The Python Language Reference

Versión 3.8.20

Guido van Rossum
and the Python development team

septiembre 08, 2024

Python Software Foundation
Email: docs@python.org

Índice general

Este manual de referencia describe la sintaxis y la «semántica base» del lenguaje. Es conciso, pero intenta ser exacto y completo. La semántica de los tipos de objetos integrados no esenciales y de las funciones y módulos integrados están descritos en library-index. Para obtener una introducción informal al lenguaje, consulte tutorial-index. Para programadores C o C++, existen dos manuales adicionales: extending-index describe detalladamente cómo escribir un módulo de extensión de Python, y c-api-index describe en detalle las interfaces disponibles para los programadores C/C++.

CAPÍTULO 1

Introducción

Este manual de referencia describe el lenguaje de programación Python. No pretende ser un tutorial.

Aunque intento ser lo más preciso posible, prefiero usar español en lugar de especificaciones formales para todo excepto para la sintaxis y el análisis léxico. Ésto debería hacer el documento más comprensible para el lector promedio, pero deja espacio para ambigüedades. De esta manera, si vinieras de Marte e intentases implementar Python utilizando únicamente este documento, tendrías que deducir cosas y, de hecho, probablemente acabarías implementando un lenguaje diferente. Por otro lado, si estás usando Python y te preguntas cuáles son las reglas concretas acerca de un área específica del lenguaje, definitivamente las encontrarás aquí. Si te gustaría ver una definición más formal del lenguaje, tal vez podrías dedicar, voluntariamente, algo de tu tiempo… O inventar una máquina de clonar :-).

Es peligroso añadir muchos detalles de implementación en un documento de referencia: la implementación puede cambiar y otras implementaciones del lenguaje pueden funcionar de forma diferente. Por otro lado, CPython es la implementación de Python más usada (aunque implementaciones alternativas están ganando soporte), y es importante mencionar sus detalles particulares especialmente donde la implementación impone limitaciones adicionales. Por lo tanto, encontrarás pequeñas «notas sobre la implementación» repartidas por todo el texto.

Cada implementación de Python viene con un número de módulos estándar incorporados. Éstos están documentados en library-index. Unos pocos de estos módulos son citados cuando interactúan de forma significativa con la definición del lenguaje.

1.1 Implementaciones alternativas

Aunque hay una implementación de Python que es, de lejos, la más popular, hay otras implementaciones alternativas que pueden ser de particular interés para diferentes audiencias.

Las implementaciones conocidas incluyen:

CPython Es la implementación original, y la más mantenida, de Python y está escrita en C. Las nuevas características del lenguaje normalmente aparecen primero aquí.

Jython Python implementado en Java. Esta implementación se puede usar como lenguaje de *scripting* para aplicaciones Java o se puede usar para crear aplicaciones usando las librerías de clases de Java. A menudo se usa para crear pruebas para librerías de Java. Se puede hallar más información en el sitio web de Jython.

Python for .NET Esta implementación, de hecho, usa la implementación CPython, pero es una aplicación .NET gestionada y usa librerías .NET. Ha sido creada por Brian Lloyd. Para más información ir al sitio web de Python for .NET.

IronPython Un Python alternativo para .NET. Al contrario que Python.NET, esta es una implementación completa de Python que genera lenguaje intermedio (IL) y compila el código directamente en ensamblados de .NET. Ha sido creado por Jim Hugunin, el creador original de Jython. Para más información ver el sitio web de IronPython.

PyPy Una implementación de Python escrita completamente en Python. Soporta varias características avanzadas que no se encuentran en otras implementaciones como el soporte *stackless* y un compilador *Just in Time*. Una de las metas del proyecto es animar a la experimentación con el lenguaje mismo haciendo más fácil la modificación del intérprete (ya que está escrito en Python). Hay información adicional disponible en el sitio web del proyecto PyPy.

Cada una de estas implementaciones varía de una forma u otra del lenguaje tal y como está documentado en este manual, o introduce información específica más allá de lo cubierto por la documentación estándar de Python. Por favor, consulte la documentación específica de cada implementación para saber qué tienes que saber acerca de la implementación específica que uses.

1.2 Notación

Las descripciones del análisis léxico y sintáctico usan una notación gramatical BNF modificada. De tal forma, utilizan el siguiente estilo de definición:

```
name       ::=   lc_letter (lc_letter | "_")*
lc_letter  ::=   "a"..."z"
```

La primera línea dice que un `name` es una `lc_letter` seguida de una secuencia de cero o más `lc_letter`s y guiones bajos. Una `lc_letter` es, a su vez, cualquiera de los caracteres de la `'a'` a la `'z'`. (Esta regla se cumple realmente para los nombres definidos en las reglas léxicas y gramaticales en este documento.)

Cada regla empieza con un nombre (que es el nombre definido por la regla) y `::=`. Una barra vertical (`|`) se usa para separar alternativas; es el operador menos vinculante en esta notación. Un asterisco (`*`) significa cero o más repeticiones del elemento anterior; del mismo modo, un signo más (`+`) significa una o más repeticiones, y una frase entre corchetes (`[]`) significa cero o una ocurrencia (en otras palabras, la frase adjunta es opcional). Los operadores `*` y `+` se vinculan lo más firmemente posible; los paréntesis se usan para agrupar. Las cadenas de caracteres literales están entre comillas. El espacio en blanco sólo es útil para separar tokens. Las reglas normalmente están contenidas en una sola línea; las reglas con varias alternativas se pueden formatear, de forma alternativa, con una barra vertical con cada línea después del primer comienzo.

En las definiciones léxicas (como en el ejemplo anterior), se utilizan dos convenciones más: dos caracteres literales separados por tres puntos significan la elección de cualquier carácter individual en el rango (inclusivo) de caracteres ASCII dado. Una frase entre paréntesis angulares (`<...>`) da una definición informal del símbolo definido; por ejemplo, ésto se puede usar, si fuera necesario, para describir la noción de "carácter de control".

Aunque la notación usada es casi la misma, hay una gran diferencia entre el significado de las definiciones léxicas y sintácticas: una definición léxica opera en los caracteres individuales de la fuente de entrada mientras que una definición sintáctica opera en el flujo de tokens generados por el análisis léxico. Todos los usos de BNF en el siguiente capítulo («Análisis Léxico») son definiciones léxicas. Usos en capítulos posteriores son definiciones sintácticas.

CAPÍTULO 2

Análisis léxico

Un programa de Python es leído por un *parser* (analizador sintáctico). Los datos introducidos en el analizador son un flujo de *tokens*, generados por el *analizador léxico*. Este capítulo describe cómo el analizador léxico desglosa un archivo en tokens.

Python lee el texto del programa como puntos de código Unicode; la codificación de un archivo fuente puede ser dada por una declaración de codificación y por defecto es UTF-8, ver **PEP 3120** para más detalles. Si el archivo fuente no puede ser decodificado, se genera un `SyntaxError`.

2.1 Estructura de línea

Un programa Python se divide en un número de *líneas lógicas*.

2.1.1 Líneas lógicas

El final de una línea lógica está representado por el token NEWLINE (nueva línea). Las declaraciones no pueden cruzar los límites de la línea lógica, excepto cuando la sintaxis permite la utilización de NEWLINE (por ejemplo, entre declaraciones en declaraciones compuestas). Una línea lógica se construye a partir de una o más *líneas físicas* siguiendo las reglas explícitas o implícitas de *unión de líneas*.

2.1.2 Líneas físicas

Una línea física es una secuencia de caracteres terminada por una secuencia de final de línea. En los archivos fuente y las cadenas, se puede utilizar cualquiera de las secuencias de terminación de línea de la plataforma estándar: el formulario Unix que utiliza ASCII LF (salto de línea, por el inglés *linefeed*), el formulario Windows que utiliza la secuencia ASCII CR LF (retorno seguido de salto de línea), o el antiguo formulario Macintosh que utiliza el carácter ASCII CR (retorno). Todas estas formas pueden ser utilizadas por igual, independientemente de la plataforma. El final de la introducción de datos también sirve como un terminador implícito para la línea física final.

Al incrustar Python, las cadenas de código fuente deben ser pasadas a las APIs de Python usando las convenciones estándar de C para los caracteres de nueva línea (el carácter `\n`, que representa ASCII LF, es el terminador de línea).

2.1.3 Comentarios

Un comentario comienza con un carácter de almohadilla (#) que no es parte de un literal de cadena, y termina al final de la línea física. Un comentario implica el final de la línea lógica, a menos que se invoque la regla implícita de unión de líneas. Los comentarios son ignorados por la sintaxis.

2.1.4 Declaración de Codificación

Si un comentario en la primera o segunda línea del script de Python coincide con la expresión regular `coding[=:]\s*([-\w.]+)`, este comentario se procesa como una declaración de codificación; el primer grupo de esta expresión denomina la codificación del archivo de código fuente. La declaración de codificación debe aparecer en una línea propia. Si se trata de la segunda línea, la primera línea debe ser también una línea solamente de comentario. Las formas recomendadas de una expresión de codificación son

```
# -*- coding: <encoding-name> -*-
```

que también es reconocido por GNU Emacs y

```
# vim:fileencoding=<encoding-name>
```

que es reconocido por el VIM de Bram Moolenaar.

Si no se encuentra una declaración de codificación, la codificación por defecto es UTF-8. Además, si los primeros bytes del archivo son la marca de orden de bytes UTF-8 (`b'\xef\xbb\xbf'`), la codificación declarada del archivo es UTF-8 (esto está soportado, entre otros, por el programa **notepad** de Microsoft).

Si se declara una codificación, el nombre de la codificación debe ser reconocido por Python. La codificación se utiliza para todos los análisis léxicos, incluidos los literales de cadena, los comentarios y los identificadores.

2.1.5 Unión explícita de líneas

Dos o más líneas físicas pueden unirse en líneas lógicas utilizando caracteres de barra invertida (\), de la siguiente manera: cuando una línea física termina en una barra invertida que no es parte de literal de cadena o de un comentario, se une con la siguiente formando una sola línea lógica, borrando la barra invertida y el siguiente carácter de fin de línea. Por ejemplo:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a valid date
        return 1
```

Una línea que termina en una barra invertida no puede llevar un comentario. Una barra invertida no continúa un comentario. Una barra invertida no continúa un token excepto para los literales de la cadena (es decir, los tokens que no sean literales de la cadena no pueden ser divididos a través de líneas físicas usando una barra invertida). La barra invertida es ilegal en cualquier parte de una línea fuera del literal de la cadena.

2.1.6 Unión implícita de líneas

Las expresiones entre paréntesis, entre corchetes o entre rizos pueden dividirse en más de una línea física sin usar barras invertidas. Por ejemplo:

```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',      'Juni',       # Dutch names
               'Juli',    'Augustus', 'September',  # for the months
               'Oktober', 'November', 'December']   # of the year
```

Las líneas continuas implícitas pueden llevar comentarios. La sangría de las líneas de continuación no es importante. Se permiten líneas de continuación en blanco. No hay ningún token NEWLINE (nueva línea) entre las líneas de continuación implícitas. Las líneas de continuación implícitas también pueden aparecer dentro de cadenas de triple comilla (ver más adelante); en ese caso no pueden llevar comentarios.

2.1.7 Líneas en blanco

Una línea lógica que contiene sólo espacios, tabulaciones, saltos de página y posiblemente un comentario, es ignorada (es decir, no se genera un símbolo de NEWLINE). Durante la introducción interactiva de declaraciones, el manejo de una línea en blanco puede variar dependiendo de la implementación del bucle de *read-eval-print* (lectura-evaluación-impresión). En el intérprete interactivo estándar, una línea lógica completamente en blanco (es decir, una que no contiene ni siquiera un espacio en blanco o un comentario) termina una declaración de varias líneas.

2.1.8 Sangría

El espacio en blanco (espacios y tabulaciones) al principio de una línea lógica se utiliza para calcular el nivel de sangría de la línea, que a su vez se utiliza para determinar la agrupación de las declaraciones.

Los tabuladores se sustituyen (de izquierda a derecha) por uno a ocho espacios, de manera que el número total de caracteres hasta el reemplazo inclusive es un múltiplo de ocho (se pretende que sea la misma regla que la utilizada por Unix). El número total de espacios que preceden al primer carácter no en blanco determina entonces la sangría de la línea. La sangría no puede dividirse en múltiples líneas físicas utilizando barras invertidas; el espacio en blanco hasta la primera barra invertida determina la sangría.

La indentación se rechaza como inconsistente si un archivo fuente mezcla tabulaciones y espacios de manera que el significado depende del valor de una tabulación en los espacios; un `TabError` se produce en ese caso.

Nota de compatibilidad entre plataformas: debido a la naturaleza de los editores de texto en plataformas que no sean UNIX, no es aconsejable utilizar una mezcla de espacios y tabuladores para la sangría en un solo archivo de origen. También debe tenerse en cuenta que las diferentes plataformas pueden limitar explícitamente el nivel máximo de sangría.

Un carácter *formfeed* puede estar presente al comienzo de la línea; será ignorado para los cálculos de sangría anteriores. Los caracteres *formfeed* que aparecen en otras partes del espacio en blanco inicial tienen un efecto indefinido (por ejemplo, pueden poner a cero el recuento de espacio).

Los niveles de sangría de las líneas consecutivas se utilizan para generar tokens INDENT y DEDENT, utilizando una pila, de la siguiente manera.

Antes de que se lea la primera línea del archivo, se empuja un solo cero en la pila; esto no volverá a saltar. Los números empujados en la pila siempre irán aumentando estrictamente de abajo hacia arriba. Al principio de cada línea lógica, el nivel de sangría de la línea se compara con la parte superior de la pila. Si es igual, no pasa nada. Si es mayor, se empuja en la pila, y se genera un token INDENT. Si es más pequeño, *debe* ser uno de los números de la pila; todos los números de la pila que son más grandes se sacan, y por cada número sacado se genera un token DEDENT. Al final del archivo, se genera un token DEDENT por cada número restante de la pila que sea mayor que cero.

Aquí hay un ejemplo de un código de Python con una correcta (aunque no tan clara) sangría:

```
def perm(l):
        # Compute the list of all permutations of l
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
             s = l[:i] + l[i+1:]
             p = perm(s)
             for x in p:
              r.append(l[i:i+1] + x)
    return r
```

El siguiente ejemplo muestra varios errores de sangría:

```
 def perm(l):                       # error: first line indented
for i in range(len(l)):             # error: not indented
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # error: unexpected indent
        for x in p:
                r.append(l[i:i+1] + x)
            return r                # error: inconsistent dedent
```

(En realidad, los tres primeros errores son detectados por el analizador; sólo el último error es encontrado por el analizador léxico — la sangría de `return r` no coincide con un nivel sacado de la pila.)

2.1.9 Espacios en blanco entre tokens

A excepción del comienzo de una línea lógica o en los literales de cadenas, los caracteres de espacio en blanco, tabulación y formfeed pueden utilizarse indistintamente para separar tokens. Los espacios en blanco se necesitan entre dos tokens sólo si su concatenación podría interpretarse de otra manera como un token diferente (por ejemplo, ab es un token, pero a b corresponde a dos tokens).

2.2 Otros tokens

Además de NEWLINE, INDENT y DEDENT, existen las siguientes categorías de fichas: *identifiers* (identificadores), *keywords* (palabras clave), *literals* (literales), *operators* (operadores) y *delimiters* (delimitadores). Los caracteres de espacio en blanco (distintos de los terminadores de línea, discutidos anteriormente) no son tokens, pero sirven para delimitarlos. En los casos en que exista ambigüedad, un token comprende la cadena más larga posible que forma un token legal cuando se lee de izquierda a derecha.

2.3 Identificadores y palabras clave

Los identificadores (también denominados *nombres*) se describen mediante las siguientes definiciones léxicas.

La sintaxis de los identificadores en Python se basa en el anexo estándar de Unicode UAX-31, con la elaboración y los cambios que se definen a continuación; ver también **PEP 3131** para más detalles.

Dentro del rango ASCII (U+0001..U+007F), los caracteres válidos para los identificadores son los mismos que en Python 2.x: las letras mayúsculas y minúsculas `A` hasta `Z`, el guión bajo _ y los dígitos `0` hasta `9`, salvo el primer carácter.

Python 3.0 introduce caracteres adicionales fuera del rango ASCII (ver **PEP 3131**). Para estos caracteres, la clasificación utiliza la versión de la base de datos de caracteres Unicode incluida en el módulo `unicodedata`.

Los identificadores son de extensión ilimitada. Las mayúsculas y minúsculas son significativas.

```
identifier    ::=  xid_start xid_continue*
id_start      ::=  <all characters in general categories Lu, Ll, Lt, Lm, Lo, Nl, the under
id_continue   ::=  <all characters in id_start, plus characters in the categories Mn, Mc,
xid_start     ::=  <all characters in id_start whose NFKC normalization is in "id_start xi
xid_continue  ::=  <all characters in id_continue whose NFKC normalization is in "id_conti
```

Los códigos de la categoría Unicode mencionados anteriormente representan:

- *Lu* - letras mayúsculas
- *Ll* - letras minúsculas
- *Lt* - letras de *titlecase*
- *Lm* - letras modificadoras
- *Lo* - otras letras
- *Nl* - números de letra
- *Mn* - marcas sin separación
- *Mc* - marcas de combinación de separación
- *Nd* - números decimales
- *Pc* - puntuaciones conectoras
- *Other_ID_Start* - lista explícita de caracteres en PropList.txt para apoyar la compatibilidad hacia atrás
- *Other_ID_Continue* - Así mismo

Todos los identificadores se convierten en la forma normal NFKC mientras se analizan; la comparación de los identificadores se basa en NFKC.

A non-normative HTML file listing all valid identifier characters for Unicode 4.1 can be found at https://www.unicode.org/Public/13.0.0/ucd/DerivedCoreProperties.txt

2.3.1 Palabras clave

Los siguientes identificadores se utilizan como palabras reservadas, o *palabras clave* del idioma, y no pueden utilizarse como identificadores ordinarios. Deben escribirse exactamente como están escritas aquí:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

2.3.2 Clases reservadas de identificadores

Ciertas clases de identificadores (además de las palabras clave) tienen significados especiales. Estas clases se identifican por los patrones de los caracteres de guión bajo que van delante y detrás:

_* No importado por `from module import *`. El identificador especial _ se utiliza en el intérprete interactivo para almacenar el resultado de la última evaluación; se almacena en el módulo `builtins`. Cuando no está en modo interactivo, _ no tiene un significado especial y no está definido. Ver la sección *The import statement*.

Nota: El nombre _ se usa a menudo en conjunción con la internacionalización; consultar la documentación del módulo `gettext` para más información sobre esta convención.

__*__ Nombres definidos por el sistema, conocidos informalmente como nombres «*dunder*». Estos nombres son definidos por el intérprete y su aplicación (incluida la biblioteca estándar). Los nombres actuales del sistema se discuten en la sección *Special method names* y en otros lugares. Es probable que se definan más en futuras versiones de Python. *Cualquier* uso de nombres __*__, en cualquier contexto, que no siga un uso explícitamente documentado, está sujeto a que se rompa sin previo aviso.

__* Nombres de clase privada. Los nombres de esta categoría, cuando se utilizan en el contexto de una definición de clase, se reescriben para utilizar una forma desfigurada que ayude a evitar conflictos de nombres entre los atributos «privados» de las clases base y derivadas. Ver la sección *Identificadores (Nombres)*.

2.4 Literales

Los literales son notaciones para los valores constantes de algunos tipos incorporados.

2.4.1 Literales de cadenas y bytes

Los literales de cadena se describen mediante las siguientes definiciones léxicas:

```
stringliteral   ::=  [stringprefix](shortstring | longstring)
stringprefix    ::=  "r" | "u" | "R" | "U" | "f" | "F"
                     | "fr" | "Fr" | "fR" | "FR" | "rf" | "rF" | "Rf" | "RF"
shortstring     ::=  "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring      ::=  "'''" longstringitem* "'''" | '"""' longstringitem* '"""'
shortstringitem ::=  shortstringchar | stringescapeseq
longstringitem  ::=  longstringchar | stringescapeseq
shortstringchar ::=  <any source character except "\" or newline or the quote>
longstringchar  ::=  <any source character except "\">
stringescapeseq ::=  "\" <any source character>
```

```
bytesliteral    ::=  bytesprefix(shortbytes | longbytes)
bytesprefix     ::=  "b" | "B" | "br" | "Br" | "bR" | "BR" | "rb" | "rB" | "Rb" | "RB"
shortbytes      ::=  "'" shortbytesitem* "'" | '"' shortbytesitem* '"'
longbytes       ::=  "'''" longbytesitem* "'''" | '"""' longbytesitem* '"""'
shortbytesitem  ::=  shortbyteschar | bytesescapeseq
longbytesitem   ::=  longbyteschar | bytesescapeseq
shortbyteschar  ::=  <any ASCII character except "\" or newline or the quote>
```

```
longbyteschar   ::=   <any ASCII character except "\">
bytesescapeseq  ::=   "\" <any ASCII character>
```

Una restricción sintáctica que no se indica en estas producciones es que no se permiten espacios en blanco entre el *stringprefix* o *bytesprefix* y el resto del literal. El conjunto de caracteres fuente está definido por la declaración de codificación; es UTF-8 si no se da una declaración de codificación en el archivo fuente; ver la sección *Declaración de Codificación*.

En lenguaje claro y sencillo: ambos tipos de literales pueden ser encerrados entre comillas simples (') o dobles ("). También pueden estar encerrados en grupos de tres comillas simples o dobles (a las que generalmente se les llama *cadenas de tres comillas*). El carácter de la barra inversa (\) se utiliza para escapar de los caracteres que de otra manera tienen un significado especial, como la línea nueva, la barra inversa en sí misma, o el carácter de comillas.

Los literales de bytes siempre se prefijan con 'b' o 'B'; producen una instancia del tipo bytes en lugar del tipo str. Sólo pueden contener caracteres ASCII; los bytes con un valor numérico de 128 o mayor deben ser expresados con escapes.

Tanto los literales de cadena como de bytes pueden ser prefijados con una letra 'r' o 'R'; tales cadenas se llaman *raw strings* y consideran las barras inversas como caracteres literales. Como resultado, en las cadenas literales, los escapes de '\U' y '\u' en las cadenas sin procesar no son tratados de manera especial. Dado que los literales *raw* de unicode de Python 2.x se comportan de manera diferente a los de Python 3.x, la sintaxis de 'ur' no está soportada.

Nuevo en la versión 3.3: El prefijo 'rb' de literales de bytes raw se ha añadido como sinónimo de 'br'.

Nuevo en la versión 3.3: Se reintrodujo el soporte para el legado unicode literal (u'value') para simplificar el mantenimiento de las bases de código dual Python 2.x y 3.x. Ver **PEP 414** para más información.

Un literal de cadena con 'f' o 'F' en su prefijo es un *formatted string literal*; ver *Literales de cadena formateados*. La 'f' puede combinarse con la 'r', pero no con la 'b' o 'u', por lo que las cadenas *raw* formateadas son posibles, pero los literales de bytes formateados no lo son.

En los literales de triple cita, se permiten (y se retienen) nuevas líneas y citas no escapadas, excepto cuando tres citas no escapadas seguidas finalizan el literal. (Una «cita» es el carácter utilizado para abrir el literal, es decir, ya sea ' o ".)

A menos que un prefijo 'r' o 'R' esté presente, las secuencias de escape en literales de cadena y bytes se interpretan según reglas similares a las usadas por C estándar. Las secuencias de escape reconocidas son:

Secuencia de escape	Significado	Notas
\newline	Barra inversa y línea nueva ignoradas	
\\	Barra inversa (\)	
\'	Comilla simple (')	
\"	Comilla doble (")	
\a	ASCII Bell (BEL)	
\b	ASCII Retroceso (BS)	
\f	ASCII Formfeed (FF)	
\n	ASCII Linefeed (LF)	
\r	ASCII Retorno de carro (CR)	
\t	ASCII Sangría horizontal (TAB)	
\v	ASCII Sangría vertical (VT)	
\ooo	Carácter con valor octal *ooo*	(1,3)
\xhh	Carácter con valor hexadecimal *hh*	(2,3)

Las secuencias de escape que sólo se reconocen en los literales de cadena son:

Secuencia de escape	Significado	Notas
`\N{name}`	El carácter llamado *name* en la base de datos de Unicode	(4)
`\uxxxx`	Carácter con valor hexadecimal de 16 bits *xxxx*	(5)
`\Uxxxxxxxx`	Carácter con valor hexadecimal de 32 bits *xxxxxxxx*	(6)

Notas:

(1) Como en C estándar, se aceptan hasta tres dígitos octales.

(2) A diferencia de C estándar, se requieren exactamente dos dígitos hexadecimales.

(3) En un literal de bytes, los escapes hexadecimal y octal denotan el byte con el valor dado. En un literal de cadena, estos escapes denotan un carácter Unicode con el valor dado.

(4) Distinto en la versión 3.3: Se ha añadido el soporte para los alias de nombres[1].

(5) Se requieren exactamente cuatro dígitos hexadecimales.

(6) Cualquier carácter Unicode puede ser codificado de esta manera. Se requieren exactamente ocho dígitos hexadecimales.

A diferencia de C estándar, todas las secuencias de escape no reconocidas se dejan en la cadena sin cambios, es decir, *la barra invertida se deja en el resultado*. (Este comportamiento es útil para la depuración: si una secuencia de escape se escribe mal, la salida resultante se reconoce más fácilmente como rota). También es importante señalar que las secuencias de escape sólo reconocidas en los literales de cadena caen en la categoría de escapes no reconocidos para los literales de bytes.

> Distinto en la versión 3.6: Las secuencias de escape no reconocidas producen un `DeprecationWarning`.
> En una futura versión de Python serán un `SyntaxWarning` y eventualmente un `SyntaxError`.

Incluso en un literal *raw*, las comillas se pueden escapar con una barra inversa, pero la barra inversa permanece en el resultado; por ejemplo, `r"\""` es un literal de cadena válido que consiste en dos caracteres: una barra inversa y una comilla doble; `r"\"` no es un literal de cadena válido (incluso una cadena en bruto no puede terminar en un número impar de barras inversas). Específicamente, *un literal raw no puede terminar en una sola barra inversa* (ya que la barra inversa se escaparía del siguiente carácter de comillas). Nótese también que una sola barra inversa seguida de una nueva línea se interpreta como esos dos caracteres como parte del literal, *no* como una continuación de línea.

2.4.2 Concatenación de literales de cadena

Se permiten múltiples literales de cadenas o bytes adyacentes (delimitados por espacios en blanco), posiblemente utilizando diferentes convenciones de citas, y su significado es el mismo que su concatenación. Por lo tanto, `"hola" 'mundo'` es equivalente a `"holamundo"`. Esta característica puede ser utilizada para reducir el número de barras inversas necesarias, para dividir largas cadenas convenientemente a través de largas líneas, o incluso para añadir comentarios a partes de las cadenas, por ejemplo:

