BZIP2 (*en el módulo zipfile*), 530
ZIP_DEFLATED (*en el módulo zipfile*), 530
zip_longest() (*en el módulo itertools*), 391
ZIP_LZMA (*en el módulo zipfile*), 530
ZIP_STORED (*en el módulo zipfile*), 530
zipapp
    módulo, 1742
zipappopción de línea de comando
    -c, 1743
    --compress, 1743
    -h, 1743
    --help, 1743
    --info, 1743
    -m <mainfn>, 1743
    --main=<mainfn>, 1743
    -o <output>, 1743
    --output=<output>, 1743
    -p <interpreter>, 1743
    --python=<interpreter>, 1743
zipfile
    módulo, 529
ZipFile (*clase en zipfile*), 530
zipfileopción de línea de comando
    -c <zipfile> <source1> ... <sourceN>, 538
    --create <zipfile> <source1> ... <sourceN>, 538

-e <zipfile> <output_dir>, 538
--extract <zipfile> <output_dir>, 538
-l <zipfile>, 538
--list <zipfile>, 538
-t <zipfile>, 538
--test <zipfile>, 538
zipimport
módulo, 1853
zipimporter (*clase en zipimport*), 1854
ZipImportError, 1854
ZipInfo (*clase en zipfile*), 530
zlib
módulo, 513
ZLIB_RUNTIME_VERSION (*en el módulo zlib*), 516
ZLIB_VERSION (*en el módulo zlib*), 516
zoneinfo
módulo, 229
ZoneInfo (*clase en zoneinfo*), 231
ZoneInfoNotFoundError, 234
zscore() (*método de statistics.NormalDist*), 378