```
re.compile("[A-Za-z_]"       # letter or underscore
           "[A-Za-z0-9_]*"   # letter, digit or underscore
          )
```

Téngase en cuenta que esta característica se define a nivel sintáctico, pero se implementa en el momento de la compilación. El operador "+" debe ser usado para concatenar expresiones de cadena al momento de la ejecución. Observar también que la concatenación de literales puede utilizar diferentes estilos de citas para cada componente (incluso mezclando cadenas *raw* y cadenas de triple comillado), y los literales de cadena formateados pueden ser concatenados con los literales de cadena simples.

[1] http://www.unicode.org/Public/11.0.0/ucd/NameAliases.txt

2.4.3 Literales de cadena formateados

Nuevo en la versión 3.6.

Un *formatted string literal* o *f-string* es un literal de cadena que se prefija con `'f'` o `'F'`. Estas cadenas pueden contener campos de reemplazo, que son expresiones delimitadas por llaves `{}`. Mientras que otros literales de cadena siempre tienen un valor constante, las cadenas formateadas son realmente expresiones evaluadas en tiempo de ejecución.

Las secuencias de escape se decodifican como en los literales de cadena ordinarios (excepto cuando un literal también se marca como cadena *raw*). Después de la decodificación, la gramática para el contenido de la cadena es:

```
f_string           ::=  (literal_char | "{{" | "}}" | replacement_field)*
replacement_field  ::=  "{" f_expression ["="] ["!" conversion] [":" format_spec] "}"
f_expression       ::=  (conditional_expression | "*" or_expr)
                        ("," conditional_expression | "," "*" or_expr)* [","]
                        | yield_expression
conversion         ::=  "s" | "r" | "a"
format_spec        ::=  (literal_char | NULL | replacement_field)*
literal_char       ::=  <any code point except "{", "}" or NULL>
```

The parts of the string outside curly braces are treated literally, except that any doubled curly braces `'{{'` or `'}}'` are replaced with the corresponding single curly brace. A single opening curly bracket `'{'` marks a replacement field, which starts with a Python expression. To display both the expression text and its value after evaluation, (useful in debugging), an equal sign `'='` may be added after the expression. A conversion field, introduced by an exclamation point `'!'` may follow. A format specifier may also be appended, introduced by a colon `':'`. A replacement field ends with a closing curly bracket `'}'`.

Las expresiones en literales de cadena formateados se tratan como expresiones regulares de Python rodeadas de paréntesis, con algunas excepciones. Una expresión vacía no está permitida, y tanto *lambda* como las expresiones de asignación `:=` deben estar rodeadas de paréntesis explícitos. Las expresiones de sustitución pueden contener saltos de línea (por ejemplo, en cadenas de tres comillas), pero no pueden contener comentarios. Cada expresión se evalúa en el contexto en el que aparece el literal de cadena formateado, en orden de izquierda a derecha.

Distinto en la versión 3.7: Antes de Python 3.7, una expresión *await* y comprensiones que contenían una cláusula *async for* eran ilegales en las expresiones en literales de cadenas formateadas debido a un problema con la implementación.

When the equal sign `'='` is provided, the output will have the expression text, the `'='` and the evaluated value. Spaces after the opening brace `'{'`, within the expression and after the `'='` are all retained in the output. By default, the `'='` causes the `repr()` of the expression to be provided, unless there is a format specified. When a format is specified it defaults to the `str()` of the expression unless a conversion `'!r'` is declared.

Nuevo en la versión 3.8: The equal sign `'='`.

Si se especifica una conversión, el resultado de la evaluación de la expresión se convierte antes del formateo. La conversión `'!s'` llama `str()` al resultado, `'!r'` llama `repr()`, y `'!a'` llama `ascii()`.

El resultado es entonces formateado usando el protocolo `format()`. El especificador de formato se pasa al método *__format__()* del resultado de la expresión o conversión. Se pasa una cadena vacía cuando se omite el especificador de formato. El resultado formateado se incluye entonces en el valor final de toda la cadena.

Los especificadores de formato de nivel superior pueden incluir campos de reemplazo anidados. Estos campos anidados pueden incluir sus propios campos de conversión y especificadores de formato, pero no pueden incluir campos de reemplazo con nidos más profundos. El mini-lenguaje de especificadores de formato es el mismo que el utilizado por el método de *string .format()*.

Los literales de cadena formateados pueden ser concatenados, pero los campos de reemplazo no pueden ser divididos entre los literales.

Algunos ejemplos de literales de cadena formateados:

```
>>> name = "Fred"
>>> f"He said his name is {name!r}."
"He said his name is 'Fred'."
>>> f"He said his name is {repr(name)}."  # repr() is equivalent to !r
"He said his name is 'Fred'."
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested fields
'result:      12.35'
>>> today = datetime(year=2017, month=1, day=27)
>>> f"{today:%B %d, %Y}"  # using date format specifier
'January 27, 2017'
>>> f"{today=:%B %d, %Y}" # using date format specifier and debugging
'today=January 27, 2017'
>>> number = 1024
>>> f"{number:#0x}"  # using integer format specifier
'0x400'
>>> foo = "bar"
>>> f"{ foo = }" # preserves whitespace
" foo = 'bar'"
>>> line = "The mill's closed"
>>> f"{line = }"
'line = "The mill\'s closed"'
>>> f"{line = :20}"
"line = The mill's closed   "
>>> f"{line = !r:20}"
'line = "The mill\'s closed" '
```

Una consecuencia de compartir la misma sintaxis que los literales de cadena regulares es que los caracteres en los campos de reemplazo no deben entrar en conflicto con la comilla usada en el literal de cadena formateado exterior:

```
f"abc {a["x"]} def"    # error: outer string literal ended prematurely
f"abc {a['x']} def"    # workaround: use different quoting
```

Las barras inversas no están permitidas en las expresiones de formato y generarán un error:

```
f"newline: {ord('\n')}"  # raises SyntaxError
```

Para incluir un valor en el que se requiere un escape de barra inversa, hay que crear una variable temporal.

```
>>> newline = ord('\n')
>>> f"newline: {newline}"
'newline: 10'
```

Los literales de cadena formateados no pueden ser usados como cadenas de documentos (*docstrings*), aunque no incluyan expresiones.

```
>>> def foo():
...     f"Not a docstring"
...
>>> foo.__doc__ is None
True
```

Ver también **PEP 498** para la propuesta que añadió literales de cadenas formateados, y `str.format()`, que utiliza un mecanismo de cadenas formateadas relacionado.

2.4.4 Literales numéricos

Hay tres tipos de literales numéricos: números enteros, números de punto flotante y números imaginarios. No hay literales complejos (los números complejos pueden formarse sumando un número real y un número imaginario).

Nótese que los literales numéricos no incluyen un signo; una frase como `-1` es en realidad una expresión compuesta por el operador unario "`-`" y el literal `1`.

2.4.5 Literales enteros

Los literales enteros se describen mediante las siguientes definiciones léxicas:

```
integer       ::=  decinteger | bininteger | octinteger | hexinteger
decinteger    ::=  nonzerodigit (["_"] digit)* | "0"+ (["_"] "0")*
bininteger    ::=  "0" ("b" | "B") (["_"] bindigit)+
octinteger    ::=  "0" ("o" | "O") (["_"] octdigit)+
hexinteger    ::=  "0" ("x" | "X") (["_"] hexdigit)+
nonzerodigit  ::=  "1"..."9"
digit         ::=  "0"..."9"
bindigit      ::=  "0" | "1"
octdigit      ::=  "0"..."7"
hexdigit      ::=  digit | "a"..."f" | "A"..."F"
```

No hay límite para la longitud de los literales enteros aparte de lo que se puede almacenar en la memoria disponible.

Los guiones bajos se ignoran para determinar el valor numérico del literal. Se pueden utilizar para agrupar los dígitos para mejorar la legibilidad. Un guión bajo puede ocurrir entre dígitos y después de especificadores de base como `0x`.

Nótese que no se permiten los ceros a la izquierda en un número decimal que no sea cero. Esto es para desambiguar con los literales octales de estilo C, que Python usaba antes de la versión 3.0.

Algunos ejemplos de literales enteros:

```
7     2147483647                        0o177    0b100110111
3     79228162514264337593543950336     0o377    0xdeadbeef
      100_000_000_000                   0b_1110_0101
```

Distinto en la versión 3.6: Los guiones bajos están ahora permitidos para agrupar en literales.

2.4.6 Literales de punto flotante

Los literales de punto flotante se describen en las siguientes definiciones léxicas:

```
floatnumber    ::=  pointfloat | exponentfloat
pointfloat     ::=  [digitpart] fraction | digitpart "."
exponentfloat  ::=  (digitpart | pointfloat) exponent
digitpart      ::=  digit (["_"] digit)*
fraction       ::=  "." digitpart
exponent       ::=  ("e" | "E") ["+" | "-"] digitpart
```

Nótese que las partes enteras y exponentes siempre se interpretan usando el radix 10. Por ejemplo, `077e010` es legal, y denota el mismo número que `77e10`. El rango permitido de los literales de punto flotante depende de la implementación. Al igual que en los literales enteros, se admiten guiones bajos para la agrupación de dígitos.

Algunos ejemplos de literales de punto flotante:

```
3.14    10.    .001    1e100    3.14e-10    0e0    3.14_15_93
```

Distinto en la versión 3.6: Los guiones bajos están ahora permitidos para agrupar en literales.

2.4.7 Literales imaginarios

Los literales imaginarios se describen en las siguientes definiciones léxicas:

```
imagnumber  ::=  (floatnumber | digitpart) ("j" | "J")
```

Un literal imaginario da un número complejo con una parte real de 0.0. Los números complejos se representan como un par de números de punto flotante y tienen las mismas restricciones en su rango. Para crear un número complejo con una parte real distinta de cero, añada un número de punto flotante, por ejemplo, `(3+4j)`. Algunos ejemplos de literales imaginarios:

```
3.14j   10.j    10j     .001j   1e100j   3.14e-10j   3.14_15_93j
```

2.5 Operadores

Los siguientes tokens son operadores:

```
+       -       *       **      /       //      %       @
<<      >>      &       |       ^       ~       :=
<       >       <=      >=      ==      !=
```

2.6 Delimitadores

Los siguientes tokens sirven como delimitadores en la gramática:

```
(       )       [       ]       {       }
,       :       .       ;       @       =       ->
+=      -=      *=      /=      //=     %=      @=
&=      |=      ^=      >>=     <<=     **=
```

El punto también puede ocurrir en los literales de punto flotante e imaginarios. Una secuencia de tres períodos tiene un significado especial como un literal de elipsis. La segunda mitad de la lista, los operadores de asignación aumentada, sirven léxicamente como delimitadores, pero también realizan una operación.

Los siguientes caracteres ASCII de impresión tienen un significado especial como parte de otros tokens o son de alguna manera significativos para el analizador léxico:

```
'       "       #       \
```

Los siguientes caracteres ASCII de impresión no se utilizan en Python. Su presencia fuera de las cadenas de caracteres y comentarios es un error incondicional:

```
$       ?       `
```

Notas al pie de página

CAPÍTULO 3

Modelo de datos

3.1 Objetos, valores y tipos

Objects son la abstracción de Python para los datos. Todos los datos en un programa Python están representados por objetos o por relaciones entre objetos. (En cierto sentido y de conformidad con el modelo de Von Neumann de una «programa almacenado de computadora», el código también está representado por objetos.)

Cada objeto tiene una identidad, un tipo y un valor. La *identidad* de un objeto nunca cambia una vez que ha sido creado; puede pensar en ello como la dirección del objeto en la memoria. El operador "`is`" compara la identidad de dos objetos; la función `id()` retorna un número entero que representa su identidad.

CPython implementation detail: Para CPython, `id(x)` es la dirección de memoria donde se almacena `x`.

El tipo de un objeto determina las operaciones que admite el objeto (por ejemplo, «¿tiene una longitud?») y también define los posibles valores para los objetos de ese tipo. La función `type()` retorna el tipo de un objeto (que es un objeto en sí mismo). Al igual que su identidad, también el *type* de un objeto es inmutable.[1]

El *valor* de algunos objetos puede cambiar. Se dice que los objetos cuyo valor puede cambiar son *mutables*; Los objetos cuyo valor no se puede modificar una vez que se crean se denominan *inmutables*. (El valor de un objeto contenedor inmutable que contiene una referencia a un objeto mutable puede cambiar cuando se cambia el valor de este último; sin embargo, el contenedor todavía se considera inmutable, porque la colección de objetos que contiene no se puede cambiar. Por lo tanto, la inmutabilidad no es estrictamente lo mismo que tener un valor inmutable, es más sutil). La mutabilidad de un objeto está determinada por su tipo; por ejemplo, los números, las cadenas de caracteres y las tuplas son inmutables, mientras que los diccionarios y las listas son mutables.

Los objetos nunca se destruyen explícitamente; sin embargo, cuando se vuelven inalcanzables, se pueden recolectar basura. Se permite a una implementación posponer la recolección de basura u omitirla por completo; es una cuestión de calidad de la implementación cómo se implementa la recolección de basura, siempre que no se recolecten objetos que todavía sean accesibles.

CPython implementation detail: CPython actualmente utiliza un esquema de conteo de referencias con detección retardada (opcional) de basura enlazada cíclicamente, que recolecta la mayoría de los objetos tan pronto como se vuelven inalcanzables, pero no se garantiza que recolecte basura que contenga referencias circulares. Vea la documentación del

[1] It *is* possible in some cases to change an object's type, under certain controlled conditions. It generally isn't a good idea though, since it can lead to some very strange behaviour if it is handled incorrectly.

módulo `gc` para información sobre el control de la recolección de basura cíclica. Otras implementaciones actúan de manera diferente y CPython puede cambiar. No dependa de la finalización inmediata de los objetos cuando se vuelvan inalcanzables (por lo que siempre debe cerrar los archivos explícitamente).

Tenga en cuenta que el uso de las funciones de rastreo o depuración de la implementación puede mantener activos los objetos que normalmente serían coleccionables. También tenga en cuenta que la captura de una excepción con una sentencia "*try*…*except*" puede mantener objetos activos.

Algunos objetos contienen referencias a recursos «externos» como archivos abiertos o ventanas. Se entiende que estos recursos se liberan cuando el objeto es eliminado por el recolector de basura, pero como no se garantiza que la recolección de basura suceda, dichos objetos también proporcionan una forma explícita de liberar el recurso externo, generalmente un método `close()`. Se recomienda encarecidamente a los programas cerrar explícitamente dichos objetos. La declaración "*try*…*finally*" y la declaración "*with*" proporcionan formas convenientes de hacer esto.

Algunos objetos contienen referencias a otros objetos; estos se llaman *contenedores*. Ejemplos de contenedores son tuplas, listas y diccionarios. Las referencias son parte del valor de un contenedor. En la mayoría de los casos, cuando hablamos del valor de un contenedor, implicamos los valores, no las identidades de los objetos contenidos; sin embargo, cuando hablamos de la mutabilidad de un contenedor, solo se implican las identidades de los objetos contenidos inmediatamente. Entonces, si un contenedor inmutable (como una tupla) contiene una referencia a un objeto mutable, su valor cambia si se cambia ese objeto mutable.

Los tipos afectan a casi todos los aspectos del comportamiento del objeto. Incluso la importancia de la identidad del objeto se ve afectada en cierto sentido: para los tipos inmutables, las operaciones que calculan nuevos valores en realidad pueden retornar una referencia a cualquier objeto existente con el mismo tipo y valor, mientras que para los objetos mutables esto no está permitido. Por ejemplo, al hacer `a = 1; b = 1`, `a` y `b` puede o no referirse al mismo objeto con el valor 1, dependiendo de la implementación, pero al hacer `c = []; d = []`, `c` y `d` se garantiza que se refieren a dos listas vacías diferentes, únicas y recién creadas. (Tenga en cuenta que `c = d = []` asigna el mismo objeto a ambos `c` y `d`.)

3.2 Jerarquía de tipos estándar

A continuación se muestra una lista de los tipos integrados en Python. Los módulos de extensión (escritos en C, Java u otros lenguajes, dependiendo de la implementación) pueden definir tipos adicionales. Las versiones futuras de Python pueden agregar tipos a la jerarquía de tipos (por ejemplo, números racionales, matrices de enteros almacenados de manera eficiente, etc.), aunque tales adiciones a menudo se proporcionarán a través de la biblioteca estándar.

Algunas de las descripciones de tipos a continuación contienen un párrafo que enumera "atributos especiales". Estos son atributos que proporcionan acceso a la implementación y no están destinados para uso general. Su definición puede cambiar en el futuro.

None Este tipo tiene un solo valor. Hay un solo objeto con este valor. Se accede a este objeto a través del nombre incorporado `None`. Se utiliza para indicar la ausencia de un valor en muchas situaciones, por ejemplo, se retorna desde funciones que no retornan nada explícitamente. Su valor de verdad es falso.

NotImplemented Este tipo tiene un solo valor. Hay un solo objeto con este valor. Se accede a este objeto a través del nombre incorporado `NotImplemented`. Los métodos numéricos y los métodos de comparación enriquecidos deberían retornar este valor si no implementan la operación para los operandos proporcionados. (El intérprete intentará la operación reflejada, o alguna otra alternativa, dependiendo del operador). Su valor de verdad es verdadero.

Vea implementing-the-arithmetic-operations para más detalles.

Elipsis Este tipo tiene un solo valor. Hay un solo objeto con este valor. Se accede a este objeto a través del literal `...` o el nombre incorporado `Ellipsis`. Su valor de verdad es verdadero.

numbers.Number Estos son creados por literales numéricos y retornados como resultados por operadores aritméticos y funciones aritméticas integradas. Los objetos numéricos son inmutables; una vez creado su valor nunca cambia.

Los números de Python están, por supuesto, fuertemente relacionados con los números matemáticos, pero están sujetos a las limitaciones de la representación numérica en las computadoras.

The string representations of the numeric classes, computed by `__repr__()` and `__str__()`, have the following properties:

- They are valid numeric literals which, when passed to their class constructor, produce an object having the value of the original numeric.
- The representation is in base 10, when possible.
- Leading zeros, possibly excepting a single zero before a decimal point, are not shown.
- Trailing zeros, possibly excepting a single zero after a decimal point, are not shown.
- A sign is shown only when the number is negative.

Python distingue entre números enteros, números de coma flotante y números complejos:

numbers.Integral Estos representan elementos del conjunto matemático de números enteros (positivo y negativo).

Hay dos tipos de números enteros:

Enteros (int) Estos representan números en un rango ilimitado, sujetos solo a la memoria (virtual) disponible. Para las operaciones de desplazamiento y máscara, se asume una representación binaria, y los números negativos se representan en una variante del complemento de 2 que da la ilusión de una cadena de caracteres infinita de bits con signo que se extiende hacia la izquierda.

Booleanos (bool) Estos representan los valores de verdad Falso y Verdadero. Los dos objetos que representan los valores `False` y `True` son los únicos objetos booleanos. El tipo booleano es un subtipo del tipo entero y los valores booleanos se comportan como los valores 0 y 1 respectivamente, en casi todos los contextos, con la excepción de que cuando se convierten en una cadena de caracteres, las cadenas de caracteres `"False"` o `"True"` son retornadas respectivamente.

Las reglas para la representación de enteros están destinadas a dar la interpretación más significativa de las operaciones de cambio y máscara que involucran enteros negativos.

numbers.Real (float) Estos representan números de punto flotante de precisión doble a nivel de máquina. Está a merced de la arquitectura de la máquina subyacente (y la implementación de C o Java) para el rango aceptado y el manejo del desbordamiento. Python no admite números de coma flotante de precisión simple; el ahorro en el uso del procesador y la memoria, que generalmente son la razón para usarlos, se ven reducidos por la sobrecarga del uso de objetos en Python, por lo que no hay razón para complicar el lenguaje con dos tipos de números de coma flotante.

numbers.Complex (complex) Estos representan números complejos como un par de números de coma flotante de precisión doble a nivel de máquina. Se aplican las mismas advertencias que para los números de coma flotante. Las partes reales e imaginarias de un número complejo `z` se pueden obtener a través de los atributos de solo lectura `z.real` y `z.imag`.

Secuencias Estos representan conjuntos ordenados finitos indexados por números no negativos. La función incorporada `len()` retorna el número de elementos de una secuencia. Cuando la longitud de una secuencia es *n*, el conjunto de índices contiene los números 0, 1, …, *n*-1. El elemento *i* de la secuencia *a* se selecciona mediante `a[i]`.

Las secuencias también admiten segmentación: `a[i:j]` selecciona todos los elementos con índice *k* de modo que *i* <= *k* < *j*. Cuando se usa como una expresión, un segmento es una secuencia del mismo tipo. Esto implica que el conjunto de índices se vuelve a enumerar para que comience en 0.

Algunas secuencias también admiten «segmentación extendida» con un tercer parámetro «paso» : `a[i:j:k]` selecciona todos los elementos de *a* con índice *x* donde `x = i + n*k`, *n* >= `0` y *i* <= *x* < *j*.

Las secuencias se distinguen según su mutabilidad:

Secuencias inmutables Un objeto de un tipo de secuencia inmutable no puede cambiar una vez que se crea. (Si el objeto contiene referencias a otros objetos, estos otros objetos pueden ser mutables y pueden cambiarse; sin embargo, la colección de objetos a los que hace referencia directamente un objeto inmutable no puede cambiar).

Los siguientes tipos son secuencias inmutables:

Cadenas de caracteres Una cadena de caracteres es una secuencia de valores que representan puntos de código *Unicode*. Todos los puntos de código en el rango `U+0000 - U+10FFFF` se puede representar en una cadena de caracteres. Python no tiene un tipo `char`; en cambio, cada punto de código en la cadena de caracteres se representa como un objeto de cadena de caracteres con longitud `1`. La función incorporada `ord()` convierte un punto de código de su forma de cadena de caracteres a un entero en el rango `0 - 10FFFF`; la función `chr()` convierte un entero en el rango `0 - 10FFFF` a la cadena de caracteres correspondiente de longitud `1`. `str.encode()` se puede usar para convertir un objeto de tipo `str` a `bytes` usando la codificación de texto dada, y `bytes.decode()` se puede usar para lograr el caso inverso.

Tuplas Los elementos de una tupla son objetos arbitrarios de Python. Las tuplas de dos o más elementos están formadas por listas de expresiones separadas por comas. Se puede formar una tupla de un elemento (un "singleton") al colocar una coma en una expresión (una expresión en sí misma no crea una tupla, ya que los paréntesis deben ser utilizables para agrupar expresiones). Una tupla vacía puede estar formada por un par de paréntesis vacío.

Bytes Un objeto de bytes es una colección inmutable. Los elementos son bytes de 8 bits, representados por enteros en el rango 0 <= x <256. Literales de bytes (como `b'abc'`) y el constructor incorporado `bytes()` se puede utilizar para crear objetos de bytes. Además, los objetos de bytes se pueden decodificar en cadenas de caracteres a través del método `decode()`.

Secuencias mutables Las secuencias mutables se pueden cambiar después de su creación. Las anotaciones de suscripción y segmentación se pueden utilizar como el objetivo de asignaciones y declaraciones *del* (eliminar).

Actualmente hay dos tipos intrínsecos de secuencias mutable:

Listas The items of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets. (Note that there are no special cases needed to form lists of length 0 or 1.)

Colecciones de bytes Un objeto bytearray es una colección mutable. Son creados por el constructor incorporado `bytearray()`. Además de ser mutables (y, por lo tanto, inquebrantable), las colecciones de bytes proporcionan la misma interfaz y funcionalidad que los objetos inmutables `bytes`.

El módulo de extensión `array` proporciona un ejemplo adicional de un tipo de secuencia mutable, al igual que el módulo `collections`.

Tipos de conjuntos Estos representan conjuntos finitos no ordenados de objetos únicos e inmutables. Como tal, no pueden ser indexados por ningún *subscript*. Sin embargo, pueden repetirse y la función incorporada `len()` retorna el número de elementos en un conjunto. Los usos comunes de los conjuntos son pruebas rápidas de membresía, eliminación de duplicados de una secuencia y cálculo de operaciones matemáticas como intersección, unión, diferencia y diferencia simétrica.

Para elementos del conjunto, se aplican las mismas reglas de inmutabilidad que para las claves de diccionario. Tenga en cuenta que los tipos numéricos obedecen las reglas normales para la comparación numérica: si dos números se comparan igual (por ejemplo, `1` y `1.0`), solo uno de ellos puede estar contenido en un conjunto.

Actualmente hay dos tipos de conjuntos intrínsecos:

Conjuntos Estos representan un conjunto mutable. Son creados por el constructor incorporado `set()` y puede ser modificado posteriormente por varios métodos, como `add()`.

Conjuntos congelados Estos representan un conjunto inmutable. Son creados por el constructor incorporado `frozenset()`. Como un conjunto congelado es inmutable y *hashable*, se puede usar nuevamente como un elemento de otro conjunto o como una clave de un diccionario.

Mapeos Estos representan conjuntos finitos de objetos indexados por conjuntos de índices arbitrarios. La notación de subíndice `a[k]` selecciona el elemento indexado por `k` del mapeo `a`; esto se puede usar en expresiones y como el objetivo de asignaciones o declaraciones *del*. La función incorporada `len()` retorna el número de elementos en un mapeo.

Actualmente hay un único tipo de mapeo intrínseco:

Diccionarios Estos representan conjuntos finitos de objetos indexados por valores casi arbitrarios. Los únicos tipos de valores no aceptables como claves son valores que contienen listas o diccionarios u otros tipos mutables que se comparan por valor en lugar de por identidad de objeto, la razón es que la implementación eficiente de los diccionarios requiere que el valor *hash* de una clave permanezca constante. Los tipos numéricos utilizados para las claves obedecen las reglas normales para la comparación numérica: si dos números se comparan igual (por ejemplo, `1` y `1.0`) entonces se pueden usar indistintamente para indexar la misma entrada del diccionario.

Los diccionarios conservan el orden de inserción, lo que significa que las claves se mantendrán en el mismo orden en que se agregaron secuencialmente sobre el diccionario. Reemplazar una clave existente no cambia el orden, sin embargo, eliminar una clave y volver a insertarla la agregará al final en lugar de mantener su lugar anterior.

Los diccionarios son mutables; pueden ser creados por la notación `{...}` (vea la sección *Despliegues de diccionario*).

Los módulos de extensión `dbm.ndbm` y `dbm.gnu` proporcionan ejemplos adicionales de tipos de mapeo, al igual que el módulo `collections`.

Distinto en la versión 3.7: Los diccionarios no conservaban el orden de inserción en las versiones de Python anteriores a 3.6. En CPython 3.6, el orden de inserción se conserva, pero se consideró un detalle de implementación en ese momento en lugar de una garantía de idioma.

Tipos invocables Estos son los tipos a los que la operación de llamada de función (vea la sección *Invocaciones*) puede ser aplicado:

Funciones definidas por el usuario A user-defined function object is created by a function definition (see section *Definiciones de funciones*). It should be called with an argument list containing the same number of items as the function's formal parameter list.

Atributos especiales:

Atributo	Significado	
`__doc__`	El texto de documentación de la función, o `None` si no está disponible; no heredado por subclases.	Escribible
`__name__`	El nombre de la función.	Escribible
`__qualname__`	Las funciones *qualified name*. Nuevo en la versión 3.3.	Escribible
`__module__`	El nombre del módulo en el que se definió la función, o `None` si no está disponible.	Escribible
`__defaults__`	Una tupla que contiene valores de argumento predeterminados para aquellos argumentos que tienen valores predeterminados, o `None` si ningún argumento tiene un valor predeterminado.	Escribible
`__code__`	El objeto de código que representa el cuerpo de la función compilada.	Escribible
`__globals__`	Una referencia al diccionario que contiene las variables globales de la función — el espacio de nombres global del módulo en el que se definió la función.	Solo lectura
`__dict__`	El espacio de nombres que admite atributos de funciones arbitrarias.	Escribible
`__closure__`	`None` o una tupla de celdas que contienen enlaces para las variables libres de la función. Vea a continuación para obtener información sobre el atributo `cell_contents`.	Solo lectura
`__annotations__`	Un diccionario que contiene anotaciones de parámetros. Las claves del dict son los nombres de los parámetros, y `'return'` para la anotación de retorno, si se proporciona.	Escribible
`__kwdefaults__`	Un diccionario que contiene valores predeterminados para parámetros de solo palabras clave.	Escribible

La mayoría de los atributos etiquetados «Escribible» verifican el tipo del valor asignado.

Function objects also support getting and setting arbitrary attributes, which can be used, for example, to attach metadata to functions. Regular attribute dot-notation is used to get and set such attributes. *Note that the current implementation only supports function attributes on user-defined functions. Function attributes on built-in functions may be supported in the future.*

Un objeto de celda tiene el atributo `cell_contents`. Esto se puede usar para obtener el valor de la celda, así como para establecer el valor.

Se puede recuperar información adicional sobre la definición de una función desde su objeto de código; Vea la descripción de los tipos internos a continuación. El tipo `cell` puede ser accedido en el módulo `types`.

Métodos de instancia An instance method object combines a class, a class instance and any callable object (normally a user-defined function).

Special read-only attributes: `__self__` is the class instance object, `__func__` is the function object; `__doc__` is the method's documentation (same as `__func__.__doc__`); `__name__` is the method name (same as `__func__.__name__`); `__module__` is the name of the module the method was defined in, or `None` if unavailable.

Methods also support accessing (but not setting) the arbitrary function attributes on the underlying function object.

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined function object or a class method object.

When an instance method object is created by retrieving a user-defined function object from a class via one of its instances, its `__self__` attribute is the instance, and the method object is said to be bound. The new method's `__func__` attribute is the original function object.

When an instance method object is created by retrieving a class method object from a class or instance, its `__self__` attribute is the class itself, and its `__func__` attribute is the function object underlying the class method.

When an instance method object is called, the underlying function (`__func__`) is called, inserting the class instance (`__self__`) in front of the argument list. For instance, when `C` is a class which contains a definition for a function `f()`, and `x` is an instance of `C`, calling `x.f(1)` is equivalent to calling `C.f(x, 1)`.

When an instance method object is derived from a class method object, the «class instance» stored in `__self__` will actually be the class itself, so that calling either `x.f(1)` or `C.f(1)` is equivalent to calling `f(C,1)` where `f` is the underlying function.

Note that the transformation from function object to instance method object happens each time the attribute is retrieved from the instance. In some cases, a fruitful optimization is to assign the attribute to a local variable and call that local variable. Also notice that this transformation only happens for user-defined functions; other callable objects (and all non-callable objects) are retrieved without transformation. It is also important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this *only* happens when the function is an attribute of the class.

Generator functions A function or method which uses the *yield* statement (see section *The yield statement*) is called a *generator function*. Such a function, when called, always returns an iterator object which can be used to execute the body of the function: calling the iterator's `iterator.__next__()` method will cause the function to execute until it provides a value using the `yield` statement. When the function executes a *return* statement or falls off the end, a `StopIteration` exception is raised and the iterator will have reached the end of the set of values to be returned.

Coroutine functions A function or method which is defined using *async def* is called a *coroutine function*. Such a function, when called, returns a *coroutine* object. It may contain *await* expressions, as well as *async with* and *async for* statements. See also the *Coroutine Objects* section.

Asynchronous generator functions A function or method which is defined using *async def* and which uses the *yield* statement is called a *asynchronous generator function*. Such a function, when called, returns an asynchronous iterator object which can be used in an *async for* statement to execute the body of the function.

Calling the asynchronous iterator's `aiterator.__anext__()` method will return an *awaitable* which when awaited will execute until it provides a value using the *yield* expression. When the function executes an empty *return* statement or falls off the end, a `StopAsyncIteration` exception is raised and the asynchronous iterator will have reached the end of the set of values to be yielded.

Built-in functions A built-in function object is a wrapper around a C function. Examples of built-in functions are `len()` and `math.sin()` (`math` is a standard built-in module). The number and type of the arguments are determined by the C function. Special read-only attributes: `__doc__` is the function's documentation string, or `None` if unavailable; `__name__` is the function's name; `__self__` is set to `None` (but see the next item); `__module__` is the name of the module the function was defined in or `None` if unavailable.

Built-in methods This is really a different disguise of a built-in function, this time containing an object passed to the C function as an implicit extra argument. An example of a built-in method is `alist.append()`, assuming *alist* is a list object. In this case, the special read-only attribute `__self__` is set to the object denoted by *alist*.

Classes Classes are callable. These objects normally act as factories for new instances of themselves, but variations are possible for class types that override *__new__()*. The arguments of the call are passed to *__new__()* and, in the typical case, to *__init__()* to initialize the new instance.

Class Instances Instances of arbitrary classes can be made callable by defining a *__call__()* method in their class.

Modules Modules are a basic organizational unit of Python code, and are created by the *import system* as invoked either by the *import* statement, or by calling functions such as `importlib.import_module()` and built-

in __import__(). A module object has a namespace implemented by a dictionary object (this is the dictionary referenced by the __globals__ attribute of functions defined in the module). Attribute references are translated to lookups in this dictionary, e.g., m.x is equivalent to m.__dict__["x"]. A module object does not contain the code object used to initialize the module (since it isn't needed once the initialization is done).

Attribute assignment updates the module's namespace dictionary, e.g., m.x = 1 is equivalent to m.__dict__["x"] = 1.

Predefined (writable) attributes: *__name__* is the module's name; __doc__ is the module's documentation string, or None if unavailable; __annotations__ (optional) is a dictionary containing *variable annotations* collected during module body execution; *__file__* is the pathname of the file from which the module was loaded, if it was loaded from a file. The *__file__* attribute may be missing for certain types of modules, such as C modules that are statically linked into the interpreter; for extension modules loaded dynamically from a shared library, it is the pathname of the shared library file.

Special read-only attribute: __dict__ is the module's namespace as a dictionary object.

CPython implementation detail: Because of the way CPython clears module dictionaries, the module dictionary will be cleared when the module falls out of scope even if the dictionary still has live references. To avoid this, copy the dictionary or keep the module around while using its dictionary directly.

Custom classes Custom class types are typically created by class definitions (see section *Definiciones de clase*). A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary, e.g., C.x is translated to C.__dict__["x"] (although there are a number of hooks which allow for other means of locating attributes). When the attribute name is not found there, the attribute search continues in the base classes. This search of the base classes uses the C3 method resolution order which behaves correctly even in the presence of "diamond" inheritance structures where there are multiple inheritance paths leading back to a common ancestor. Additional details on the C3 MRO used by Python can be found in the documentation accompanying the 2.3 release at https://www.python.org/download/releases/2.3/mro/.

When a class attribute reference (for class C, say) would yield a class method object, it is transformed into an instance method object whose __self__ attribute is C. When it would yield a static method object, it is transformed into the object wrapped by the static method object. See section *Implementing Descriptors* for another way in which attributes retrieved from a class may differ from those actually contained in its __dict__.

Class attribute assignments update the class's dictionary, never the dictionary of a base class.

A class object can be called (see above) to yield a class instance (see below).

Special attributes: __name__ is the class name; __module__ is the module name in which the class was defined; __dict__ is the dictionary containing the class's namespace; __bases__ is a tuple containing the base classes, in the order of their occurrence in the base class list; __doc__ is the class's documentation string, or None if undefined; __annotations__ (optional) is a dictionary containing *variable annotations* collected during class body execution.

Class instances A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object, it is transformed into an instance method object whose __self__ attribute is the instance. Static method and class method objects are also transformed; see above under «Classes». See section *Implementing Descriptors* for another way in which attributes of a class retrieved via its instances may differ from the objects actually stored in the class's __dict__. If no class attribute is found, and the object's class has a *__getattr__()* method, that is called to satisfy the lookup.

Attribute assignments and deletions update the instance's dictionary, never a class's dictionary. If the class has a *__setattr__()* or *__delattr__()* method, this is called instead of updating the instance dictionary directly.

Class instances can pretend to be numbers, sequences, or mappings if they have methods with certain special names. See section *Special method names*.

Special attributes: `__dict__` is the attribute dictionary; `__class__` is the instance's class.

I/O objects (also known as file objects) A *file object* represents an open file. Various shortcuts are available to create file objects: the `open()` built-in function, and also `os.popen()`, `os.fdopen()`, and the `makefile()` method of socket objects (and perhaps by other functions or methods provided by extension modules).

The objects `sys.stdin`, `sys.stdout` and `sys.stderr` are initialized to file objects corresponding to the interpreter's standard input, output and error streams; they are all open in text mode and therefore follow the interface defined by the `io.TextIOBase` abstract class.

Internal types A few types used internally by the interpreter are exposed to the user. Their definitions may change with future versions of the interpreter, but they are mentioned here for completeness.

Code objects Code objects represent *byte-compiled* executable Python code, or *bytecode*. The difference between a code object and a function object is that the function object contains an explicit reference to the function's globals (the module in which it was defined), while a code object contains no context; also the default argument values are stored in the function object, not in the code object (because they represent values calculated at run-time). Unlike function objects, code objects are immutable and contain no references (directly or indirectly) to mutable objects.

Special read-only attributes: `co_name` gives the function name; `co_argcount` is the total number of positional arguments (including positional-only arguments and arguments with default values); `co_posonlyargcount` is the number of positional-only arguments (including arguments with default values); `co_kwonlyargcount` is the number of keyword-only arguments (including arguments with default values); `co_nlocals` is the number of local variables used by the function (including arguments); `co_varnames` is a tuple containing the names of the local variables (starting with the argument names); `co_cellvars` is a tuple containing the names of local variables that are referenced by nested functions; `co_freevars` is a tuple containing the names of free variables; `co_code` is a string representing the sequence of bytecode instructions; `co_consts` is a tuple containing the literals used by the bytecode; `co_names` is a tuple containing the names used by the bytecode; `co_filename` is the filename from which the code was compiled; `co_firstlineno` is the first line number of the function; `co_lnotab` is a string encoding the mapping from bytecode offsets to line numbers (for details see the source code of the interpreter); `co_stacksize` is the required stack size; `co_flags` is an integer encoding a number of flags for the interpreter.

The following flag bits are defined for `co_flags`: bit `0x04` is set if the function uses the `*arguments` syntax to accept an arbitrary number of positional arguments; bit `0x08` is set if the function uses the `**keywords` syntax to accept arbitrary keyword arguments; bit `0x20` is set if the function is a generator.

Future feature declarations (`from __future__ import division`) also use bits in `co_flags` to indicate whether a code object was compiled with a particular feature enabled: bit `0x2000` is set if the function was compiled with future division enabled; bits `0x10` and `0x1000` were used in earlier versions of Python.

Other bits in `co_flags` are reserved for internal use.

If a code object represents a function, the first item in `co_consts` is the documentation string of the function, or `None` if undefined.

Frame objects Frame objects represent execution frames. They may occur in traceback objects (see below), and are also passed to registered trace functions.

Special read-only attributes: `f_back` is to the previous stack frame (towards the caller), or `None` if this is the bottom stack frame; `f_code` is the code object being executed in this frame; `f_locals` is the dictionary used to look up local variables; `f_globals` is used for global variables; `f_builtins` is used for built-in (intrinsic) names; `f_lasti` gives the precise instruction (this is an index into the bytecode string of the code object).

Accessing `f_code` raises an auditing event `object.__getattr__` with arguments `obj` and `"f_code"`.

Special writable attributes: `f_trace`, if not `None`, is a function called for various events during code execution (this is used by the debugger). Normally an event is triggered for each new source line - this can be disabled by setting `f_trace_lines` to `False`.

Implementations *may* allow per-opcode events to be requested by setting `f_trace_opcodes` to `True`. Note that this may lead to undefined interpreter behaviour if exceptions raised by the trace function escape to the function being traced.

`f_lineno` is the current line number of the frame — writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a Jump command (aka Set Next Statement) by writing to f_lineno.

Frame objects support one method:

`frame.`**`clear`**`()`
: This method clears all references to local variables held by the frame. Also, if the frame belonged to a generator, the generator is finalized. This helps break reference cycles involving frame objects (for example when catching an exception and storing its traceback for later use).

  `RuntimeError` is raised if the frame is currently executing.

  Nuevo en la versión 3.4.

Traceback objects Traceback objects represent a stack trace of an exception. A traceback object is implicitly created when an exception occurs, and may also be explicitly created by calling `types.TracebackType`.

For implicitly created tracebacks, when the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section *La sentencia try*.) It is accessible as the third item of the tuple returned by `sys.exc_info()`, and as the `__traceback__` attribute of the caught exception.

When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as `sys.last_traceback`.

For explicitly created tracebacks, it is up to the creator of the traceback to determine how the `tb_next` attributes should be linked to form a full stack trace.

Special read-only attributes: `tb_frame` points to the execution frame of the current level; `tb_lineno` gives the line number where the exception occurred; `tb_lasti` indicates the precise instruction. The line number and last instruction in the traceback may differ from the line number of its frame object if the exception occurred in a *`try`* statement with no matching except clause or with a finally clause.

Accessing `tb_frame` raises an auditing event `object.__getattr__` with arguments `obj` and `"tb_frame"`.

Special writable attribute: `tb_next` is the next level in the stack trace (towards the frame where the exception occurred), or `None` if there is no next level.

Distinto en la versión 3.7: Traceback objects can now be explicitly instantiated from Python code, and the `tb_next` attribute of existing instances can be updated.

Slice objects Slice objects are used to represent slices for *`__getitem__()`* methods. They are also created by the built-in `slice()` function.

Special read-only attributes: `start` is the lower bound; `stop` is the upper bound; `step` is the step value; each is `None` if omitted. These attributes can have any type.

Slice objects support one method:

slice.**indices**(*self*, *length*)
This method takes a single integer argument *length* and computes information about the slice that the slice object would describe if applied to a sequence of *length* items. It returns a tuple of three integers; respectively these are the *start* and *stop* indices and the *step* or stride length of the slice. Missing or out-of-bounds indices are handled in a manner consistent with regular slices.

Static method objects Static method objects provide a way of defeating the transformation of function objects to method objects described above. A static method object is a wrapper around any other object, usually a user-defined method object. When a static method object is retrieved from a class or a class instance, the object actually returned is the wrapped object, which is not subject to any further transformation. Static method objects are not themselves callable, although the objects they wrap usually are. Static method objects are created by the built-in `staticmethod()` constructor.

Class method objects A class method object, like a static method object, is a wrapper around another object that alters the way in which that object is retrieved from classes and class instances. The behaviour of class method objects upon such retrieval is described above, under «User-defined methods». Class method objects are created by the built-in `classmethod()` constructor.

3.3 Special method names

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python's approach to *operator overloading*, allowing classes to define their own behavior with respect to language operators. For instance, if a class defines a method named *__getitem__()*, and `x` is an instance of this class, then `x[i]` is roughly equivalent to `type(x).__getitem__(x, i)`. Except where mentioned, attempts to execute an operation raise an exception when no appropriate method is defined (typically `AttributeError` or `TypeError`).

Setting a special method to `None` indicates that the corresponding operation is not available. For example, if a class sets *__iter__()* to `None`, the class is not iterable, so calling `iter()` on its instances will raise a `TypeError` (without falling back to *__getitem__()*).[2]

When implementing a class that emulates any built-in type, it is important that the emulation only be implemented to the degree that it makes sense for the object being modelled. For example, some sequences may work well with retrieval of individual elements, but extracting a slice may not make sense. (One example of this is the `NodeList` interface in the W3C's Document Object Model.)

3.3.1 Basic customization

object.**__new__**(*cls*[, ...])
Called to create a new instance of class *cls*. *__new__()* is a static method (special-cased so you need not declare it as such) that takes the class of which an instance was requested as its first argument. The remaining arguments are those passed to the object constructor expression (the call to the class). The return value of *__new__()* should be the new object instance (usually an instance of *cls*).

Typical implementations create a new instance of the class by invoking the superclass's *__new__()* method using `super().__new__(cls[, ...])` with appropriate arguments and then modifying the newly-created instance as necessary before returning it.

If *__new__()* is invoked during object construction and it returns an instance of *cls*, then the new instance's *__init__()* method will be invoked like `__init__(self[, ...])`, where *self* is the new instance and the remaining arguments are the same as were passed to the object constructor.

[2] The *__hash__()*, *__iter__()*, *__reversed__()*, and *__contains__()* methods have special handling for this; others will still raise a `TypeError`, but may do so by relying on the behavior that `None` is not callable.

If *__new__()* does not return an instance of *cls*, then the new instance's *__init__()* method will not be invoked.

__new__() is intended mainly to allow subclasses of immutable types (like int, str, or tuple) to customize instance creation. It is also commonly overridden in custom metaclasses in order to customize class creation.

`object.`**`__init__`**(*self*[, ...])
Called after the instance has been created (by *__new__()*), but before it is returned to the caller. The arguments are those passed to the class constructor expression. If a base class has an *__init__()* method, the derived class's *__init__()* method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example: `super().__init__([args...])`.

Because *__new__()* and *__init__()* work together in constructing objects (*__new__()* to create it, and *__init__()* to customize it), no non-`None` value may be returned by *__init__()*; doing so will cause a `TypeError` to be raised at runtime.

`object.`**`__del__`**(*self*)
Called when the instance is about to be destroyed. This is also called a finalizer or (improperly) a destructor. If a base class has a *__del__()* method, the derived class's *__del__()* method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.

It is possible (though not recommended!) for the *__del__()* method to postpone destruction of the instance by creating a new reference to it. This is called object *resurrection*. It is implementation-dependent whether *__del__()* is called a second time when a resurrected object is about to be destroyed; the current *CPython* implementation only calls it once.

It is not guaranteed that *__del__()* methods are called for objects that still exist when the interpreter exits.

Nota: `del x` doesn't directly call `x.__del__()` — the former decrements the reference count for `x` by one, and the latter is only called when `x`'s reference count reaches zero.

CPython implementation detail: It is possible for a reference cycle to prevent the reference count of an object from going to zero. In this case, the cycle will be later detected and deleted by the *cyclic garbage collector*. A common cause of reference cycles is when an exception has been caught in a local variable. The frame's locals then reference the exception, which references its own traceback, which references the locals of all frames caught in the traceback.

Ver también:

Documentation for the `gc` module.

Advertencia: Due to the precarious circumstances under which *__del__()* methods are invoked, exceptions that occur during their execution are ignored, and a warning is printed to `sys.stderr` instead. In particular:

- *__del__()* can be invoked when arbitrary code is being executed, including from any arbitrary thread. If *__del__()* needs to take a lock or invoke any other blocking resource, it may deadlock as the resource may already be taken by the code that gets interrupted to execute *__del__()*.
- *__del__()* can be executed during interpreter shutdown. As a consequence, the global variables it needs to access (including other modules) may already have been deleted or set to `None`. Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the *__del__()* method is called.

`object.`**`__repr__`**(*self*)
Called by the `repr()` built-in function to compute the «official» string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with

the same value (given an appropriate environment). If this is not possible, a string of the form `<...some useful description...>` should be returned. The return value must be a string object. If a class defines *__repr__()* but not *__str__()*, then *__repr__()* is also used when an «informal» string representation of instances of that class is required.

This is typically used for debugging, so it is important that the representation is information-rich and unambiguous.

`object.`**`__str__`**(*self*)
Called by `str(object)` and the built-in functions `format()` and `print()` to compute the «informal» or nicely printable string representation of an object. The return value must be a string object.

This method differs from *object.__repr__()* in that there is no expectation that *__str__()* return a valid Python expression: a more convenient or concise representation can be used.

The default implementation defined by the built-in type `object` calls *object.__repr__()*.

`object.`**`__bytes__`**(*self*)
Called by bytes to compute a byte-string representation of an object. This should return a `bytes` object.

`object.`**`__format__`**(*self*, *format_spec*)
Called by the `format()` built-in function, and by extension, evaluation of *formatted string literals* and the `str.format()` method, to produce a «formatted» string representation of an object. The *format_spec* argument is a string that contains a description of the formatting options desired. The interpretation of the *format_spec* argument is up to the type implementing *__format__()*, however most classes will either delegate formatting to one of the built-in types, or use a similar formatting option syntax.

See formatspec for a description of the standard formatting syntax.

The return value must be a string object.

Distinto en la versión 3.4: The __format__ method of `object` itself raises a `TypeError` if passed any non-empty string.

Distinto en la versión 3.7: `object.__format__(x, '')` is now equivalent to `str(x)` rather than `format(str(self), '')`.

`object.`**`__lt__`**(*self*, *other*)
`object.`**`__le__`**(*self*, *other*)
`object.`**`__eq__`**(*self*, *other*)
`object.`**`__ne__`**(*self*, *other*)
`object.`**`__gt__`**(*self*, *other*)
`object.`**`__ge__`**(*self*, *other*)
These are the so-called «rich comparison» methods. The correspondence between operator symbols and method names is as follows: `x<y` calls `x.__lt__(y)`, `x<=y` calls `x.__le__(y)`, `x==y` calls `x.__eq__(y)`, `x!=y` calls `x.__ne__(y)`, `x>y` calls `x.__gt__(y)`, and `x>=y` calls `x.__ge__(y)`.

A rich comparison method may return the singleton `NotImplemented` if it does not implement the operation for a given pair of arguments. By convention, `False` and `True` are returned for a successful comparison. However, these methods can return any value, so if the comparison operator is used in a Boolean context (e.g., in the condition of an `if` statement), Python will call `bool()` on the value to determine if the result is true or false.

By default, `object` implements *__eq__()* by using `is`, returning `NotImplemented` in the case of a false comparison: `True if x is y else NotImplemented`. For *__ne__()*, by default it delegates to *__eq__()* and inverts the result unless it is `NotImplemented`. There are no other implied relationships among the comparison operators or default implementations; for example, the truth of `(x<y or x==y)` does not imply `x<=y`. To automatically generate ordering operations from a single root operation, see `functools.total_ordering()`.

See the paragraph on *__hash__()* for some important notes on creating *hashable* objects which support custom comparison operations and are usable as dictionary keys.

There are no swapped-argument versions of these methods (to be used when the left argument does not support the operation but the right argument does); rather, *__lt__()* and *__gt__()* are each other's reflection, *__le__()* and *__ge__()* are each other's reflection, and *__eq__()* and *__ne__()* are their own reflection. If the operands are of different types, and right operand's type is a direct or indirect subclass of the left operand's type, the reflected method of the right operand has priority, otherwise the left operand's method has priority. Virtual subclassing is not considered.

`object.`**`__hash__`**(*self*)

Called by built-in function `hash()` and for operations on members of hashed collections including `set`, `frozenset`, and `dict`. *__hash__()* should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a tuple and hashing the tuple. Example:

```
def __hash__(self):
    return hash((self.name, self.nick, self.color))
```

Nota: `hash()` truncates the value returned from an object's custom *__hash__()* method to the size of a `Py_ssize_t`. This is typically 8 bytes on 64-bit builds and 4 bytes on 32-bit builds. If an object's *__hash__()* must interoperate on builds of different bit sizes, be sure to check the width on all supported builds. An easy way to do this is with `python -c "import sys; print(sys.hash_info.width)"`.

If a class does not define an *__eq__()* method it should not define a *__hash__()* operation either; if it defines *__eq__()* but not *__hash__()*, its instances will not be usable as items in hashable collections. If a class defines mutable objects and implements an *__eq__()* method, it should not implement *__hash__()*, since the implementation of hashable collections requires that a key's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

User-defined classes have *__eq__()* and *__hash__()* methods by default; with them, all objects compare unequal (except with themselves) and `x.__hash__()` returns an appropriate value such that `x == y` implies both that `x is y` and `hash(x) == hash(y)`.

A class that overrides *__eq__()* and does not define *__hash__()* will have its *__hash__()* implicitly set to `None`. When the *__hash__()* method of a class is `None`, instances of the class will raise an appropriate `TypeError` when a program attempts to retrieve their hash value, and will also be correctly identified as unhashable when checking `isinstance(obj, collections.abc.Hashable)`.

If a class that overrides *__eq__()* needs to retain the implementation of *__hash__()* from a parent class, the interpreter must be told this explicitly by setting `__hash__ = <ParentClass>.__hash__`.

If a class that does not override *__eq__()* wishes to suppress hash support, it should include `__hash__ = None` in the class definition. A class which defines its own *__hash__()* that explicitly raises a `TypeError` would be incorrectly identified as hashable by an `isinstance(obj, collections.abc.Hashable)` call.

Nota: By default, the *__hash__()* values of str and bytes objects are «salted» with an unpredictable random value. Although they remain constant within an individual Python process, they are not predictable between repeated invocations of Python.

This is intended to provide protection against a denial-of-service caused by carefully-chosen inputs that exploit the worst case performance of a dict insertion, O(n^2) complexity. See http://www.ocert.org/advisories/ocert-2011-003.html for details.

Changing hash values affects the iteration order of sets. Python has never made guarantees about this ordering (and it typically varies between 32-bit and 64-bit builds).

See also PYTHONHASHSEED.

Distinto en la versión 3.3: Hash randomization is enabled by default.

object.**__bool__**(*self*)
Called to implement truth value testing and the built-in operation bool(); should return False or True. When this method is not defined, *__len__()* is called, if it is defined, and the object is considered true if its result is nonzero. If a class defines neither *__len__()* nor *__bool__()*, all its instances are considered true.

3.3.2 Customizing attribute access

The following methods can be defined to customize the meaning of attribute access (use of, assignment to, or deletion of x.name) for class instances.

object.**__getattr__**(*self*, *name*)
Called when the default attribute access fails with an AttributeError (either *__getattribute__()* raises an AttributeError because *name* is not an instance attribute or an attribute in the class tree for self; or *__get__()* of a *name* property raises AttributeError). This method should either return the (computed) attribute value or raise an AttributeError exception.

Note that if the attribute is found through the normal mechanism, *__getattr__()* is not called. (This is an intentional asymmetry between *__getattr__()* and *__setattr__()*.) This is done both for efficiency reasons and because otherwise *__getattr__()* would have no way to access other attributes of the instance. Note that at least for instance variables, you can fake total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object). See the *__getattribute__()* method below for a way to actually get total control over attribute access.

object.**__getattribute__**(*self*, *name*)
Called unconditionally to implement attribute accesses for instances of the class. If the class also defines *__getattr__()*, the latter will not be called unless *__getattribute__()* either calls it explicitly or raises an AttributeError. This method should return the (computed) attribute value or raise an AttributeError exception. In order to avoid infinite recursion in this method, its implementation should always call the base class method with the same name to access any attributes it needs, for example, object.__getattribute__(self, name).

Nota: This method may still be bypassed when looking up special methods as the result of implicit invocation via language syntax or built-in functions. See *Special method lookup*.

For certain sensitive attribute accesses, raises an auditing event object.__getattr__ with arguments obj and name.

object.**__setattr__**(*self*, *name*, *value*)
Called when an attribute assignment is attempted. This is called instead of the normal mechanism (i.e. store the value in the instance dictionary). *name* is the attribute name, *value* is the value to be assigned to it.

If *__setattr__()* wants to assign to an instance attribute, it should call the base class method with the same name, for example, object.__setattr__(self, name, value).

For certain sensitive attribute assignments, raises an auditing event object.__setattr__ with arguments obj, name, value.

object.**__delattr__**(*self*, *name*)
Like *__setattr__()* but for attribute deletion instead of assignment. This should only be implemented if del obj.name is meaningful for the object.

For certain sensitive attribute deletions, raises an auditing event `object.__delattr__` with arguments `obj` and `name`.

`object.`**`__dir__`**`(`*`self`*`)`
Called when `dir()` is called on the object. A sequence must be returned. `dir()` converts the returned sequence to a list and sorts it.

Customizing module attribute access

Special names `__getattr__` and `__dir__` can be also used to customize access to module attributes. The `__getattr__` function at the module level should accept one argument which is the name of an attribute and return the computed value or raise an `AttributeError`. If an attribute is not found on a module object through the normal lookup, i.e. *`object.__getattribute__()`*, then `__getattr__` is searched in the module `__dict__` before raising an `AttributeError`. If found, it is called with the attribute name and the result is returned.

The `__dir__` function should accept no arguments, and return a sequence of strings that represents the names accessible on module. If present, this function overrides the standard `dir()` search on a module.

For a more fine grained customization of the module behavior (setting attributes, properties, etc.), one can set the `__class__` attribute of a module object to a subclass of `types.ModuleType`. For example:

```
import sys
from types import ModuleType

class VerboseModule(ModuleType):
    def __repr__(self):
        return f'Verbose {self.__name__}'

    def __setattr__(self, attr, value):
        print(f'Setting {attr}...')
        super().__setattr__(attr, value)

sys.modules[__name__].__class__ = VerboseModule
```

Nota: Defining module `__getattr__` and setting module `__class__` only affect lookups made using the attribute access syntax – directly accessing the module globals (whether by code within the module, or via a reference to the module's globals dictionary) is unaffected.

Distinto en la versión 3.5: `__class__` module attribute is now writable.

Nuevo en la versión 3.7: `__getattr__` and `__dir__` module attributes.

Ver también:

PEP 562 - Module __getattr__ and __dir__ Describes the `__getattr__` and `__dir__` functions on modules.

Implementing Descriptors

The following methods only apply when an instance of the class containing the method (a so-called *descriptor* class) appears in an *owner* class (the descriptor must be in either the owner's class dictionary or in the class dictionary for one of its parents). In the examples below, «the attribute» refers to the attribute whose name is the key of the property in the owner class" `__dict__`.

`object.`**`__get__`**(*self*, *instance*, *owner=None*)
Called to get the attribute of the owner class (class attribute access) or of an instance of that class (instance attribute access). The optional *owner* argument is the owner class, while *instance* is the instance that the attribute was accessed through, or `None` when the attribute is accessed through the *owner*.

This method should return the computed attribute value or raise an `AttributeError` exception.

PEP 252 specifies that *`__get__()`* is callable with one or two arguments. Python's own built-in descriptors support this specification; however, it is likely that some third-party tools have descriptors that require both arguments. Python's own *`__getattribute__()`* implementation always passes in both arguments whether they are required or not.

`object.`**`__set__`**(*self*, *instance*, *value*)
Called to set the attribute on an instance *instance* of the owner class to a new value, *value*.

Note, adding *`__set__()`* or *`__delete__()`* changes the kind of descriptor to a «data descriptor». See *Invoking Descriptors* for more details.

`object.`**`__delete__`**(*self*, *instance*)
Called to delete the attribute on an instance *instance* of the owner class.

`object.`**`__set_name__`**(*self*, *owner*, *name*)
Called at the time the owning class *owner* is created. The descriptor has been assigned to *name*.

Nota: *`__set_name__()`* is only called implicitly as part of the `type` constructor, so it will need to be called explicitly with the appropriate parameters when a descriptor is added to a class after initial creation:

```
class A:
   pass
descr = custom_descriptor()
A.attr = descr
descr.__set_name__(A, 'attr')
```

See *Creating the class object* for more details.

Nuevo en la versión 3.6.

The attribute `__objclass__` is interpreted by the `inspect` module as specifying the class where this object was defined (setting this appropriately can assist in runtime introspection of dynamic class attributes). For callables, it may indicate that an instance of the given type (or a subclass) is expected or required as the first positional argument (for example, CPython sets this attribute for unbound methods that are implemented in C).

Invoking Descriptors

In general, a descriptor is an object attribute with «binding behavior», one whose attribute access has been overridden by methods in the descriptor protocol: *__get__()*, *__set__()*, and *__delete__()*. If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, `a.x` has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the base classes of `type(a)` excluding metaclasses.

However, if the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined and how they were called.

The starting point for descriptor invocation is a binding, `a.x`. How the arguments are assembled depends on `a`:

Direct Call The simplest and least common call is when user code directly invokes a descriptor method: `x.__get__(a)`.

Instance Binding If binding to an object instance, `a.x` is transformed into the call: `type(a).__dict__['x'].__get__(a, type(a))`.

Class Binding If binding to a class, `A.x` is transformed into the call: `A.__dict__['x'].__get__(None, A)`.

Super Binding If `a` is an instance of `super`, then the binding `super(B, obj).m()` searches `obj.__class__.__mro__` for the base class `A` immediately preceding `B` and then invokes the descriptor with the call: `A.__dict__['m'].__get__(obj, obj.__class__)`.

For instance bindings, the precedence of descriptor invocation depends on which descriptor methods are defined. A descriptor can define any combination of *__get__()*, *__set__()* and *__delete__()*. If it does not define *__get__()*, then accessing the attribute will return the descriptor object itself unless there is a value in the object's instance dictionary. If the descriptor defines *__set__()* and/or *__delete__()*, it is a data descriptor; if it defines neither, it is a non-data descriptor. Normally, data descriptors define both *__get__()* and *__set__()*, while non-data descriptors have just the *__get__()* method. Data descriptors with *__set__()* and *__get__()* defined always override a redefinition in an instance dictionary. In contrast, non-data descriptors can be overridden by instances.

Python methods (including `staticmethod()` and `classmethod()`) are implemented as non-data descriptors. Accordingly, instances can redefine and override methods. This allows individual instances to acquire behaviors that differ from other instances of the same class.

The `property()` function is implemented as a data descriptor. Accordingly, instances cannot override the behavior of a property.

__slots__

__slots__ allow us to explicitly declare data members (like properties) and deny the creation of *__dict__* and *__weakref__* (unless explicitly declared in *__slots__* or available in a parent.)

The space saved over using *__dict__* can be significant. Attribute lookup speed can be significantly improved as well.

`object.`**`__slots__`**

This class variable can be assigned a string, iterable, or sequence of strings with variable names used by instances. *__slots__* reserves space for the declared variables and prevents the automatic creation of *__dict__* and *__weakref__* for each instance.

Notes on using *__slots__*

- When inheriting from a class without *__slots__*, the *__dict__* and *__weakref__* attribute of the instances will always be accessible.
- Without a *__dict__* variable, instances cannot be assigned new variables not listed in the *__slots__* definition. Attempts to assign to an unlisted variable name raises `AttributeError`. If dynamic assignment of new variables is desired, then add `'__dict__'` to the sequence of strings in the *__slots__* declaration.
- Without a *__weakref__* variable for each instance, classes defining *__slots__* do not support weak references to its instances. If weak reference support is needed, then add `'__weakref__'` to the sequence of strings in the *__slots__* declaration.
- *__slots__* are implemented at the class level by creating descriptors (*Implementing Descriptors*) for each variable name. As a result, class attributes cannot be used to set default values for instance variables defined by *__slots__*; otherwise, the class attribute would overwrite the descriptor assignment.
- The action of a *__slots__* declaration is not limited to the class where it is defined. *__slots__* declared in parents are available in child classes. However, child subclasses will get a *__dict__* and *__weakref__* unless they also define *__slots__* (which should only contain names of any *additional* slots).
- If a class defines a slot also defined in a base class, the instance variable defined by the base class slot is inaccessible (except by retrieving its descriptor directly from the base class). This renders the meaning of the program undefined. In the future, a check may be added to prevent this.
- Nonempty *__slots__* does not work for classes derived from «variable-length» built-in types such as `int`, `bytes` and `tuple`.
- Any non-string iterable may be assigned to *__slots__*. Mappings may also be used; however, in the future, special meaning may be assigned to the values corresponding to each key.
- *__class__* assignment works only if both classes have the same *__slots__*.
- Multiple inheritance with multiple slotted parent classes can be used, but only one parent is allowed to have attributes created by slots (the other bases must have empty slot layouts) - violations raise `TypeError`.
- If an iterator is used for *__slots__* then a descriptor is created for each of the iterator's values. However, the *__slots__* attribute will be an empty iterator.

3.3.3 Customizing class creation

Whenever a class inherits from another class, *__init_subclass__* is called on that class. This way, it is possible to write classes which change the behavior of subclasses. This is closely related to class decorators, but where class decorators only affect the specific class they're applied to, `__init_subclass__` solely applies to future subclasses of the class defining the method.

classmethod `object.`**`__init_subclass__`**`(`*cls*`)`

This method is called whenever the containing class is subclassed. *cls* is then the new subclass. If defined as a normal instance method, this method is implicitly converted to a class method.

Keyword arguments which are given to a new class are passed to the parent's class `__init_subclass__`. For compatibility with other classes using `__init_subclass__`, one should take out the needed keyword arguments and pass the others over to the base class, as in:

```
class Philosopher:
    def __init_subclass__(cls, /, default_name, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.default_name = default_name
```

(continué en la próxima página)

(proviene de la página anterior)

```
class AustralianPhilosopher(Philosopher, default_name="Bruce"):
    pass
```

The default implementation `object.__init_subclass__` does nothing, but raises an error if it is called with any arguments.

Nota: The metaclass hint `metaclass` is consumed by the rest of the type machinery, and is never passed to `__init_subclass__` implementations. The actual metaclass (rather than the explicit hint) can be accessed as `type(cls)`.

Nuevo en la versión 3.6.

Metaclasses

By default, classes are constructed using `type()`. The class body is executed in a new namespace and the class name is bound locally to the result of `type(name, bases, namespace)`.

The class creation process can be customized by passing the `metaclass` keyword argument in the class definition line, or by inheriting from an existing class that included such an argument. In the following example, both `MyClass` and `MySubclass` are instances of `Meta`:

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

Any other keyword arguments that are specified in the class definition are passed through to all metaclass operations described below.

When a class definition is executed, the following steps occur:

- MRO entries are resolved;
- the appropriate metaclass is determined;
- the class namespace is prepared;
- the class body is executed;
- the class object is created.

Resolving MRO entries

If a base that appears in class definition is not an instance of `type`, then an `__mro_entries__` method is searched on it. If found, it is called with the original bases tuple. This method must return a tuple of classes that will be used instead of this base. The tuple may be empty, in such case the original base is ignored.

Ver también:

PEP 560 - Core support for typing module and generic types

Determining the appropriate metaclass

The appropriate metaclass for a class definition is determined as follows:

- if no bases and no explicit metaclass are given, then `type()` is used;
- if an explicit metaclass is given and it is *not* an instance of `type()`, then it is used directly as the metaclass;
- if an instance of `type()` is given as the explicit metaclass, or bases are defined, then the most derived metaclass is used.

The most derived metaclass is selected from the explicitly specified metaclass (if any) and the metaclasses (i.e. `type(cls)`) of all specified base classes. The most derived metaclass is one which is a subtype of *all* of these candidate metaclasses. If none of the candidate metaclasses meets that criterion, then the class definition will fail with `TypeError`.

Preparing the class namespace

Once the appropriate metaclass has been identified, then the class namespace is prepared. If the metaclass has a `__prepare__` attribute, it is called as `namespace = metaclass.__prepare__(name, bases, **kwds)` (where the additional keyword arguments, if any, come from the class definition). The `__prepare__` method should be implemented as a `classmethod()`. The namespace returned by `__prepare__` is passed in to `__new__`, but when the final class object is created the namespace is copied into a new `dict`.

If the metaclass has no `__prepare__` attribute, then the class namespace is initialised as an empty ordered mapping.

Ver también:

PEP 3115 - Metaclasses in Python 3000 Introduced the `__prepare__` namespace hook

Executing the class body

The class body is executed (approximately) as `exec(body, globals(), namespace)`. The key difference from a normal call to `exec()` is that lexical scoping allows the class body (including any methods) to reference names from the current and outer scopes when the class definition occurs inside a function.

However, even when the class definition occurs inside the function, methods defined inside the class still cannot see names defined at the class scope. Class variables must be accessed through the first parameter of instance or class methods, or through the implicit lexically scoped `__class__` reference described in the next section.

Creating the class object

Once the class namespace has been populated by executing the class body, the class object is created by calling `metaclass(name, bases, namespace, **kwds)` (the additional keywords passed here are the same as those passed to `__prepare__`).

This class object is the one that will be referenced by the zero-argument form of `super()`. `__class__` is an implicit closure reference created by the compiler if any methods in a class body refer to either `__class__` or `super`. This allows the zero argument form of `super()` to correctly identify the class being defined based on lexical scoping, while the class or instance that was used to make the current call is identified based on the first argument passed to the method.

CPython implementation detail: In CPython 3.6 and later, the `__class__` cell is passed to the metaclass as a `__classcell__` entry in the class namespace. If present, this must be propagated up to the `type.__new__` call in order for the class to be initialised correctly. Failing to do so will result in a `RuntimeError` in Python 3.8.

When using the default metaclass `type`, or any metaclass that ultimately calls `type.__new__`, the following additional customisation steps are invoked after creating the class object:

- first, `type.__new__` collects all of the descriptors in the class namespace that define a *`__set_name__()`* method;
- second, all of these `__set_name__` methods are called with the class being defined and the assigned name of that particular descriptor;
- finally, the *`__init_subclass__()`* hook is called on the immediate parent of the new class in its method resolution order.

After the class object is created, it is passed to the class decorators included in the class definition (if any) and the resulting object is bound in the local namespace as the defined class.

When a new class is created by `type.__new__`, the object provided as the namespace parameter is copied to a new ordered mapping and the original object is discarded. The new copy is wrapped in a read-only proxy, which becomes the `__dict__` attribute of the class object.

Ver también:

PEP 3135 - New super Describes the implicit `__class__` closure reference

Uses for metaclasses

The potential uses for metaclasses are boundless. Some ideas that have been explored include enum, logging, interface checking, automatic delegation, automatic property creation, proxies, frameworks, and automatic resource locking/synchronization.

3.3.4 Customizing instance and subclass checks

The following methods are used to override the default behavior of the `isinstance()` and `issubclass()` built-in functions.

In particular, the metaclass `abc.ABCMeta` implements these methods in order to allow the addition of Abstract Base Classes (ABCs) as «virtual base classes» to any class or type (including built-in types), including other ABCs.

`class.__instancecheck__`(*self*, *instance*)
: Return true if *instance* should be considered a (direct or indirect) instance of *class*. If defined, called to implement `isinstance(instance, class)`.

`class.__subclasscheck__`(*self*, *subclass*)
: Return true if *subclass* should be considered a (direct or indirect) subclass of *class*. If defined, called to implement `issubclass(subclass, class)`.

Note that these methods are looked up on the type (metaclass) of a class. They cannot be defined as class methods in the actual class. This is consistent with the lookup of special methods that are called on instances, only in this case the instance is itself a class.

Ver también:

PEP 3119 - Introducing Abstract Base Classes Includes the specification for customizing `isinstance()` and `issubclass()` behavior through *`__instancecheck__()`* and *`__subclasscheck__()`*, with motivation for this functionality in the context of adding Abstract Base Classes (see the `abc` module) to the language.

3.3.5 Emulating generic types

One can implement the generic class syntax as specified by **PEP 484** (for example `List[int]`) by defining a special method:

classmethod object.**__class_getitem__**(*cls*, *key*)
Return an object representing the specialization of a generic class by type arguments found in *key*.

This method is looked up on the class object itself, and when defined in the class body, this method is implicitly a class method. Note, this mechanism is primarily reserved for use with static type hints, other usage is discouraged.

Ver también:

PEP 560 - Core support for typing module and generic types

3.3.6 Emulating callable objects

object.**__call__**(*self*[, *args...*])
Called when the instance is «called» as a function; if this method is defined, `x(arg1, arg2, ...)` roughly translates to `type(x).__call__(x, arg1, ...)`.

3.3.7 Emulating container types

The following methods can be defined to implement container objects. Containers usually are sequences (such as lists or tuples) or mappings (like dictionaries), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping; the difference is that for a sequence, the allowable keys should be the integers *k* for which `0 <= k < N` where *N* is the length of the sequence, or slice objects, which define a range of items. It is also recommended that mappings provide the methods `keys()`, `values()`, `items()`, `get()`, `clear()`, `setdefault()`, `pop()`, `popitem()`, `copy()`, and `update()` behaving similar to those for Python's standard dictionary objects. The `collections.abc` module provides a `MutableMapping` abstract base class to help create those methods from a base set of *`__getitem__()`*, *`__setitem__()`*, *`__delitem__()`*, and `keys()`. Mutable sequences should provide methods `append()`, `count()`, `index()`, `extend()`, `insert()`, `pop()`, `remove()`, `reverse()` and `sort()`, like Python standard list objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods *`__add__()`*, *`__radd__()`*, *`__iadd__()`*, *`__mul__()`*, *`__rmul__()`* and *`__imul__()`* described below; they should not define other numerical operators. It is recommended that both mappings and sequences implement the *`__contains__()`* method to allow efficient use of the `in` operator; for mappings, `in` should search the mapping's keys; for sequences, it should search through the values. It is further recommended that both mappings and sequences implement the *`__iter__()`* method to allow efficient iteration through the container; for mappings, *`__iter__()`* should iterate through the object's keys; for sequences, it should iterate through the values.

object.**__len__**(*self*)
Called to implement the built-in function `len()`. Should return the length of the object, an integer >= 0. Also, an

object that doesn't define a *__bool__()* method and whose *__len__()* method returns zero is considered to be false in a Boolean context.

CPython implementation detail: In CPython, the length is required to be at most `sys.maxsize`. If the length is larger than `sys.maxsize` some features (such as `len()`) may raise `OverflowError`. To prevent raising `OverflowError` by truth value testing, an object must define a *__bool__()* method.

`object.`**`__length_hint__`**(*self*)
Called to implement `operator.length_hint()`. Should return an estimated length for the object (which may be greater or less than the actual length). The length must be an integer >= 0. The return value may also be `NotImplemented`, which is treated the same as if the `__length_hint__` method didn't exist at all. This method is purely an optimization and is never required for correctness.

Nuevo en la versión 3.4.

Nota: Slicing is done exclusively with the following three methods. A call like

```
a[1:2] = b
```

is translated to

```
a[slice(1, 2, None)] = b
```

and so forth. Missing slice items are always filled in with `None`.

`object.`**`__getitem__`**(*self*, *key*)
Called to implement evaluation of `self[key]`. For sequence types, the accepted keys should be integers and slice objects. Note that the special interpretation of negative indexes (if the class wishes to emulate a sequence type) is up to the *__getitem__()* method. If *key* is of an inappropriate type, `TypeError` may be raised; if of a value outside the set of indexes for the sequence (after any special interpretation of negative values), `IndexError` should be raised. For mapping types, if *key* is missing (not in the container), `KeyError` should be raised.

Nota: *for* loops expect that an `IndexError` will be raised for illegal indexes to allow proper detection of the end of the sequence.

`object.`**`__setitem__`**(*self*, *key*, *value*)
Called to implement assignment to `self[key]`. Same note as for *__getitem__()*. This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper *key* values as for the *__getitem__()* method.

`object.`**`__delitem__`**(*self*, *key*)
Called to implement deletion of `self[key]`. Same note as for *__getitem__()*. This should only be implemented for mappings if the objects support removal of keys, or for sequences if elements can be removed from the sequence. The same exceptions should be raised for improper *key* values as for the *__getitem__()* method.

`object.`**`__missing__`**(*self*, *key*)
Called by `dict.`*__getitem__()* to implement `self[key]` for dict subclasses when key is not in the dictionary.

`object.`**`__iter__`**(*self*)
This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container.

Iterator objects also need to implement this method; they are required to return themselves. For more information on iterator objects, see typeiter.

object.**__reversed__**(*self*)
Called (if present) by the `reversed()` built-in to implement reverse iteration. It should return a new iterator object that iterates over all the objects in the container in reverse order.

If the *__reversed__()* method is not provided, the `reversed()` built-in will fall back to using the sequence protocol (*__len__()* and *__getitem__()*). Objects that support the sequence protocol should only provide *__reversed__()* if they can provide an implementation that is more efficient than the one provided by `reversed()`.

The membership test operators (*in* and *not in*) are normally implemented as an iteration through a container. However, container objects can supply the following special method with a more efficient implementation, which also does not require the object be iterable.

object.**__contains__**(*self*, *item*)
Called to implement membership test operators. Should return true if *item* is in *self*, false otherwise. For mapping objects, this should consider the keys of the mapping rather than the values or the key-item pairs.

For objects that don't define *__contains__()*, the membership test first tries iteration via *__iter__()*, then the old sequence iteration protocol via *__getitem__()*, see *this section in the language reference*.

3.3.8 Emulating numeric types

The following methods can be defined to emulate numeric objects. Methods corresponding to operations that are not supported by the particular kind of number implemented (e.g., bitwise operations for non-integral numbers) should be left undefined.

object.**__add__**(*self*, *other*)
object.**__sub__**(*self*, *other*)
object.**__mul__**(*self*, *other*)
object.**__matmul__**(*self*, *other*)
object.**__truediv__**(*self*, *other*)
object.**__floordiv__**(*self*, *other*)
object.**__mod__**(*self*, *other*)
object.**__divmod__**(*self*, *other*)
object.**__pow__**(*self*, *other*[, *modulo*])
object.**__lshift__**(*self*, *other*)
object.**__rshift__**(*self*, *other*)
object.**__and__**(*self*, *other*)
object.**__xor__**(*self*, *other*)
object.**__or__**(*self*, *other*)
These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). For instance, to evaluate the expression `x + y`, where *x* is an instance of a class that has an *__add__()* method, `x.__add__(y)` is called. The *__divmod__()* method should be the equivalent to using *__floordiv__()* and *__mod__()*; it should not be related to *__truediv__()*. Note that *__pow__()* should be defined to accept an optional third argument if the ternary version of the built-in `pow()` function is to be supported.

If one of those methods does not support the operation with the supplied arguments, it should return `NotImplemented`.

object.**__radd__**(*self*, *other*)
object.**__rsub__**(*self*, *other*)
object.**__rmul__**(*self*, *other*)
object.**__rmatmul__**(*self*, *other*)
object.**__rtruediv__**(*self*, *other*)
object.**__rfloordiv__**(*self*, *other*)

object.**__rmod__**(*self*, *other*)
object.**__rdivmod__**(*self*, *other*)
object.**__rpow__**(*self*, *other*[, *modulo*])
object.**__rlshift__**(*self*, *other*)
object.**__rrshift__**(*self*, *other*)
object.**__rand__**(*self*, *other*)
object.**__rxor__**(*self*, *other*)
object.**__ror__**(*self*, *other*)

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation[3] and the operands are of different types.[4] For instance, to evaluate the expression `x - y`, where *y* is an instance of a class that has an *__rsub__()* method, `y.__rsub__(x)` is called if `x.__sub__(y)` returns *NotImplemented*.

Note that ternary `pow()` will not try calling *__rpow__()* (the coercion rules would become too complicated).

Nota: If the right operand's type is a subclass of the left operand's type and that subclass provides a different implementation of the reflected method for the operation, this method will be called before the left operand's non-reflected method. This behavior allows subclasses to override their ancestors" operations.

---

object.**__iadd__**(*self*, *other*)
object.**__isub__**(*self*, *other*)
object.**__imul__**(*self*, *other*)
object.**__imatmul__**(*self*, *other*)
object.**__itruediv__**(*self*, *other*)
object.**__ifloordiv__**(*self*, *other*)
object.**__imod__**(*self*, *other*)
object.**__ipow__**(*self*, *other*[, *modulo*])
object.**__ilshift__**(*self*, *other*)
object.**__irshift__**(*self*, *other*)
object.**__iand__**(*self*, *other*)
object.**__ixor__**(*self*, *other*)
object.**__ior__**(*self*, *other*)

These methods are called to implement the augmented arithmetic assignments (`+=`, `-=`, `*=`, `@=`, `/=`, `//=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`). These methods should attempt to do the operation in-place (modifying *self*) and return the result (which could be, but does not have to be, *self*). If a specific method is not defined, the augmented assignment falls back to the normal methods. For instance, if *x* is an instance of a class with an *__iadd__()* method, `x += y` is equivalent to `x = x.__iadd__(y)` . Otherwise, `x.__add__(y)` and `y.__radd__(x)` are considered, as with the evaluation of `x + y`. In certain situations, augmented assignment can result in unexpected errors (see faq-augmented-assignment-tuple-error), but this behavior is in fact part of the data model.

Nota: Due to a bug in the dispatching mechanism for `**=`, a class that defines *__ipow__()* but returns `NotImplemented` would fail to fall back to `x.__pow__(y)` and `y.__rpow__(x)`. This bug is fixed in Python 3.10.

---

object.**__neg__**(*self*)
object.**__pos__**(*self*)
object.**__abs__**(*self*)

[3] «Does not support» here means that the class has no such method, or the method returns `NotImplemented`. Do not set the method to `None` if you want to force fallback to the right operand's reflected method—that will instead have the opposite effect of explicitly *blocking* such fallback.

[4] For operands of the same type, it is assumed that if the non-reflected method – such as *__add__()* – fails then the overall operation is not supported, which is why the reflected method is not called.

object.**__invert__**(*self*)
Called to implement the unary arithmetic operations (-, +, abs() and ~).

object.**__complex__**(*self*)
object.**__int__**(*self*)
object.**__float__**(*self*)
Called to implement the built-in functions complex(), int() and float(). Should return a value of the appropriate type.

object.**__index__**(*self*)
Called to implement operator.index(), and whenever Python needs to losslessly convert the numeric object to an integer object (such as in slicing, or in the built-in bin(), hex() and oct() functions). Presence of this method indicates that the numeric object is an integer type. Must return an integer.

If *__int__()*, *__float__()* and *__complex__()* are not defined then corresponding built-in functions int(), float() and complex() fall back to *__index__()*.

object.**__round__**(*self*[, *ndigits*])
object.**__trunc__**(*self*)
object.**__floor__**(*self*)
object.**__ceil__**(*self*)
Called to implement the built-in function round() and math functions trunc(), floor() and ceil(). Unless *ndigits* is passed to __round__() all these methods should return the value of the object truncated to an Integral (typically an int).

The built-in function int() falls back to *__trunc__()* if neither *__int__()* nor *__index__()* is defined.

3.3.9 With Statement Context Managers

A *context manager* is an object that defines the runtime context to be established when executing a *with* statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the with statement (described in section *La sentencia with*), but can also be used by directly invoking their methods.

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.

For more information on context managers, see typecontextmanager.

object.**__enter__**(*self*)
Enter the runtime context related to this object. The *with* statement will bind this method's return value to the target(s) specified in the as clause of the statement, if any.

object.**__exit__**(*self*, *exc_type*, *exc_value*, *traceback*)
Exit the runtime context related to this object. The parameters describe the exception that caused the context to be exited. If the context was exited without an exception, all three arguments will be None.

If an exception is supplied, and the method wishes to suppress the exception (i.e., prevent it from being propagated), it should return a true value. Otherwise, the exception will be processed normally upon exit from this method.

Note that *__exit__()* methods should not reraise the passed-in exception; this is the caller's responsibility.

Ver también:

PEP 343 - The «with» statement The specification, background, and examples for the Python *with* statement.

3.3.10 Special method lookup

For custom classes, implicit invocations of special methods are only guaranteed to work correctly if defined on an object's type, not in the object's instance dictionary. That behaviour is the reason why the following code raises an exception:

```
>>> class C:
...     pass
...
>>> c = C()
>>> c.__len__ = lambda: 5
>>> len(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'C' has no len()
```

The rationale behind this behaviour lies with a number of special methods such as *__hash__()* and *__repr__()* that are implemented by all objects, including type objects. If the implicit lookup of these methods used the conventional lookup process, they would fail when invoked on the type object itself:

```
>>> 1 .__hash__() == hash(1)
True
>>> int.__hash__() == hash(int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor '__hash__' of 'int' object needs an argument
```

Incorrectly attempting to invoke an unbound method of a class in this way is sometimes referred to as "metaclass confusion", and is avoided by bypassing the instance when looking up special methods:

```
>>> type(1).__hash__(1) == hash(1)
True
>>> type(int).__hash__(int) == hash(int)
True
```

In addition to bypassing any instance attributes in the interest of correctness, implicit special method lookup generally also bypasses the *__getattribute__()* method even of the object's metaclass:

```
>>> class Meta(type):
...     def __getattribute__(*args):
...         print("Metaclass getattribute invoked")
...         return type.__getattribute__(*args)
...
>>> class C(object, metaclass=Meta):
...     def __len__(self):
...         return 10
...     def __getattribute__(*args):
...         print("Class getattribute invoked")
...         return object.__getattribute__(*args)
...
>>> c = C()
>>> c.__len__()                 # Explicit lookup via instance
Class getattribute invoked
10
>>> type(c).__len__(c)          # Explicit lookup via type
Metaclass getattribute invoked
10
>>> len(c)                      # Implicit lookup
10
```

Bypassing the *__getattribute__()* machinery in this fashion provides significant scope for speed optimisations within the interpreter, at the cost of some flexibility in the handling of special methods (the special method *must* be set on the class object itself in order to be consistently invoked by the interpreter).

3.4 Coroutines

3.4.1 Awaitable Objects

An *awaitable* object generally implements an *__await__()* method. *Coroutine objects* returned from *async def* functions are awaitable.

Nota: The *generator iterator* objects returned from generators decorated with `types.coroutine()` or `asyncio.coroutine()` are also awaitable, but they do not implement *__await__()*.

`object.`**`__await__`**`(`*self*`)`
Must return an *iterator*. Should be used to implement *awaitable* objects. For instance, `asyncio.Future` implements this method to be compatible with the *await* expression.

Nuevo en la versión 3.5.

Ver también:

PEP 492 for additional information about awaitable objects.

3.4.2 Coroutine Objects

Coroutine objects are *awaitable* objects. A coroutine's execution can be controlled by calling *__await__()* and iterating over the result. When the coroutine has finished executing and returns, the iterator raises `StopIteration`, and the exception's `value` attribute holds the return value. If the coroutine raises an exception, it is propagated by the iterator. Coroutines should not directly raise unhandled `StopIteration` exceptions.

Coroutines also have the methods listed below, which are analogous to those of generators (see *Métodos generador-iterador*). However, unlike generators, coroutines do not directly support iteration.

Distinto en la versión 3.5.2: It is a `RuntimeError` to await on a coroutine more than once.

`coroutine.`**`send`**`(`*value*`)`
Starts or resumes execution of the coroutine. If *value* is `None`, this is equivalent to advancing the iterator returned by *__await__()*. If *value* is not `None`, this method delegates to the *send()* method of the iterator that caused the coroutine to suspend. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the *__await__()* return value, described above.

`coroutine.`**`throw`**`(`*type*`[`, *value*`[`, *traceback*`]]`)
Raises the specified exception in the coroutine. This method delegates to the *throw()* method of the iterator that caused the coroutine to suspend, if it has such a method. Otherwise, the exception is raised at the suspension point. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the *__await__()* return value, described above. If the exception is not caught in the coroutine, it propagates back to the caller.

`coroutine.`**`close`**`()`
Causes the coroutine to clean itself up and exit. If the coroutine is suspended, this method first delegates to the *close()* method of the iterator that caused the coroutine to suspend, if it has such a method. Then it raises `GeneratorExit` at the suspension point, causing the coroutine to immediately clean itself up. Finally, the coroutine is marked as having finished executing, even if it was never started.

Coroutine objects are automatically closed using the above process when they are about to be destroyed.

3.4.3 Asynchronous Iterators

An *asynchronous iterator* can call asynchronous code in its `__anext__` method.

Asynchronous iterators can be used in an `async for` statement.

`object.__aiter__`(*self*)
Must return an *asynchronous iterator* object.

`object.__anext__`(*self*)
Must return an *awaitable* resulting in a next value of the iterator. Should raise a `StopAsyncIteration` error when the iteration is over.

An example of an asynchronous iterable object:

```
class Reader:
    async def readline(self):
        ...

    def __aiter__(self):
        return self

    async def __anext__(self):
        val = await self.readline()
        if val == b'':
            raise StopAsyncIteration
        return val
```

Nuevo en la versión 3.5.

Distinto en la versión 3.7: Prior to Python 3.7, `__aiter__` could return an *awaitable* that would resolve to an *asynchronous iterator*.

Starting with Python 3.7, `__aiter__` must return an asynchronous iterator object. Returning anything else will result in a `TypeError` error.

3.4.4 Asynchronous Context Managers

An *asynchronous context manager* is a *context manager* that is able to suspend execution in its `__aenter__` and `__aexit__` methods.

Asynchronous context managers can be used in an `async with` statement.

`object.__aenter__`(*self*)
Semantically similar to `__enter__()`, the only difference being that it must return an *awaitable*.

`object.__aexit__`(*self*, *exc_type*, *exc_value*, *traceback*)
Semantically similar to `__exit__()`, the only difference being that it must return an *awaitable*.

An example of an asynchronous context manager class:

```
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```

Nuevo en la versión 3.5.

CAPÍTULO 4

Modelo de ejecución

4.1 Estructura de un programa

Un programa Python está construido a partir de bloques de código. Un *block* es un trozo de texto de un programa Python que se ejecuta como una unidad. Los siguientes son bloques: un módulo, el cuerpo de una función y la definición de una clase. Cada comando ingresado en el intérprete interactivo es un bloque. Un archivo de script (un archivo provisto como entrada estándar al intérprete, o especificado como argumento en la línea de comando al intérprete) es un bloque de código. Un comando de script (un comando especificado en la línea de comandos del intérprete con la opción `-c` es un bloque de código. El argumento cadena de caracteres que se envía a las funciones incorporadas `eval()` y `exec()` es también un bloque de código.

Un bloque de código se ejecuta en un *execution frame*. Un marco contiene alguna información administrativa (que se usa para depuración) y determina dónde y cómo continuará la ejecución una vez que el bloque de código se haya completado.

4.2 Nombres y vínculos

4.2.1 Vinculación de nombres

Los *Names* refieren a objetos. Los nombres se introducen por las operaciones de vinculación de nombre (*name binding operations*).

The following constructs bind names: formal parameters to functions, *import* statements, class and function definitions (these bind the class or function name in the defining block), and targets that are identifiers if occurring in an assignment, *for* loop header, or after `as` in a *with* statement or *except* clause. The `import` statement of the form `from ... import *` binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

A target occurring in a *del* statement is also considered bound for this purpose (though the actual semantics are to unbind the name).

Cada declaración de asignación o importación ocurre dentro de un bloque determinado por una definición de clase o de función, o a nivel de módulo (el bloque de código de máximo nivel).

Si un nombre está vinculado en un bloque, es una variable local en ese bloque, salvo que sea declarado como *nonlocal* o *global*. Si un nombre está vinculado a nivel de módulo, es una variable global. (Las variables del bloque de código del módulo son locales y globales.) Si una variable se una en un bloque de código pero no está definida ahí, es una *free variable*.

Cada ocurrencia de un nombre en el texto del programa se refiere al *binding* de ese nombre, establecido por las siguientes reglas de resolución de nombres.

4.2.2 Resolución de nombres

Un *scope* define la visibilidad de un nombre en un bloque. Si una variable local se define en un bloque, su ámbito (*scope*) incluye ese bloque. Si la definición ocurre en un bloque de función, el ámbito se extiende a cualquier bloque contenido en el bloque en donde está la definición, a menos que uno de los bloques contenidos introduzca un vínculo diferente para el nombre.

When a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's *environment*.

Cuando un nombre no se encuentra, se lanza una excepción `NameError`. Si el ámbito actual es una función, y el nombre se refiere a una variable local que todavía no ha sido vinculada a un valor en el punto en el que nombre es utilizado, se lanza una excepción `UnboundLocalError`. `UnboundLocalError` es una subclase de `NameError`.

Si una operación de vinculación de nombre ocurre en cualquier parte dentro de un bloque de código, todos los usos del nombre dentro de ese bloque son tratados como referencias al bloque actual. Esto puede llevar a errores cuando el nombre es utilizado dentro del bloque antes de su vinculación. Esta regla es sutil. Python carece de declaraciones y permite que las operaciones de vinculación de nombres ocurran en cualquier lugar dentro del bloque de código. Las variables locales de un bloque de código pueden determinarse buscando operaciones de vinculación de nombres en el texto completo del bloque.

Si la declaración *global* ocurre dentro de un bloque, todos los usos del nombre especificado en la declaración se refieren a la vinculación que ese nombre tiene en el espacio de nombres (*namespace*) de nivel superior. Los nombres se resuelven en el espacio de nombres de nivel superior buscando en el espacio de nombres global, es decir, el espacio de nombres del módulo que contiene el bloque de código, y en el espacio de nombres incorporado, el *namespace* del módulo `builtins`. La búsqueda se realiza primero en el espacio de nombres global. Si el nombre no se encuentra ahí, se busca en el espacio de nombres incorporado (*builtins namespace*). La declaración `global` debe preceder a todos los usos del nombre.

The *global* statement has the same scope as a name binding operation in the same block. If the nearest enclosing scope for a free variable contains a global statement, the free variable is treated as a global.

The *nonlocal* statement causes corresponding names to refer to previously bound variables in the nearest enclosing function scope. `SyntaxError` is raised at compile time if the given name does not exist in any enclosing function scope.

El espacio de nombres (*namespace*) para un módulo se crea automáticamente la primera vez que se importa el módulo. El módulo principal de un *script* siempre se llama `__main__`.

Los bloques de definición de clase y los argumentos para `exec()` y `eval()` son especiales en el contexto de la resolución de nombres. Una definición de clase es una declaración ejecutable que puede usar y definir nombres. Estas referencias siguen las reglas normales para la resolución de nombres con la excepción de que se buscan las variables locales no vinculadas en el espacio de nombres global. El espacio de nombres de la definición de clase se vuelve el diccionario de atributos de la clase. El ámbito de nombres definido en un bloque de clase está limitado a dicho bloque; no se extiende a los bloques de código de los métodos. Esto incluye las comprensiones y las expresiones generadoras (*generator expressions*), dado que están implementadas usando el alcance de función. Esto implica que lo siguiente fallará:

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

4.2.3 Builtins and restricted execution

CPython implementation detail: Los usuarios no deberían tocar `__builtins__`; es un detalle de la implementación en sentido estricto. Los usuarios que quieran sobreescribir valores en los espacios de nombres incorporados deberían usar *import* con el módulo `builtins` y modificar sus atributos de un modo adecuado.

El espacio de nombres incorporado (*builtin namespace*) asociado a la ejecución de un bloque de código es encontrado buscando el nombre `__builtins__` en su espacio de nombres global; debería ser un diccionario o un módulo (en este último caso, se usa el diccionario del módulo). Por defecto, en el módulo `__main__`, `__builtins__` es el módulo `builtins`. En cualquier otro módulo, `__builtins__` es un alias para el diccionario del propio módulo `builtins`.

4.2.4 Interacción con funcionalidades dinámicas

La resolución de variables libres sucede en tiempo de ejecución, no en tiempo de compilación. Esto significa que el siguiente código va a mostrar 42:

```
i = 10
def f():
    print(i)
i = 42
f()
```

The `eval()` and `exec()` functions do not have access to the full environment for resolving names. Names may be resolved in the local and global namespaces of the caller. Free variables are not resolved in the nearest enclosing namespace, but in the global namespace.[1] The `exec()` and `eval()` functions have optional arguments to override the global and local namespace. If only one namespace is specified, it is used for both.

4.3 Excepciones

Las excepciones son un medio para salir del flujo de control normal de un bloque de código, para gestionar errores u otras condiciones excepcionales. Una excepción es *lanzada* (*raised*) en el momento en que se detecta el error; puede ser *gestionada* (*handled*) por el bloque de código que la rodea o por cualquier bloque de código que directa o indirectamente ha invocado al bloque de código en el que ocurrió el error.

El intérprete Python lanza una excepción cuando detecta un error en tiempo de ejecución (como una división por cero). Un programa Python también puede lanzar una excepción explícitamente, con la declaración *raise*. Los gestores de excepciones se especifican con la declaración *try* … *except*. La cláusula *finally* de tales declaraciones puede utilizarse para especificar código de limpieza que no es el que gestiona la excepción, sino que se ejecutará en cualquier caso, tanto cuando la excepción ha ocurrido en el código que la precede, como cuando esto no ha sucedido.

Python usa el modelo de gestión de errores de «terminación» (»*termination*»): un gestor de excepción puede descubrir qué sucedió y continuar la ejecución en un nivel exterior, pero no puede reparar la causa del error y reintentar la operación que ha fallado (excepto que se reingrese al trozo de código fallido desde su inicio).

Cuando una excepción no está gestionada en absoluto, el intérprete termina la ejecución del programa, o retorna a su bucle principal interactivo. En cualquier caso, imprime un seguimiento de pila, excepto cuando la excepción es `SystemExit`.

Las excepciones están identificadas por instancias de clase. Se selecciona la cláusula *except* dependiendo de la clase de la instancia: debe referenciar a la clase de la instancia o a una clase base de la misma. La instancia puede ser recibida por el gestor y puede contener información adicional acerca de la condición excepcional.

[1] Esta limitación se da porque el código ejecutado por estas operaciones no está disponible en el momento en que se compila el módulo.

Nota: Los mensajes de excepción no forman parte de la API Python. Su contenido puede cambiar entre una versión de Python y la siguiente sin ningún tipo de advertencia; el código que corre bajo múltiples versiones del intérprete no debería basarse en estos mensajes.

Mira también la descripción de la declaración `try` en la sección *La sentencia try*, y la declaración `raise` en la sección *The raise statement*.

Notas al pie

CAPÍTULO 5

El sistema de importación

El código Python en un *módulo* obtiene acceso al código en otro módulo por el proceso de *importarlo*. La instrucción `import` es la forma más común de invocar la maquinaria de importación, pero no es la única manera. Funciones como `importlib.import_module()` y built-in `__import__()` también se pueden utilizar para invocar la maquinaria de importación.

La instrucción `import` combina dos operaciones; busca el módulo con nombre y, a continuación, enlaza los resultados de esa búsqueda a un nombre en el ámbito local. La operación de búsqueda de la instrucción `import` se define como una llamada a la función `__import__()`, con los argumentos adecuados. El valor retornado de `__import__()` se utiliza para realizar la operación de enlace de nombre de la instrucción `import`. Consulte la instrucción `import` para obtener los detalles exactos de esa operación de enlace de nombres.

Una llamada directa a `__import__()` realiza solo la búsqueda del módulo y, si se encuentra, la operación de creación del módulo. Aunque pueden producirse ciertos efectos secundarios, como la importación de paquetes primarios y la actualización de varias memorias caché (incluidas `sys.modules`), solo la instrucción `import` realiza una operación de enlace de nombres.

Cuando se ejecuta una instrucción `import`, se llama a la función estándar incorporada `__import__()`. Otros mecanismos para invocar el sistema de importación (como `importlib.import_module()`) pueden optar por omitir `__import__()` y utilizar sus propias soluciones para implementar la semántica de importación.

Cuando se importa un módulo por primera vez, Python busca el módulo y, si se encuentra, crea un objeto de módulo[1], inicializándolo. Si no se encuentra el módulo con nombre, se genera un `ModuleNotFoundError`. Python implementa varias estrategias para buscar el módulo con nombre cuando se invoca la maquinaria de importación. Estas estrategias se pueden modificar y ampliar mediante el uso de varios ganchos descritos en las secciones siguientes.

Distinto en la versión 3.3: El sistema de importación se ha actualizado para aplicar plenamente la segunda fase de **PEP 302**. Ya no hay ninguna maquinaria de importación implícita: todo el sistema de importación se expone a través de `sys.meta_path`. Además, se ha implementado la compatibilidad con paquetes de espacio de nombres nativos (consulte **PEP 420**).

[1] Véase `types.ModuleType`.

5.1 importlib

El módulo `importlib` proporciona una API enriquecida para interactuar con el sistema de importación. Por ejemplo `importlib.import_module()` proporciona una API recomendada y más sencilla que la integrada `__import__()` para invocar la maquinaria de importación. Consulte la documentación de la biblioteca `importlib` para obtener más detalles.

5.2 Paquetes

Python sólo tiene un tipo de objeto módulo, y todos los módulos son de este tipo, independientemente de si el módulo está implementado en Python, C, o en cualquier otro lenguage. Para ayudar a organizar los módulos y proporcionar una jerarquía de nombres, Python tiene un concepto de *paquete*.

Puedes pensar en los paquetes como los directorios de un sistema de archivos y en los módulos como archivos dentro de los directorios, pero no te tomes esta analogía demasiado literalmente, ya que los paquetes y los módulos no tienen por qué originarse en el sistema de archivos. Para los propósitos de esta documentación, usaremos esta conveniente analogía de directorios y archivos. Al igual que los directorios del sistema de archivos, los paquetes están organizados jerárquicamente, y los paquetes pueden contener subpaquetes, así como módulos regulares.

Es importante tener en cuenta que todos los paquetes son módulos, pero no todos los módulos son paquetes. O dicho de otro modo, los paquetes son sólo un tipo especial de módulo. Específicamente, cualquier módulo que contenga un atributo `__path__` se considera un paquete.

Todos los módulos tienen un nombre. Los nombres de los subpaquetes se separan del nombre del paquete padre por puntos, similar a la sintaxis de acceso a atributos estándar de Python. Así, puedes tener un módulo llamado `sys` y un paquete llamado `email`, que a su vez tiene un subpaquete llamado `email.mime` y un módulo dentro de ese subpaquete llamado `email.mime.text`.

5.2.1 Paquetes regulares

Python define dos tipos de paquetes, *paquetes regulares* y *paquetes de espacio de nombres*. Los paquetes regulares son los paquetes tradicionales tal y como existían en Python 3.2 y anteriores. Un paquete regular se implementa típicamente como un directorio que contiene un archivo `init__.py`. Cuando se importa un paquete regular, este archivo `__init__.py` se ejecuta implícitamente, y los objetos que define están vinculados a nombres en el espacio de nombres del paquete. El archivo `__init__.py` puede contener el mismo código Python que puede contener cualquier otro módulo, y Python añadirá algunos atributos adicionales al módulo cuando se importe.

Por ejemplo, la siguiente disposición del sistema de archivos define un paquete `parent` de nivel superior con tres subpaquetes:

```
parent/
    __init__.py
    one/
        __init__.py
    two/
        __init__.py
    three/
        __init__.py
```

Importando `parent.one` se ejecutará implícitamente `parent/__init__.py` y `parent/one/__init__.py`. La importación posterior de `parent.two` o `parent.three` ejecutará `parent/two/__init__.py` y `parent/three/__init__.py` respectivamente.

5.2.2 Paquetes de espacio de nombres

Un paquete de espacio de nombres es un compuesto de varias *porciones*, donde cada porción contribuye con un subpaquete al paquete padre. Las porciones pueden residir en diferentes lugares del sistema de archivos. Las porciones también pueden encontrarse en archivos zip, en la red, o en cualquier otro lugar que Python busque durante la importación. Los paquetes de espacios de nombres pueden corresponder o no directamente a objetos del sistema de archivos; pueden ser módulos virtuales que no tienen una representación concreta.

Los paquetes de espacios de nombres no usan una lista ordinaria para su atributo `__path__`. En su lugar utilizan un tipo iterable personalizado que realizará automáticamente una nueva búsqueda de porciones de paquete en el siguiente intento de importación dentro de ese paquete si la ruta de su paquete padre (o `sys.path` para un paquete de nivel superior) cambia.

Con los paquetes de espacio de nombres, no hay ningún archivo `parent/__init__.py`. De hecho, puede haber varios directorios `padre` encontrados durante la búsqueda de importación, donde cada uno de ellos es proporcionado por una parte diferente. Por lo tanto, `padre/one` no puede estar físicamente situado junto a `padre/two`. En este caso, Python creará un paquete de espacio de nombres para el paquete `parent` de nivel superior siempre que se importe él o uno de sus subpaquetes.

Consulte también **PEP 420** para conocer la especificación del paquete de espacio de nombres.

5.3 Buscando

Para comenzar la búsqueda, Python necesita el nombre *totalmente calificado* del módulo (o paquete, pero para los fines de esta discusión, la diferencia es irrelevante) que se está importando. Este nombre puede provenir de varios argumentos a la instrucción *import*, o de los parámetros de las funciones `importlib.import_module()` o `__import__()`.

Este nombre se utilizará en varias fases de la búsqueda de importación, y puede ser la ruta de acceso punteada a un submódulo, por ejemplo, `foo.bar.baz`. En este caso, Python primero intenta importar `foo`, luego `foo.bar`, y finalmente `foo.bar.baz`. Si se produce un error en cualquiera de las importaciones intermedias, se genera un `ModuleNotFoundError`.

5.3.1 La caché del módulo

El primer lugar comprobado durante la búsqueda de importación es `sys.modules`. Esta asignación sirve como caché de todos los módulos que se han importado previamente, incluidas las rutas intermedias. Por lo tanto, si `foo.bar.baz` se importó previamente, `sys.modules` contendrá entradas para `foo`, `foo.bar`, y `foo.bar.baz`. Cada clave tendrá como valor el objeto de módulo correspondiente.

Durante la importación, el nombre del módulo se busca en `sys.modules` y si está presente, el valor asociado es el módulo que satisface la importación y el proceso se completa. Sin embargo, si el valor es `None`, se genera un `ModuleNotFoundError`. Si falta el nombre del módulo, Python continuará buscando el módulo.

`sys.modules` se puede escribir. La eliminación de una clave no puede destruir el módulo asociado (ya que otros módulos pueden contener referencias a él), pero invalidará la entrada de caché para el módulo con nombre, lo que hará que Python busque de nuevo el módulo con nombre en su próxima importación. La clave también se puede asignar a `None`, lo que obliga a la siguiente importación del módulo a dar como resultado un `ModuleNotFoundError`.

Tenga cuidado, sin embargo, como si mantiene una referencia al objeto module, invalide su entrada de caché en `sys.modules` y, a continuación, vuelva a importar el módulo con nombre, los dos objetos de módulo *no* serán los mismos. Por el contrario, `importlib.reload()` reutilizará el objeto de módulo *same* y simplemente reinicializará el contenido del módulo volviendo a ejecutar el código del módulo.

5.3.2 Buscadores y cargadores

Si el módulo con nombre no se encuentra en `sys.modules`, se invoca el protocolo de importación de Python para buscar y cargar el módulo. Este protocolo consta de dos objetos conceptuales, *buscadores* y *cargadores*. El trabajo de un buscador es determinar si puede encontrar el módulo con nombre utilizando cualquier estrategia que conozca. Los objetos que implementan ambas interfaces se conocen como *importadores* se retornan a sí mismos cuando descubren que pueden cargar el módulo solicitado.

Python incluye una serie de buscadores e importadores predeterminados. El primero sabe cómo localizar módulos integrados, y el segundo sabe cómo localizar módulos congelados. Un tercer buscador predeterminado busca módulos en *import path*. El *import path* es una lista de ubicaciones que pueden nombrar rutas del sistema de archivos o archivos zip. También se puede ampliar para buscar cualquier recurso localizable, como los identificados por las direcciones URL.

La maquinaria de importación es extensible, por lo que se pueden añadir nuevos buscadores para ampliar el alcance y el alcance de la búsqueda de módulos.

En realidad, los buscadores no cargan módulos. Si pueden encontrar el módulo con nombre, retornan un *module spec*, una encapsulación de la información relacionada con la importación del módulo, que la maquinaria de importación utiliza al cargar el módulo.

En las secciones siguientes se describe el protocolo para buscadores y cargadores con más detalle, incluido cómo puede crear y registrar otros nuevos para ampliar la maquinaria de importación.

Distinto en la versión 3.4: En versiones anteriores de Python, los buscadores retornaban *cargadores* directamente, mientras que ahora retornen especificaciones de módulo que *contienen* cargadores. Los cargadores todavía se utilizan durante la importación, pero tienen menos responsabilidades.

5.3.3 Ganchos de importación

La maquinaria de importación está diseñada para ser extensible; el mecanismo principal para esto son los *ganchos de importación* (import hooks). Hay dos tipos de ganchos de importación: *meta hooks* (meta ganchos) y *import path hooks* (ganchos de ruta de acceso de importación).

Los meta ganchos se llaman al inicio del procesamiento de importación, antes de que se haya producido cualquier otro procesamiento de importación, que no sea búsqueda de caché de `sys.modules`. Esto permite que los metaganchos reemplacen el procesamiento de `sys.path`, módulos congelados o incluso módulos integrados. Los meta ganchos se registran agregando nuevos objetos de buscador a `sys.meta_path`, como se describe a continuación.

Los ganchos de ruta de acceso de importación se invocan como parte del procesamiento `sys.path` (o `package.__path__`), en el punto donde se encuentra su elemento de ruta de acceso asociado. Los ganchos de ruta de acceso de importación se registran agregando nuevos invocables a `sys.path_hooks` como se describe a continuación.

5.3.4 La meta ruta (*path*)

Cuando el módulo con nombre no se encuentra en `sys.modules`, Python busca a continuación `sys.meta_path`, que contiene una lista de objetos buscadores de metarutas. Estos buscadores se consultan para ver si saben cómo manejar el módulo nombrado. Los buscadores de rutas de meta deben implementar un método llamado `find_spec()` que toma tres argumentos: un nombre, una ruta de importación y (opcionalmente) un módulo de destino. El buscador de metarutas puede usar cualquier estrategia que desee para determinar si puede manejar el módulo con nombre o no.

Si el buscador de metarutas sabe cómo controlar el módulo con nombre, retorna un objeto de especificación. Si no puede controlar el módulo con nombre, retorna `None`. Si el procesamiento de `sys.meta_path` llega al final de su lista sin retornar una especificación, se genera un `ModuleNotFoundError`. Cualquier otra excepción provocada simplemente se propaga hacia arriba, anulando el proceso de importación.

El método de los buscadores de metarutas de `find_spec()` se llama con dos o tres argumentos. El primero es el nombre completo del módulo que se está importando, por ejemplo `foo.bar.baz`. El segundo argumento son las entradas de

ruta de acceso que se utilizarán para la búsqueda de módulos. Para los módulos de nivel superior, el segundo argumento es `None`, pero para submódulos o subpaquetes, el segundo argumento es el valor del atributo `__path__` del paquete primario. Si no se puede tener acceso al atributo `__path__` adecuado, se genera un `ModuleNotFoundError`. El tercer argumento es un objeto de módulo existente que será el destino de la carga más adelante. El sistema de importación pasa un módulo de destino solo durante la recarga.

La metaruta se puede recorrer varias veces para una sola solicitud de importación. Por ejemplo, suponiendo que ninguno de los módulos implicados ya se haya almacenado en caché, la importación de `foo.bar.baz` realizará primero una importación de nivel superior, llamando a `mpf.find_spec("foo", None, None)` en cada buscador de metarutas (`mpf`). Después de importar `foo` , `foo.bar` se importará atravesando la meta ruta por segunda vez, llamando a `mpf.find_spec("foo.bar", foo.__path__, None)`. Una vez importado `foo.bar`, el recorrido final llamará a `mpf.find_spec("foo.bar.baz", foo.bar.__path__, None)`.

Algunos buscadores de metarutas solo admiten importaciones de nivel superior. Estos importadores siempre retornarán `None` cuando se pase algo distinto de `None` como segundo argumento.

El valor predeterminado de Python `sys.meta_path` tiene tres buscadores de metarutas, uno que sabe cómo importar módulos integrados, uno que sabe cómo importar módulos congelados y otro que sabe cómo importar módulos desde un *import path* (es decir, el *path based finder*).

Distinto en la versión 3.4: El método `find_spec()` de los buscadores de metarutas de la ruta de acceso reemplazó `find_module()`, que ahora está en desuso. Aunque seguirá funcionando sin cambios, la maquinaria de importación sólo lo intentará si el buscador no implementa `find_spec()`.

5.4 Cargando

Si se encuentra una especificación de módulo, la maquinaria de importación la utilizará (y el cargador que contiene) al cargar el módulo. Aquí está una aproximación de lo que sucede durante la porción de carga de la importación:

```
module = None
if spec.loader is not None and hasattr(spec.loader, 'create_module'):
    # It is assumed 'exec_module' will also be defined on the loader.
    module = spec.loader.create_module(spec)
if module is None:
    module = ModuleType(spec.name)
# The import-related module attributes get set here:
_init_module_attrs(spec, module)

if spec.loader is None:
    # unsupported
    raise ImportError
if spec.origin is None and spec.submodule_search_locations is not None:
    # namespace package
    sys.modules[spec.name] = module
elif not hasattr(spec.loader, 'exec_module'):
    module = spec.loader.load_module(spec.name)
    # Set __loader__ and __package__ if missing.
else:
    sys.modules[spec.name] = module
    try:
        spec.loader.exec_module(module)
    except BaseException:
        try:
            del sys.modules[spec.name]
        except KeyError:
            pass
```

(continué en la próxima página)

(proviene de la página anterior)

```
        raise
return sys.modules[spec.name]
```

Tenga en cuenta los siguientes detalles:

- Si hay un objeto de módulo existente con el nombre dado en `sys.modules`, la importación ya lo habrá retornado.
- El módulo existirá en `sys.modules` antes de que el cargador ejecute el código del módulo. Esto es crucial porque el código del módulo puede (directa o indirectamente) importarse a sí mismo; agregándolo a `sys.modules` de antemano evita la recursividad sin límites en el peor de los casos y la carga múltiple en el mejor.
- Si se produce un error en la carga, el módulo con errores – y solo el módulo con errores – se elimina de `sys.modules`. Cualquier módulo que ya esté en la caché de `sys.modules` y cualquier módulo que se haya cargado correctamente como efecto secundario, debe permanecer en la memoria caché. Esto contrasta con la recarga donde incluso el módulo que falla se deja en `sys.modules`.
- Después de crear el módulo pero antes de la ejecución, la maquinaria de importación establece los atributos del módulo relacionados con la importación («_init_module_attrs» en el ejemplo de pseudocódigo anterior), como se resume en una *sección posterior*.
- La ejecución del módulo es el momento clave de la carga en el que se rellena el espacio de nombres del módulo. La ejecución se delega por completo en el cargador, lo que llega a decidir qué se rellena y cómo.
- El módulo creado durante la carga y pasado a exec_module() puede no ser el que se retorna al final de la importación[2].

Distinto en la versión 3.4: El sistema de importación se ha hecho cargo de las responsabilidades reutilizables de los cargadores. Estos fueron realizados previamente por el método `importlib.abc.Loader.load_module()`.

5.4.1 Cargadores

Los cargadores de módulos proporcionan la función crítica de carga: ejecución del módulo. La maquinaria de importación llama al método `importlib.abc.Loader.exec_module()` con un único argumento, el objeto module que se va a ejecutar. Se omite cualquier valor retornado de `exec_module()`.

Los cargadores deben cumplir los siguientes requisitos:

- Si el módulo es un módulo Python (a diferencia de un módulo integrado o una extensión cargada dinámicamente), el cargador debe ejecutar el código del módulo en el espacio de nombres global del módulo (`module.__dict__`).
- Si el cargador no puede ejecutar el módulo, debe generar un `ImportError`, aunque se propagará cualquier otra excepción provocada durante `exec_module()`.

En muchos casos, el buscador y el cargador pueden ser el mismo objeto; en tales casos, el método `find_spec()` simplemente retornaría una especificación con el cargador establecido en `self`.

Los cargadores de módulos pueden optar por crear el objeto de módulo durante la carga mediante la implementación de un método `create_module()`. Toma un argumento, el module spec, y retorna el nuevo objeto de módulo que se usará durante la carga. `create_module()` no necesita establecer ningún atributo en el objeto module. Si el método retorna `None`, la maquinaria de importación creará el nuevo módulo en sí.

Nuevo en la versión 3.4: El método de cargadores `create_module()`.

Distinto en la versión 3.4: El método `load_module()` fue reemplazado por `exec_module()` y la maquinaria de importación asumió todas las responsabilidades reutilizables de la carga.

[2] La implementación de importlib evita usar el valor retornado directamente. En su lugar, obtiene el objeto module buscando el nombre del módulo en `sys.modules`. El efecto indirecto de esto es que un módulo importado puede sustituirse a sí mismo en `sys.modules`. Este es un comportamiento específico de la implementación que no se garantiza que funcione en otras implementaciones de Python.

Para la compatibilidad con los cargadores existentes, la maquinaria de importación utilizará el método de cargadores `load_module()` si existe y el cargador no implementa también `exec_module()`. Sin embargo, `load_module()` ha quedado obsoleto y los cargadores deben implementar `exec_module()` en su lugar.

El método `load_module()` debe implementar toda la funcionalidad de carga reutilizable descrita anteriormente, además de ejecutar el módulo. Se aplican todas las mismas restricciones, con algunas aclaraciones adicionales:

- Si hay un objeto de módulo existente con el nombre dado en `sys.modules`, el cargador debe utilizar ese módulo existente. (De lo contrario, `importlib.reload()` no funcionará correctamente.) Si el módulo con nombre no existe en `sys.modules`, el cargador debe crear un nuevo objeto de módulo y agregarlo a `sys.modules`.
- El módulo *debe* existir en `sys.modules` antes de que el cargador ejecute el código del módulo, para evitar la recursividad sin límites o la carga múltiple.
- Si se produce un error en la carga, el cargador debe quitar los módulos que ha insertado en `sys.modules`, pero debe quitar **solo** los módulos con errores, y solo si el propio cargador ha cargado los módulos explícitamente.

Distinto en la versión 3.5: A `DeprecationWarning` se genera cuando se define `exec_module()` pero `create_module()` no lo es.

Distinto en la versión 3.6: Un `ImportError` se genera cuando `exec_module()` está definido, pero `create_module()` no lo es.

5.4.2 Sub-modulos

Cuando se carga un submódulo mediante cualquier mecanismo (por ejemplo, API `importlib`, las instrucciones `import` o `import-from`, o `__import__()`) integradas, se coloca un enlace en el espacio de nombres del módulo primario al objeto submodule. Por ejemplo, si el paquete `spam` tiene un submódulo `foo`, después de importar `spam.foo`, `spam` tendrá un atributo `foo` que está enlazado al submódulo. Supongamos que tiene la siguiente estructura de directorios:

```
spam/
    __init__.py
    foo.py
    bar.py
```

y `spam/__init__.py` tiene las siguientes líneas:

```
from .foo import Foo
from .bar import Bar
```

a continuación, la ejecución de lo siguiente pone un nombre vinculante para `foo` y `bar` en el módulo `spam`:

```
>>> import spam
>>> spam.foo
<module 'spam.foo' from '/tmp/imports/spam/foo.py'>
>>> spam.bar
<module 'spam.bar' from '/tmp/imports/spam/bar.py'>
```

Dadas las reglas de enlace de nombres familiares de Python, esto puede parecer sorprendente, pero en realidad es una característica fundamental del sistema de importación. La retención invariable es que si tiene `sys.modules[`spam`]` y `sys.modules[`spam.foo`]` (como lo haría después de la importación anterior), este último debe aparecer como el atributo `foo` de la primera.

5.4.3 Especificaciones del módulo

La maquinaria de importación utiliza una variedad de información sobre cada módulo durante la importación, especialmente antes de la carga. La mayor parte de la información es común a todos los módulos. El propósito de las especificaciones de un módulo es encapsular esta información relacionada con la importación por módulo.

El uso de una especificación durante la importación permite transferir el estado entre los componentes del sistema de importación, por ejemplo, entre el buscador que crea la especificación del módulo y el cargador que la ejecuta. Lo más importante es que permite a la maquinaria de importación realizar las operaciones de caldera de carga, mientras que sin una especificación de módulo el cargador tenía esa responsabilidad.

La especificación del módulo se expone como el atributo `__spec__` en un objeto de módulo. Consulte `ModuleSpec` para obtener más información sobre el contenido de la especificación del módulo.

Nuevo en la versión 3.4.

5.4.4 Atributos de módulo relacionados con la importación

La máquina de importación rellena estos atributos en cada objeto de módulo durante la carga, en función de las especificaciones del módulo, antes de que el cargador ejecute el módulo.

`__name__`
: El atributo `__name__` debe establecerse en el nombre completo del módulo. Este nombre se utiliza para identificar de forma única el módulo en el sistema de importación.

`__loader__`
: El atributo `__loader__` debe establecerse en el objeto de cargador que utilizó la máquina de importación al cargar el módulo. Esto es principalmente para la introspección, pero se puede utilizar para la funcionalidad específica del cargador adicional, por ejemplo, obtener datos asociados con un cargador.

`__package__`
: Se debe establecer el atributo `__package__` del módulo. Su valor debe ser una cadena, pero puede ser el mismo valor que su `__name__`. Cuando el módulo es un paquete, su valor `__package__` debe establecerse en su `__name__`. Cuando el módulo no es un paquete, `__package__` debe establecerse en la cadena vacía para los módulos de nivel superior, o para los submódulos, en el nombre del paquete primario. Consulte **PEP 366** para obtener más detalles.

  Este atributo se utiliza en lugar de `__name__` para calcular importaciones relativas explícitas para los módulos principales, tal como se define en **PEP 366**. Se espera que tenga el mismo valor que `__spec__.parent`.

  Distinto en la versión 3.6: Se espera que el valor de `__package__` sea el mismo que `__spec__.parent`.

`__spec__`
: El atributo `__spec__` debe establecerse en la especificación de módulo que se utilizó al importar el módulo. Establecer `__spec__` se aplica correctamente por igual a *módulos inicializados durante el inicio del intérprete*. La única excepción es `__main__`, donde `__spec__` es *establecido None en algunos casos*.

  Cuando `__package__` no está definido, `__spec__.parent` se utiliza como reserva.

  Nuevo en la versión 3.4.

  Distinto en la versión 3.6: `__spec__.parent` se utiliza como reserva cuando `__package__`` no está definido.

`__path__`
: Si el módulo es un paquete (normal o espacio de nombres), se debe establecer el atributo `__path__` del objeto de módulo. El valor debe ser iterable, pero puede estar vacío si `__path__` no tiene más importancia. Si `__path__` no está vacío, debe producir cadenas cuando se itera. Más detalles sobre la semántica de `__path__` se dan *below*.

  Los módulos que no son de paquete no deben tener un atributo `__path__`.

__file__

__cached__

__file__ es opcional. Si se establece, el valor de este atributo debe ser una cadena. El sistema de importación puede optar por dejar __file__ sin establecer si no tiene un significado semántico (por ejemplo, un módulo cargado desde una base de datos).

Si se establece __file__, también puede ser apropiado establecer el atributo __cached__, que es la ruta de acceso a cualquier versión compilada del código (por ejemplo, archivo compilado por bytes). No es necesario que exista el archivo para establecer este atributo; la ruta de acceso puede simplemente apuntar a donde existiría el archivo compilado (consulte **PEP 3147**).

También es apropiado establecer __cached__ cuando __file__ no está establecido. Sin embargo, ese escenario es bastante atípico. En última instancia, el cargador es lo que hace uso de __file__ y/o __cached__. Por lo tanto, si un cargador puede cargar desde un módulo almacenado en caché pero no se carga desde un archivo, ese escenario atípico puede ser adecuado.

5.4.5 module.__path__

Por definición, si un módulo tiene un atributo __path__, es un paquete.

El atributo __path__ de un paquete se utiliza durante las importaciones de sus subpaquetes. Dentro de la maquinaria de importación, funciona de la misma manera que sys.path, es decir, proporcionando una lista de ubicaciones para buscar módulos durante la importación. Sin embargo, __path__ suele estar mucho más restringido que sys.path.

__path__ debe ser un iterable de cadenas, pero puede estar vacío. Las mismas reglas utilizadas para sys.path también se aplican a la __path__ de un paquete, y sys.path_hooks (descrito a continuación) se consultan al recorrer el __path__ de un paquete.

El archivo __init__.py de un paquete puede establecer o modificar el atributo __path__ del paquete, y esta era normalmente la forma en que los paquetes de espacio de nombres se implementaban antes de **PEP 420**. Con la adopción de **PEP 420**, los paquetes de espacio de nombres ya no necesitan proporcionar archivos __init__.py que contienen solo el código de manipulación __path__; la máquina de importación establece automáticamente __path__ correctamente para el paquete de espacio de nombres.

5.4.6 Representación (*Reprs*) de módulos

De forma predeterminada, todos los módulos tienen un repr utilizable, sin embargo, dependiendo de los atributos establecidos anteriormente, y en las especificaciones del módulo, puede controlar más explícitamente el repr de los objetos de módulo.

Si el módulo tiene una especificación (__spec__), la maquinaria de importación intentará generar un repr a partir de él. Si eso falla o no hay ninguna especificación, el sistema de importación creará un repr predeterminado usando cualquier información disponible en el módulo. Intentará utilizar el module.__name__, module.__file__ y module.__loader__ como entrada en el repr, con valores predeterminados para cualquier información que falte.

Aquí están las reglas exactas utilizadas:

- Si el módulo tiene un atributo __spec__, la información de la especificación se utiliza para generar el repr. Se consultan los atributos «name», «loader», «origin» y «has_location».
- Si el módulo tiene un atributo __file__, se utiliza como parte del repr del módulo.
- Si el módulo no tiene __file__ pero tiene un __loader__ que no es None, entonces el repr del cargador se utiliza como parte del repr del módulo.
- De lo contrario, sólo tiene que utilizar el __name__ del módulo en el repr.

Distinto en la versión 3.4: El uso de `loader.module_repr()` ha quedado obsoleto y la máquina de importación utiliza ahora la especificación del módulo para generar un repr de módulo.

Para la compatibilidad con versiones anteriores de Python 3.3, el repr del módulo se generará llamando al método `module_repr()` del cargador, si se define, antes de probar cualquiera de los enfoques descritos anteriormente. Sin embargo, el método está en desuso.

5.4.7 Invalidación del código de bytes en caché

Before Python loads cached bytecode from a `.pyc` file, it checks whether the cache is up-to-date with the source `.py` file. By default, Python does this by storing the source's last-modified timestamp and size in the cache file when writing it. At runtime, the import system then validates the cache file by checking the stored metadata in the cache file against the source's metadata.

Python también admite archivos de caché «basados en hash», que almacenan un hash del contenido del archivo de origen en lugar de sus metadatos. Hay dos variantes de archivos `.pyc` basados en hash: marcados y desmarcados. Para los archivos ``.pyc` marcados basados en hash, Python valida el archivo de caché mediante el hash del archivo de origen y la comparación del hash resultante con el hash en el archivo de caché. Si se encuentra que un archivo de caché basado en hash comprobado no es válido, Python lo regenera y escribe un nuevo archivo de caché basado en hash comprobado. Para los archivos `.pyc` sin marcar en hash, Python simplemente asume que el archivo de caché es válido si existe. El comportamiento de validación de archivos basado en hash `.pyc` se puede invalidar con el indicador `--check-hash-based-pycs`.

Distinto en la versión 3.7: Se han añadido archivos `.pyc` basados en hash. Anteriormente, Python solo admitía la invalidación basada en la marca de tiempo de la caché del código de bytes.

5.5 El buscador basado en rutas

Como se mencionó anteriormente, Python viene con varios buscadores de meta rutas predeterminados. Uno de ellos, llamado el buscador *path based finder* (`PathFinder`), busca una *import path*, que contiene una lista de *entradas de ruta*. Cada entrada de ruta de acceso nombra una ubicación para buscar módulos.

El buscador basado en rutas en sí no sabe cómo importar nada. En su lugar, atraviesa las entradas de ruta individuales, asociando cada una de ellas con un buscador de entrada de ruta que sabe cómo manejar ese tipo particular de ruta de acceso.

El conjunto predeterminado de buscadores de entradas de ruta implementa toda la semántica para encontrar módulos en el sistema de archivos, controlando tipos de archivos especiales como el código fuente de Python (archivos `.py`), el código de bytes de Python (archivos `.pyc`) y las bibliotecas compartidas (por ejemplo, archivos `.so`). Cuando es compatible con el módulo `zipimport` en la biblioteca estándar, los buscadores de entradas de ruta de acceso predeterminados también controlan la carga de todos estos tipos de archivo (excepto las bibliotecas compartidas) desde zipfiles.

Las entradas de ruta de acceso no deben limitarse a las ubicaciones del sistema de archivos. Pueden hacer referencia a direcciones URL, consultas de base de datos o cualquier otra ubicación que se pueda especificar como una cadena.

El buscador basado en rutas proporciona enlaces y protocolos adicionales para que pueda ampliar y personalizar los tipos de entradas de ruta de acceso que se pueden buscar. Por ejemplo, si desea admitir entradas de ruta de acceso como direcciones URL de red, podría escribir un enlace que implemente la semántica HTTP para buscar módulos en la web. Este gancho (un al que se puede llamar) retornaría un *path entry finder* compatible con el protocolo descrito a continuación, que luego se utilizó para obtener un cargador para el módulo de la web.

Una palabra de advertencia: esta sección y la anterior utilizan el término *finder*, distinguiendo entre ellos utilizando los términos *meta path finder* y *path entry finder*. Estos dos tipos de buscadores son muy similares, admiten protocolos similares y funcionan de maneras similares durante el proceso de importación, pero es importante tener en cuenta que

son sutilmente diferentes. En particular, los buscadores de meta path operan al principio del proceso de importación, como se indica en el recorrido `sys.meta_path`.

Por el contrario, los buscadores de entradas de ruta son en cierto sentido un detalle de implementación del buscador basado en rutas y, de hecho, si el buscador basado en rutas se eliminara de `sys.meta_path`, no se invocaría ninguna semántica del buscador de entradas de ruta.

5.5.1 Buscadores de entradas de ruta

El *path based finder* es responsable de encontrar y cargar módulos y paquetes de Python cuya ubicación se especifica con una cadena *path entry*. La mayoría de las ubicaciones de nombres de entradas de ruta de acceso en el sistema de archivos, pero no es necesario limitarlas a esto.

Como buscador de meta rutas, el buscador *path based finder* implementa el protocolo `find_spec()` descrito anteriormente, sin embargo, expone enlaces adicionales que se pueden usar para personalizar cómo se encuentran y cargan los módulos desde la ruta *import path*.

Tres variables son usadas por *path based finder*, `sys.path`, `sys.path_hooks` y `sys.path_importer_cache`. También se utilizan los atributos `__path__` en los objetos de paquete. Estos proporcionan formas adicionales de personalizar la maquinaria de importación.

`sys.path` contiene una lista de cadenas que proporcionan ubicaciones de búsqueda para módulos y paquetes. Se inicializa a partir de la variable de entorno `PYTHONPATH` y varios otros valores predeterminados específicos de la instalación e implementación. Las entradas de `sys.path` pueden nombrar directorios en el sistema de archivos, archivos zip y potencialmente otras «ubicaciones» (consulte el módulo `site`) que se deben buscar para módulos, como direcciones URL o consultas de base de datos. Solo las cadenas y bytes deben estar presentes en `sys.path`; todos los demás tipos de datos se omiten. La codificación de las entradas de bytes viene determinada por los *buscadores de entrada de ruta*.

El buscador *path based finder* es un *meta path finder*, por lo que la maquinaria de importación comienza la búsqueda *import path* llamando al método `find_spec()` basado en la ruta de acceso, tal como se describió anteriormente. Cuando se proporciona el argumento `path` a `find_spec()`, será una lista de rutas de acceso de cadena para recorrer - normalmente el atributo `__path__` de un paquete para una importación dentro de ese paquete. Si el argumento `path` es `None`, esto indica una importación de nivel superior y se utiliza `sys.path`.

El buscador basado en rutas de acceso recorre en iteración cada entrada de la ruta de búsqueda y, para cada una de ellas, busca un *path entry finder* adecuado (`PathEntryFinder`) para la entrada de ruta de acceso. Dado que esto puede ser una operación costosa (por ejemplo, puede haber sobrecargas de llamadas *stat()* para esta búsqueda), el buscador basado en rutas mantiene una ruta de acceso de asignación de caché entradas a los buscadores de entrada de ruta. Esta memoria caché se mantiene en `sys.path_importer_cache` (a pesar del nombre, esta caché almacena realmente objetos de buscador en lugar de limitarse a objetos *importer*). De esta manera, la costosa búsqueda de una ubicación en particular *path entry path entry finder* solo debe hacerse una vez. El código de usuario es libre de eliminar las entradas de caché de `sys.path_importer_cache` obligando al buscador basado en ruta de acceso a realizar de nuevo la búsqueda de entrada de ruta[3].

Si la entrada de ruta de acceso no está presente en la memoria caché, el buscador basado en rutas de acceso recorre en iteración cada llamada que se puede llamar en `sys.path_hooks`. Cada uno de los enlaces de *ganchos de rutas de entrada* en esta lista se llama con un solo argumento, la entrada de ruta de acceso que se va a buscar. Esta invocable puede retornar un *path entry finder* que puede controlar la entrada de ruta de acceso, o puede generar `ImportError`. Un `ImportError` es utilizado por el buscador basado en ruta para indicar que el gancho no puede encontrar un *path entry finder* para eso *entrada de ruta*. Se omite la excepción y la iteración *import path* continúa. El enlace debe esperar un objeto de rutas o bytes; la codificación de objetos bytes está hasta el enlace (por ejemplo, puede ser una codificación del sistema de archivos, UTF-8, o algo más), y si el gancho no puede decodificar el argumento, debe generar `ImportError`.

Si la iteración `sys.path_hooks` termina sin que se retorne ningún valor *path entry finder*, a continuación, el método de búsqueda basado en la ruta de acceso `find_spec()` almacenará `None` en `sys.path_importer_cache` (para

[3] En el código heredado, es posible encontrar instancias de `imp. NullImporter` en el `sys.path_importer_cache`. Se recomienda cambiar el código para usar `None` en su lugar. Consulte portingpythoncode para obtener más detalles.

indicar que no hay ningún buscador para esta entrada de ruta) y retornará `None`, lo que indica que este *meta path finder* no pudo encontrar el módulo.

Si un *path entry finder* *is* retornado por uno de los *path entry hook* invocables en `sys.path_hooks`, entonces el siguiente protocolo se utiliza para pedir al buscador una especificación de módulo, que luego se utiliza al cargar el módulo.

El directorio de trabajo actual, denotado por una cadena vacía, se controla de forma ligeramente diferente de otras entradas de `sys.path`. En primer lugar, si se encuentra que el directorio de trabajo actual no existe, no se almacena ningún valor en `sys.path_importer_cache`. En segundo lugar, el valor del directorio de trabajo actual se busca actualizado para cada búsqueda de módulo. En tercer lugar, la ruta de acceso utilizada para `sys.path_importer_cache` y retornada por `importlib.machinery.PathFinder.find_spec()` será el directorio de trabajo actual real y no la cadena vacía.

5.5.2 Buscadores de entradas de ruta

Para admitir las importaciones de módulos y paquetes inicializados y también para contribuir con partes a paquetes de espacio de nombres, los buscadores de entradas de ruta de acceso deben implementar el método `importlib.abc.PathEntryFinder.find_spec()`.

`importlib.abc.PathEntryFinder.find_spec` `()` toma dos argumentos: el nombre completo del módulo que se va a importar y el módulo de destino (opcional). `find_spec()` retorna una especificación completamente poblada para el módulo. Esta especificación siempre tendrá «cargador» establecido (con una excepción).

To indicate to the import machinery that the spec represents a namespace *portion*, the path entry finder sets «submodule_search_locations» to a list containing the portion.

Distinto en la versión 3.4: `find_spec()` reemplazó a `find_loader()` y `find_module()`, los cuales ahora están en desuso, pero se utilizarán si `find_spec()` no está definido.

Los buscadores de entradas de ruta más antiguos pueden implementar uno de estos dos métodos en desuso en lugar de `find_spec()`. Los métodos todavía se respetan en aras de la compatibilidad con versiones anteriores. Sin embargo, si `find_spec()` se implementa en el buscador de entrada de ruta, se omiten los métodos heredados.

`find_loader()` takes one argument, the fully qualified name of the module being imported. `find_loader()` returns a 2-tuple where the first item is the loader and the second item is a namespace *portion*.

Para la compatibilidad con versiones anteriores con otras implementaciones del protocolo de importación, muchos buscadores de entradas de ruta de acceso también admiten el mismo método tradicional `find_module()` que admiten los buscadores de rutas de acceso meta. Sin embargo, nunca se llama a los métodos del buscador de entradas de ruta `find_module()` con un argumento `path` (se espera que registren la información de ruta adecuada desde la llamada inicial al enlace de ruta).

El método `find_module()` en los buscadores de entrada de ruta está en desuso, ya que no permite que el buscador de entradas de ruta de acceso aporte partes a paquetes de espacio de nombres. Si existen tanto `find_loader()` como `find_module()` en un buscador de entrada de ruta, el sistema de importación siempre llamará a `find_loader()` en lugar de `find_module()`.

5.6 Reemplazando el sistema de importación estándar

El mecanismo más confiable para reemplazar todo el sistema de importación es eliminar el contenido predeterminado de `sys.meta_path`, sustituyéndolos por completo por un enlace de meta path personalizado.

Si es aceptable alterar únicamente el comportamiento de las declaraciones de importación sin afectar a otras API que acceden al sistema de importación, puede ser suficiente reemplazar la función incorporada `__import__()`. Esta técnica también puede emplearse a nivel de módulo para alterar únicamente el comportamiento de las declaraciones de importación dentro de ese módulo.

Para evitar selectivamente la importación de algunos módulos de un enlace al principio de la meta path (en lugar de deshabilitar completamente el sistema de importación estándar), es suficiente elevar `ModuleNotFoundError` directamente desde `find_spec()` en lugar de retornar `None`. Este último indica que la búsqueda de meta path debe continuar, mientras que la generación de una excepción termina inmediatamente.

5.7 Paquete Importaciones relativas

Las importaciones relativas utilizan puntos iniciales. Un único punto inicial indica una importación relativa, empezando por el paquete actual. Dos o más puntos iniciales indican una importación relativa a los elementos primarios del paquete actual, un nivel por punto después del primero. Por ejemplo, dado el siguiente diseño de paquete:

```
package/
    __init__.py
    subpackage1/
        __init__.py
        moduleX.py
        moduleY.py
    subpackage2/
        __init__.py
        moduleZ.py
    moduleA.py
```

En `subpackage1/moduleX.py` o `subpackage1/__init__.py`, las siguientes son importaciones relativas válidas:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
```

Las importaciones absolutas pueden utilizar la sintaxis `import <>` o `from <> import <>`, pero las importaciones relativas solo pueden usar el segundo formulario; la razón de esto es que:

```
import XXX.YYY.ZZZ
```

debe exponer `XXX. Yyy. ZZZ` como una expresión utilizable, pero .moduleY no es una expresión válida.

5.8 Consideraciones especiales para __main__

El módulo `__main__` es un caso especial relativo al sistema de importación de Python. Como se señaló *elsewhere*, el módulo `__main__` se inicializa directamente al inicio del intérprete, al igual que `sys` y `builtins`. Sin embargo, a diferencia de esos dos, no califica estrictamente como un módulo integrado. Esto se debe a que la forma en que se inicializa `__main__` depende de las marcas y otras opciones con las que se invoca el intérprete.</programs>

5.8.1 __main__.__spec__

Dependiendo de cómo se inicializa `__main__`, `__main__.__spec__` se establece correctamente o en `None`.

Cuando Python se inicia con la opción `-m`, `__spec__` se establece en la especificación de módulo del módulo o paquete correspondiente. `__spec__` también se rellena cuando el módulo `__main__` se carga como parte de la ejecución de un directorio, zipfile u otro `sys.path` entrada.

En los casos restantes `__main__.__spec__` se establece en `None`, ya que el código utilizado para rellenar el `__main__` no se corresponde directamente con un módulo importable:</using-on-interface-options>

- mensaje interactivo
- opción `-c`
- ejecutando desde stdin
- que se ejecuta directamente desde un archivo de código fuente o de código de bytes

Tenga en cuenta que `__main__.__spec__` siempre es `None` en el último caso, *incluso si* el archivo técnicamente podría importarse directamente como un módulo en su lugar. Utilice el modificador `-m` si se desean metadatos de módulo válidos en `__main__`.

Tenga en cuenta también que incluso cuando `__main__` corresponde a un módulo importable y `__main__.__spec__` se establece en consecuencia, todavía se consideran módulos *distinct*. Esto se debe al hecho de que los bloques protegidos por las comprobaciones `if __name__ == "__main__":` solo se ejecutan cuando el módulo se utiliza para rellenar el espacio de nombres `__main__`, y no durante la importación normal.

5.9 Problemas sin resolver

XXX Sería muy agradable tener un diagrama.

XXX * (import_machinery.rst) ¿qué tal una sección dedicada sólo a los atributos de módulos y paquetes, tal vez ampliando o suplantando las entradas relacionadas en la página de referencia del modelo de datos?

XXX runpy, pkgutil, et al en el manual de la biblioteca deben obtener enlaces «Ver también» en la parte superior que apuntan a la nueva sección del sistema de importación.

XXX Añadir más explicación con respecto a las diferentes formas en que `__main__` se inicializa?

XXX Añadir más información sobre las peculiaridades/trampas `__main__` (es decir, copia de **PEP 395**).

5.10 Referencias

La maquinaria de importación ha evolucionado considerablemente desde los primeros días de Python. La especificación original para paquetes todavía está disponible para leer, aunque algunos detalles han cambiado desde la escritura de ese documento.

La especificación original de `sys.meta_path` era **PEP 302**, con posterior extensión en **PEP 420**.

PEP 420 introdujo *paquetes de espacio de nombres* para Python 3.3. **PEP 420** también introdujo el protocolo `find_loader()` como alternativa a `find_module()`.

PEP 366 describe la adición del atributo `__package__` para las importaciones relativas explícitas en los módulos principales.

PEP 328 introdujo importaciones relativas absolutas y explícitas e inicialmente propuestas `__name__` para la semántica **PEP 366** eventualmente especificaría para `__package__`.

PEP 338 define la ejecución de módulos como scripts.

PEP 451 adds the encapsulation of per-module import state in spec objects. It also off-loads most of the boilerplate responsibilities of loaders back onto the import machinery. These changes allow the deprecation of several APIs in the import system and also addition of new methods to finders and loaders.

Notas al Pie de Pagina

CAPÍTULO 6

Expresiones

Este capítulo explica el significado de los elementos de expresiones en Python.

Notas de Sintaxis: En este y los siguientes capítulos será usada notación BNF extendida para describir sintaxis, no análisis léxico. Cuando (una alternativa de) una regla de sintaxis tiene la forma

```
name  ::=  othername
```

y no han sido dadas semánticas, las semánticas de esta forma de `name` son las mismas que para `othername`.

6.1 Conversiones aritméticas

Cuando una descripción de un operador aritmético a continuación usa la frase «los argumentos numéricos son convertidos a un tipo común», esto significa que la implementación de operador para tipos incorporados funciona de la siguiente forma:

- Si cualquiera de los argumentos es un número complejo, el otro es convertido a complejo;
- de otra forma, si cualquier de los argumentos es un número de punto flotante, el otro es convertido a punto flotante;
- de otra forma, ambos deben ser enteros y no se necesita conversión.

Algunas reglas adicionales aplican para ciertos operadores (ej., una cadena de caracteres como argumento a la izquierda del operador "%"). Las extensiones deben definir su comportamiento de conversión.

6.2 Átomos

Los átomos son los elementos más básicos de las expresiones. Los átomos más simples son identificadores o literales. Las formas encerradas en paréntesis, corchetes o llaves son también sintácticamente categorizadas como átomos. La sintaxis para átomos es:

```
atom      ::=  identifier | literal | enclosure
enclosure ::=  parenth_form | list_display | dict_display | set_display
               | generator_expression | yield_atom
```

6.2.1 Identificadores (Nombres)

Un identificador encontrándose como un átomo es un nombre. Vea la sección *Identificadores y palabras clave* para la definición léxica y la sección *Nombres y vínculos* para documentación de nombrar y vincular.

Cuando el nombre es vinculado a un objeto, la evaluación del átomo produce ese objeto. Cuando un nombre no es vinculado, un intento de evaluarlo genera una excepción `NameError`.

Alteración de nombre privado: Cuando un identificador que ocurre textualmente en una definición de clase comienza con dos o más caracteres de guión bajo y no termina en dos o más guiones bajos, es considerado un *private name* de esa clase. Los nombres privados son transformados a una forma más larga antes de que sea generado código para ellos. La transformación inserta el nombre de clase, con los guiones bajos iniciales eliminados y un solo guión bajo insertado, delante del nombre. Por ejemplo, el identificador `__spam` que se encuentra en una clase denominada `Ham` será transformado a `_Ham__spam`. Esta transformación es independiente del contexto sintáctico en el cual es usado el identificador. Si el nombre transformado es extremadamente largo (más largo que 255 caracteres), puede ocurrir el truncamiento definido por la implementación. Si el nombre de clase consiste únicamente de guiones bajos, no se realiza transformación.

6.2.2 Literales

Python soporta literales de cadenas de caracteres y bytes y varios literales numéricos:

```
literal  ::=  stringliteral | bytesliteral
              | integer | floatnumber | imagnumber
```

La evaluación de un literal produce un objeto del tipo dado (cadena de caracteres, bytes, entero, número de punto flotante, número complejo) con el valor dado. El valor puede ser aproximado en el caso de literales de número de punto flotante e imaginarios (complejos). Vea la sección *Literales* para más detalles.

Todos los literales corresponden a tipos de datos inmutables y, por lo tanto, la identidad del objeto es menos importante que su valor. Múltiples evaluaciones de literales con el mismo valor (ya sea la misma ocurrencia en el texto del programa o una ocurrencia diferente) pueden obtener el mismo objeto o un objeto diferente con el mismo valor.

6.2.3 Formas entre paréntesis

Una forma entre paréntesis es una lista de expresiones opcionales encerradas entre paréntesis:

```
parenth_form  ::=   "(" [starred_expression] ")"
```

Una expresión entre paréntesis produce lo que la lista de expresión produce: si la lista contiene al menos una coma, produce una tupla; en caso contrario, produce la única expresión que que forma la lista de expresiones.

Un par de paréntesis vacío producen un objeto de tupla vacío. Debido a que las tuplas son inmutables, se aplican las mismas reglas que aplican para literales (ej., dos ocurrencias de una tupla vacía puede o no producir el mismo objeto).

Note que las tuplas no son formadas por los paréntesis, sino más bien mediante el uso del operador de coma. La excepción es la tupla vacía, para la cual los paréntesis *son* requeridos – permitir «nada» sin paréntesis en expresiones causaría ambigüedades y permitiría que errores tipográficos comunes pasaran sin ser detectados.

6.2.4 Despliegues para listas, conjuntos y diccionarios

Para construir una lista, un conjunto o un diccionario, Python provee sintaxis especial denominada «despliegue», cada una de ellas en dos sabores:

- los contenidos del contenedor son listados explícitamente o
- son calculados mediante un conjunto de instrucciones de bucle y filtrado, denominadas una *comprehension*.

Los elementos comunes de sintaxis para las comprensiones son:

```
comprehension  ::=   assignment_expression comp_for
comp_for       ::=   ["async"] "for" target_list "in" or_test [comp_iter]
comp_iter      ::=   comp_for | comp_if
comp_if        ::=   "if" expression_nocond [comp_iter]
```

La comprensión consiste en una única expresión seguida por al menos una cláusula `for` y cero o más cláusulas `for` o `if`. En este caso, los elementos del nuevo contenedor son aquellos que serían producidos mediante considerar cada una de las cláusulas `for` o `if` un bloque, anidado de izquierda a derecha y evaluando la expresión para producir un elemento cada vez que se alcanza el bloque más interno.

Sin embargo, aparte de la expresión iterable en la cláusula `for` más a la izquierda, la comprensión es ejecutada en un alcance separado implícitamente anidado. Esto asegura que los nombres asignados a en la lista objetiva no se «filtren» en el alcance adjunto.

La expresión iterable en la cláusula más a la izquierda `for` es evaluada directamente en el alcance anidado y luego pasada como un argumento al alcance implícitamente anidado. Subsecuentes cláusulas `for` y cualquier condición de filtro en la cláusula `for` más a la izquierda no pueden ser evaluadas en el alcance adjunto ya que pueden depender de los valores obtenidos del iterable de más a la izquierda. Por ejemplo, `[x*y for x in range(10) for y in range(x, x+10)]`.

Para asegurar que la comprensión siempre resulta en un contenedor del tipo apropiado, las expresiones `yield` y `yield from` están prohibidas en el alcance implícitamente anidado.

A partir de Python 3.6, en una función *`async def`*, una cláusula `async for` puede ser usada para iterar sobre un *asynchronous iterator*. Una comprensión en una función `async def` puede consistir en una cláusula `for` o `async for` siguiendo la expresión inicial, puede contener cláusulas adicionales `for` o `async for` y también pueden usar expresiones *`await`*. Si una comprensión contiene cláusulas `async for` o expresiones `await` es denominada una

asynchronous comprehension. Una comprensión asincrónica puede suspender la ejecución de la función de corrutina en la cual aparece. Vea también **PEP 530**.

Nuevo en la versión 3.6: Fueron introducidas las comprensiones asincrónicas.

Distinto en la versión 3.8: Prohibidas `yield` y `yield from` en el alcance implícitamente anidado.

6.2.5 Despliegues de lista

Un despliegue de lista es una serie de expresiones posiblemente vacía encerrada entre corchetes:

```
list_display  ::=  "[" [starred_list | comprehension] "]"
```

Un despliegue de lista produce un nuevo objeto lista, el contenido se especifica por una lista de expresiones o una comprensión. Cuando se proporciona una lista de expresiones, sus elementos son evaluados desde la izquierda a la derecha y colocados en el objeto lista en ese orden. Cuando se proporciona una comprensión, la lista es construida desde los elementos resultantes de la comprensión.

6.2.6 Despliegues de conjuntos

Un despliegue de conjunto se denota mediante llaves y se distinguen de los despliegues de diccionarios por la ausencia de caracteres de doble punto separando claves y valores:

```
set_display  ::=  "{" (starred_list | comprehension) "}"
```

Un despliegue de conjunto produce un nuevo objeto conjunto mutable, el contenido se especifica mediante una secuencia de expresiones o una comprensión. Cuando se proporciona una lista de expresiones separadas por comas, sus elementos son evaluados desde la izquierda a la derecha y añadidos al objeto de conjunto. Cuando se proporciona una comprensión, el conjunto es construido de los elementos resultantes de la comprensión.

Un conjunto vacío no puede ser construido con `{}`; este literal construye un diccionario vacío.

6.2.7 Despliegues de diccionario

Un despliegue de diccionario es una serie posiblemente vacía de pares clave/datos encerrados entre llaves:

```
dict_display        ::=  "{" [key_datum_list | dict_comprehension] "}"
key_datum_list      ::=  key_datum ("," key_datum)* [","]
key_datum           ::=  expression ":" expression | "**" or_expr
dict_comprehension  ::=  expression ":" expression comp_for
```

Un despliegue de diccionario produce un nuevo objeto diccionario.

Si es dada una secuencia separada por comas de pares clave/datos, son evaluadas desde la izquierda a la derecha para definir las entradas del diccionario: cada objeto clave es usado como una clave dentro del diccionario para almacenar el dato correspondiente. Esto significa que puedes especificar la misma clave múltiples veces en la lista clave/datos y el valor final del diccionario para esa clave será la última dada.

Un doble asterisco `**` denota *dictionary unpacking*. Su operando debe ser un *mapping*. Cada elemento de mapeo es añadido al nuevo diccionario. Valores más tardíos remplazan los valores ya establecidos para los pares clave/dato y para los desempaquetados de diccionario anteriores.

Nuevo en la versión 3.5: Desempaquetar en despliegues de diccionarios, originalmente propuesto por **PEP 448**.

Una comprensión de diccionario, en contraste a las compresiones de lista y conjunto, necesita dos expresiones separadas con un caracter de doble punto seguido por las cláusulas usuales «for» e «if». Cuando la comprensión se ejecuta, los elementos resultantes clave y valor son insertados en el nuevo diccionario en el orden que son producidos.

Las restricciones de los tipos de los valores de clave son listados anteriormente en la sección *Jerarquía de tipos estándar*. (Para resumir, el tipo de la clave debe ser *hashable*, el cual excluye todos los objetos mutables.) No se detectan choques entre claves duplicadas; el último dato (textualmente el más a la derecha en el despliegue) almacenado para una clave dada prevalece.

Distinto en la versión 3.8: Antes de Python 3.8, en las comprensiones de diccionarios, el orden de evaluación de clave y valor no fue bien definido. En CPython, el valor fue evaluado antes de la clave. A partir de 3.8, la clave es evaluada antes que el valor, como fue propuesto por **PEP 572**.

6.2.8 Expresiones de generador

Una expresión de generador es una notación compacta de generador en paréntesis:

```
generator_expression  ::=  "(" expression comp_for ")"
```

Una expresión de generador produce un nuevo objeto generador. Su sintaxis es la misma que para las comprensiones, excepto que es encerrado en paréntesis en lugar de corchetes o llaves.

Las variables usadas en la expresión de generador son evaluadas perezosamente cuando se ejecuta el método *__next__()* para el objeto generador (de la misma forma que los generadores normales). Sin embargo, la expresión iterable en la cláusula `for` más a la izquierda es inmediatamente evaluada, de forma que un error producido por ella será emitido en el punto en el que se define la expresión de generador, en lugar de en el punto donde se obtiene el primer valor. Subsecuentes cláusulas `for` y cualquier condición en la cláusula `for` más a la izquierda no pueden ser evaluadas en el alcance adjunto, ya que puede depender de los valores obtenidos por el iterable de más a la izquierda. Por ejemplo: `(x*y for x in range(10) for y in range(x, x+10))`.

Los paréntesis pueden ser omitidos en ejecuciones con un solo argumento. Vea la sección *Invocaciones* para más detalles.

Para evitar interferir con la operación esperada de la expresión misma del generador, las expresiones `yield` y `yield from` están prohibidas en el generador definido implícitamente.

Si una expresión de generador contiene cláusulas `async for` o expresiones *await*, se ejecuta una *asynchronous generator expression*. Una expresión de generador asincrónica retorna un nuevo objeto de generador asincrónico, el cual es un iterador asincrónico (ver *Asynchronous Iterators*).

Nuevo en la versión 3.6: Las expresiones de generador asincrónico fueron introducidas.

Distinto en la versión 3.7: Antes de Python 3.7, las expresiones de generador asincrónico podrían aparecer sólo en corrutinas *async def*. Desde 3.7, cualquier función puede usar expresiones de generador asincrónico.

Distinto en la versión 3.8: Prohibidas `yield` y `yield from` en el alcance implícitamente anidado.

6.2.9 Expresiones yield

```
yield_atom       ::=  "(" yield_expression ")"
yield_expression ::=  "yield" [expression_list | "from" expression]
```

La expresión yield se usa al definir una función *generator* o una función *asynchronous generator* y, por lo tanto, solo se puede usar en el cuerpo de una definición de función. Usar una expresión yield en el cuerpo de una función hace que esa función sea un generador y usarla en el cuerpo de una función `async def` hace que la función de corrutina sea un generador asincrónico. Por ejemplo:

```
def gen():  # defines a generator function
    yield 123

async def agen(): # defines an asynchronous generator function
    yield 123
```

Debido a sus efectos secundarios sobre el alcance contenedor, las expresiones `yield` no están permitidas como parte de los alcances implícitamente definidos usados para implementar comprensiones y expresiones de generador.

Distinto en la versión 3.8: Expresiones yield prohibidas en los ámbitos anidados implícitamente utilizados para implementar comprensiones y expresiones de generador.

Las funciones generadoras son descritas a continuación, mientras que las funciones generadoras asincrónicas son descritas separadamente en la sección *Funciones generadoras asincrónicas*.

Cuando una función generadora es invocada, retorna un iterador conocido como un generador. Este generador controla la ejecución de la función generadora. La ejecución empieza cuando uno de los métodos del generador es invocado. En ese momento, la ejecución procede a la primera expresión yield, donde es suspendida de nuevo, retornando el valor de *expression_list* al invocador del generador. Por suspendido, nos referimos a que se retiene todo el estado local, incluyendo los enlaces actuales de variables locales, el puntero de instrucción, la pila de evaluación interna y el estado de cualquier manejo de excepción. Cuando la ejecución se reanuda al invocar uno de los métodos del generador, la función puede proceder como si la expresión yield fuera sólo otra invocación externa. El valor de la expresión yield después de la reanudación depende del método que ha reanudado la ejecución. Si se usa *__next__()* (típicamente mediante un *for* o la función incorporada `next()`) entonces el resultado es `None`. De otra forma, si se usa *send()*, entonces el resultado será el valor pasado a ese método.

Todo este hace a las funciones generadores similar a las corrutinas; producen múltiples veces, tienen más de un punto de entrada y su ejecución puede ser suspendida. La única diferencia es que una función generadora no puede controlar si la ejecución debe continuar después de producir; el control siempre es transferido al invocador del generador.

Las expresiones yield están permitidas en cualquier lugar en un constructo *try*. Si el generador no es reanudado antes de finalizar (alcanzando un recuento de referencia cero o colectando basura), el método generador-iterador *close()* será invocado, permitiendo la ejecución de cualquier cláusula *finally* pendiente.

When `yield from <expr>` is used, the supplied expression must be an iterable. The values produced by iterating that iterable are passed directly to the caller of the current generator's methods. Any values passed in with *send()* and any exceptions passed in with *throw()* are passed to the underlying iterator if it has the appropriate methods. If this is not the case, then *send()* will raise `AttributeError` or `TypeError`, while *throw()* will just raise the passed in exception immediately.

Cuando el iterador subyacente está completo, el atributo `value` de la instancia `StopIteration` generada se convierte en el valor de la expresión yield. Puede ser establecido explícitamente al generar `StopIteration` o automáticamente cuando el subiterador es un generador (retornando un valor del subgenerador).

Distinto en la versión 3.3: Añadido `yield from <expr>` para delegar el control de flujo a un subiterador.

Los paréntesis pueden ser omitidos cuando la expresión yield es la única expresión en el lado derecho de una sentencia de asignación.

Ver también:

PEP 255 - Generadores Simples La propuesta para añadir generadores y la sentencia *yield* a Python.

PEP 342 - Corrutinas mediante Generadores Mejorados La propuesta para mejorar la API y la sintaxis de generadores, haciéndolos utilizables como corrutinas simples.

PEP 380 - Sintaxis para Delegar a un Subgenerador La propuesta para introducir la sintaxis `yield_from`, facilitando la delegación a subgeneradores.

PEP 525- Generadores Asincrónicos La propuesta que expandió **PEP 492** añadiendo capacidades de generador a las funciones corrutina.

Métodos generador-iterador

Esta subsección describe los métodos de un generador iterador. Estos pueden ser usados para controlar la ejecución de una función generadora.

Tenga en cuenta que invocar cualquiera de los métodos de generador siguientes cuando el generador está todavía en ejecución genera una excepción `ValueError`.

`generator.__next__()`
: Comienza la ejecución de una función generadora o la reanuda en la última expresión yield ejecutada. Cuando una función generadora es reanudada con un método *__next__()*, la expresión yield actual siempre evalúa a `None`. La ejecución entonces continúa a la siguiente expresión yield, donde el generador se suspende de nuevo y el valor de *expression_list* se retorna al invocador de *__next__()*. Si el generador termina sin producir otro valor, se genera una excepción `StopIteration`.

  Este método es normalmente invocado implícitamente, por ejemplo, por un bucle *for* o por la función incorporada `next()`.

`generator.send(`*value*`)`
: Reanuda la ejecución y «envía» un valor dentro de la función generadora. El argumento *value* se convierte en el resultado de la expresión yield actual. El método *send()* retorna el siguiente valor producido por el generador o genera `StopIteration` si el generador termina sin producir otro valor. Cuando se ejecuta *send()* para comenzar el generador, debe ser invocado con `None` como el argumento, debido a que no hay expresión yield que pueda recibir el valor.

`generator.throw(`*type*[, *value*[, *traceback*]]`)`
: Genera una excepción de tipo `type` en el punto donde el generador fue pausado y retorna el siguiente valor producido por la función generadora. Si el generador termina sin producir otro valor se genera una excepción `StopIteration`. Si la función generadora no caza la excepción pasada o genera una excepción diferente, entonces se propaga esa excepción al invocador.

`generator.close()`
: Genera `GeneratorExit` en el punto donde la función generadora fue pausada. Si la función generadora termina sin errores, está ya cerrada o genera `GeneratorExit` (sin cazar la excepción), close retorna a su invocador. Si el generador produce un valor, se genera un `RuntimeError`. Si el generador genera cualquier otra excepción, es propagado al invocador. *close()* no hace nada si el generador ya fue terminado debido a una excepción o una salida normal.

Ejemplos

Aquí hay un ejemplo simple que demuestra el comportamiento de generadores y funciones generadoras:

```
>>> def echo(value=None):
...     print("Execution starts when 'next()' is called for the first time.")
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception as e:
...                 value = e
...     finally:
...         print("Don't forget to clean up when 'close()' is called.")
...
>>> generator = echo(1)
>>> print(next(generator))
Execution starts when 'next()' is called for the first time.
1
>>> print(next(generator))
None
>>> print(generator.send(2))
2
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
>>> generator.close()
Don't forget to clean up when 'close()' is called.
```

Para ejemplos usando `yield from`, ver pep-380 en «Qué es nuevo en Python.»

Funciones generadoras asincrónicas

La presencia de una expresión yield en una función o método definido usando *`async def`* adicionalmente define la función como una función *asynchronous generator*.

Cuando se invoca una función generadora asincrónica, retorna un iterador asincrónico conocido como un objeto generador asincrónico. Este objeto entonces controla la ejecución de la función generadora. Un objeto generador asincrónico se usa típicamente en una sentencia *`async for`* en una función corrutina análogamente a como sería usado un objeto generador en una sentencia *`for`*.

Invocar uno de los métodos de un generador asincrónico retorna un objeto *awaitable* y la ejecución comienza cuando este objeto es esperado. En ese momento, la ejecución avanza a la primera expresión yield, donde es suspendida de nuevo, retornando el valor de *`expression_list`* a la corrutina en espera. Como con un generador, la suspensión significa que todo el estado local es retenido, incluyendo los enlaces actuales de variables locales, el puntero de instrucción, la pila de evaluación interna y el estado de cualquier manejo de excepción. Cuando se reanuda la ejecución al espera al siguiente objeto retornado por los métodos del generador asincrónico, la función puede avanzar exactamente igual que si la expresión yield fuera otra invocación externa. El valor de la expresión yield después de la reanudación dependen del método que ha resumido la ejecución. Si se usa *`__anext__()`* entonces el resultado es `None`. De otra forma, si se usa *`asend()`*, entonces el resultado será el valor pasado a ese método.

En una función generadora asincrónica, las expresiones yield están permitidas en cualquier lugar de un constructo *`try`*. Sin embargo, si un generador asincrónico no es reanudado antes de finalizar (alcanzando un contador de referencia cero o recogiendo basura), entonces una expresión yield dentro de un constructo `try` podría fallar al ejecutar cláusulas *`finally`* pendientes. En este caso, es responsabilidad del bucle de eventos o del planificador ejecutando el generador asincrónico invocar el método *`aclose()`* del generador-iterador asincrónico y ejecutar el objeto corrutina resultante, permitiendo así la ejecución de cualquier cláusula `finally` pendiente.

Para encargarse de la finalización, un bucle de eventos debe definir una función *finalizadora* la cual toma un generador-iterador asincrónico y presumiblemente invoca *aclose()* y ejecuta la corrutina. Este *finalizador* puede ser registrado invocando `sys.set_asyncgen_hooks()`. Cuando es iterada por primera vez, un generador-iterador asincrónico almacenará el *finalizador* registrado para ser invocado en la finalización. Para un ejemplo de referencia de un método *finalizador* vea la implementación de `asyncio.Loop.shutdown_asyncgens` en Lib/asyncio/base_events.py.

La expresión `yield from <expr>` es un error de sintaxis cuando es usada en una función generadora asincrónica.

Métodos asincrónicos de generador-iterador

Esta subsección describe los métodos de un generador iterador asincrónico, los cuales son usados para controlar la ejecución de una función generadora.

coroutine agen.**__anext__**()
: Retorna un esperable el cual, cuando corre, comienza a ejecutar el generador asincrónico o lo reanuda en la última expresión yield ejecutada. Cuando se reanuda una función generadora asincrónica con un método *__anext__()*, la expresión yield actual siempre evalúa a `None` en el esperable retornado, el cual cuando corre continuará a la siguiente expresión yield. El valor de *expression_list* de la expresión yield es el valor de la excepción `StopIteration` generada por la corrutina completa. Si el generador asincrónico termina sin producir otro valor, el esperable en su lugar genera una excepción `StopAsyncIteration`, señalando que la iteración asincrónica se ha completado.

  Este método es invocado normalmente de forma implícita por un bucle *async for*.

coroutine agen.**asend**(*value*)
: Retorna un esperable el cual cuando corre reanuda la ejecución del generador asincrónico. Como el método *send()* para un generador, este «envía» un valor a la función generadora asincrónica y el argumento *value* se convierte en el resultado de la expresión yield actual. El esperable retornado por el método *asend()* retornará el siguiente valor producido por el generador como el valor de la `StopIteration` generada o genera `StopAsyncIteration` si el generador asincrónico termina sin producir otro valor. Cuando se invoca *asend()* para empezar el generador asincrónico, debe ser invocado con `None` como argumento, porque no hay expresión yield que pueda recibir el valor.

coroutine agen.**athrow**(*value*)
coroutine agen.**athrow**(*type*[, *value*[, *traceback*]])
: Retorna un esperable que genera una excepción de tipo `type` en el punto donde el generador asincrónico fue pausado y retorna el siguiente valor producido por la función generadora como el valor de la excepción `StopIteration` generada. Si el generador asincrónico termina sin producir otro valor, el esperable genera una excepción `StopAsyncIteration`. Si la función generadora no caza la excepción pasada o genera una excepción diferente, entonces cuando se ejecuta el esperable esa excepción se propaga al invocador del esperable.

coroutine agen.**aclose**()
: Retorna un esperable que cuando corre lanza un `GeneratorExit` a la función generadora asincrónica en el punto donde fue pausada. Si la función generadora asincrónica termina exitosamente, ya está cerrada o genera `GeneratorExit` (sin cazar la excepción), el esperable retornado generará una excepción `StopIteration`. Otros esperables retornados por subsecuentes invocaciones al generador asincrónico generarán una excepción `StopAsyncIteration`. Si el generador asincrónico produce un valor, el esperable genera un `RuntimeError`. Si el generador asincrónico genera cualquier otra excepción, esta es propagada al invocador del esperable. Si el generador asincrónico ha terminado debido a una excepción o una terminación normal, entonces futuras invocaciones a *aclose()* retornarán un esperable que no hace nada.

6.3 Primarios

Los primarios representan las operaciones más fuertemente ligadas al lenguaje. Su sintaxis es:

```
primary  ::=  atom | attributeref | subscription | slicing | call
```

6.3.1 Referencias de atributos

Una referencia de atributo es un primario seguido de un punto y un nombre:

```
attributeref  ::=  primary "." identifier
```

El primario debe evaluar a un objeto de un tipo que soporte referencias de atributos, lo cual la mayoría de los objetos soportan. Luego se le pide a este objeto que produzca el atributo cuyo nombre es el identificador. Esta producción puede ser personalizada sobrescribiendo el método *__getattr__()*. Si este atributo no es esperable, se genera la excepción `AtributeError`. De otra forma, el tipo y el valor del objeto producido es determinado por el objeto. Múltiples evaluaciones la misma referencia de atributo pueden producir diferentes objetos.

6.3.2 Suscripciones

Una subscripción selecciona un elemento de una objeto secuencia (cadena de caracteres, tupla o lista) o mapeo (diccionario):

```
subscription  ::=  primary "[" expression_list "]"
```

El primario debe evaluar a un objeto que soporta subscripción (listas o diccionarios por ejemplo). Los objetos definidos por usuarios pueden soportar subscripción definiendo un método *__getitem__()*.

Para objetos incorporados, hay dos tipos de objetos que soportan subscripción:

Si el primario es un mapeo, la expresión de lista debe evaluar a un objeto cuyo valor es una de las claves del mapeo y la subscripción selecciona el valor en el mapeo que corresponda a esa clave. (La expresión de lista es una tupla excepto si tiene exactamente un elemento.)

Si el primario es una secuencia, la expresión de lista debe evaluar a un entero o a un segmento (como es discutido en la siguiente sección).

La sintaxis formal no hace ninguna provisión especial para índices negativos en secuencias; sin embargo, todas las secuencias incorporadas proveen un método *__getitem__()* que interpreta índices negativos añadiendo al largo de la secuencia al índice (así es que `x[-1]` selecciona el último elemento de `x`). El valor resultante debe ser un entero no negativo menor que el número de elementos en la secuencia y la subscripción selecciona el elemento cuyo índice es ese valor (contando desde cero). Ya que el soporte para índices negativos y el troceado ocurre en el método del objeto *__getitem__()*, las subclases que sobrescriben este método necesitarán añadir explícitamente ese soporte.

Los elementos de una cadena de caracteres son caracteres. Un caracter no es un tipo de datos separado sino una cadena de exactamente un caracter.

6.3.3 Segmentos

Un segmento selecciona un rango de elementos en una objeto secuencia (ej., una cadena de caracteres, tupla o lista). Los segmentos pueden ser usados como expresiones o como objetivos en asignaciones o sentencias `del`. La sintaxis para un segmento:

```
slicing      ::=  primary "[" slice_list "]"
slice_list   ::=  slice_item ("," slice_item)* [","]
slice_item   ::=  expression | proper_slice
proper_slice ::=  [lower_bound] ":" [upper_bound] [ ":" [stride] ]
lower_bound  ::=  expression
upper_bound  ::=  expression
stride       ::=  expression
```

Hay ambigüedad en la sintaxis formal aquí: todo lo que parezca una expresión de lista también parece una segmento de lista, así que cualquier subscripción puede ser interpretada como un segmento. En lugar de complicar aún más la sintaxis, esta es desambiguada definiendo que en este caso la interpretación como una subscripción toma prioridad sobre la interpretación como un segmento (este es el caso si el segmento de lista no contiene un segmento adecuado).

Las semánticas para un segmento son las siguientes. El primario es indexado (usando el mismo método `__getitem__()` de una subscripción normal) con una clave que se construye del segmento de lista, tal como sigue. Si el segmento de lista contiene al menos una coma, la clave es una tupla que contiene la conversión de los elementos del segmento; de otra forma, la conversión del segmento de lista solitario es la clave. La conversión de un elemento de segmento que es una expresión es esa expresión. La conversión de un segmento apropiado es un objeto segmento (ver sección *Jerarquía de tipos estándar*) cuyos atributos `start`, `stop` y `step` son los valores de las expresiones dadas como límite inferior, límite superior y paso, respectivamente, substituyendo `None` para las expresiones faltantes.

6.3.4 Invocaciones

Una invocación invoca un objeto invocable (ej., una *function*) con una serie posiblemente vacía de *argumentos*:

```
call                 ::=  primary "(" [argument_list [","] | comprehension] ")"
argument_list        ::=  positional_arguments ["," starred_and_keywords]
                          ["," keywords_arguments]
                          | starred_and_keywords ["," keywords_arguments]
                          | keywords_arguments
positional_arguments ::=  positional_item ("," positional_item)*
positional_item      ::=  assignment_expression | "*" expression
starred_and_keywords ::=  ("*" expression | keyword_item)
                          ("," "*" expression | "," keyword_item)*
keywords_arguments   ::=  (keyword_item | "**" expression)
                          ("," keyword_item | "," "**" expression)*
keyword_item         ::=  identifier "=" expression
```

Una coma final opcional puede estar presente después de los argumentos posicionales y de palabra clave pero no afecta a las semánticas.

La clave primaria debe evaluar a un objeto invocable (funciones definidas por el usuario, funciones incorporadas, métodos de objetos incorporados, métodos de instancias de clases y todos los objetos que tienen un método `__call__()` son invocables). Todas las expresiones de argumento son evaluadas antes de que la invocación sea intentada. Por favor, refiera a la sección *Definiciones de funciones* para la sintaxis formal de listas de *parameter*.

Si hay argumentos de palabra clave, primero se convierten en argumentos posicionales, como se indica a continuación. En primer lugar, se crea una lista de ranuras sin rellenar para los parámetros formales. Si hay N argumentos posicionales, se colocan en las primeras N ranuras. A continuación, para cada argumento de palabra clave, el identificador se utiliza para determinar la ranura correspondiente (si el identificador es el mismo que el primer nombre de parámetro formal, se utiliza la primera ranura, etc.). Si la ranura ya está llena, se genera una excepción `TypeError`. De lo contrario, el valor del argumento se coloca en la ranura, llenándolo (incluso si la expresión es `None`, esta llena la ranura). Cuando se han procesado todos los argumentos, las ranuras que aún no han sido rellenadas se rellenan con el valor predeterminado correspondiente de la definición de función. (Los valores predeterminados son calculados una vez, cuando se define la función; por lo tanto, un objeto mutable como una lista o diccionario utilizado como valor predeterminado será compartido por todas las llamadas que no especifican un valor de argumento para la ranura correspondiente; esto normalmente debe ser evitado.) Si hay ranuras sin rellenar para las que no se especifica ningún valor predeterminado, se genera una excepción `TypeError`. De lo contrario, la lista de ranuras rellenas se utiliza como la lista de argumentos para la llamada.

CPython implementation detail: Una implementación puede proveer funciones incorporadas cuyos argumentos posicionales no tienen nombres, incluso si son «nombrados» a efectos de documentación y los cuales por consiguiente no pueden ser suplidos por palabras clave. En CPython, este es el caso para funciones implementadas en C que usan `PyArg_ParseTuple()` para analizar sus argumentos.

Si hay más argumentos posicionales que ranuras formales de parámetros, se genera una excepción `TypeError`, a no ser que un parámetro formal usando la sintaxis `*identifier` se encuentre presente; en este caso, ese parámetro formal recibe una tupla conteniendo los argumentos posicionales sobrantes (o una tupla vacía su no hay argumentos posicionales sobrantes).

Si un argumento de palabra clave no corresponde a un nombre de parámetro formal, se genera una excepción `TypeError`, a no ser que un parámetro formal usando la sintaxis `**identifier` esté presente; en este caso, ese parámetro formal recibe un diccionario que contiene los argumentos de palabra clave sobrantes (usando las palabras clave como claves y los valores de argumento como sus valores correspondientes), o un (nuevo) diccionario vacío si no hay argumentos de palabra clave sobrantes.

Si la sintaxis `*expression` aparece en la invocación de función, `expression` debe evaluar a un *iterable*. Elementos de esos iterables son tratados como si fueran argumentos posicionales adicionales. Para la invocación `f(x1, x2, *y, x3, x4)`, si *y* evalúa a una secuencia *y1*, …, *yM*, equivale a una invocación con M+4 argumentos posicionales *x1*, *x2*, *y1*, …, *yM*, *x3*, *x4*.

Una consecuencia de esto es que aunque la sintaxis `*expression` puede aparecer *después* de argumentos de palabra clave explícitos, es procesada *antes* de los argumentos de palabra clave (y cualquiera de los argumentos `*expression` – ver abajo). Así que:

```
>>> def f(a, b):
...     print(a, b)
...
>>> f(b=1, *(2,))
2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

Es inusual usar en la misma invocación tanto argumentos de palabra clave como la sintaxis `*expression`, así que en la práctica no surge esta confusión.

Si la sintaxis `*expression` aparece en la invocación de función, `expression` debe evaluar a un *mapping*, los contenidos del mismo son tratados como argumentos de palabra clave adicionales. Si una palabra clave está ya presente (como un argumento de palabra clave explícito o desde otro desempaquetado), se genera una excepción `TypeError`.

No pueden ser usados parámetros formales usando la sintaxis `*identifier` o `**identifier` como ranuras de argumentos posicionales o como nombres de argumentos de palabra clave.

Distinto en la versión 3.5: Las invocaciones de función aceptan cualquier número de desempaquetados `*` y `**`, los argumentos posicionales pueden seguir a desempaquetados de iterable (`*`) y los argumentos de palabra clave pueden seguir a desempaquetados de diccionario (`*`). Originalmente propuesto por **PEP 448**.

Una invocación siempre retorna algún valor, posiblemente `None`, a no ser que genere una excepción. Cómo se calcula este valor depende del tipo del objeto invocable.

Si es—

una función definida por el usuario: Se ejecuta el bloque de código para la función, pasándole la lista de argumentos. Lo primero que hace el bloque de código es enlazar los parámetros formales a los argumentos; esto es descrito en la sección *Definiciones de funciones*. Cuando el bloque de código ejecuta una sentencia `return`, esto especifica el valor de retorno de la invocación de función.

una función o método incorporado: El resultado depende del intérprete; ver built-in-funcs para las descripciones de funciones y métodos incorporados.

un objeto de clase: Se retorna una nueva instancia de esa clase.

un método de una instancia de clase: Se invoca la función definida por el usuario correspondiente, con una lista de argumentos con un largo uno mayor que la lista de argumentos de la invocación: la instancia se convierte en el primer argumento.

una instancia de clase: La clase debe definir un método `__call__()`; el efecto es entonces el mismo que si ese método fuera invocado.

6.4 Expresión await

Suspende la ejecución de *coroutine* o un objeto *awaitable*. Puede ser usado sólo dentro de una *coroutine function*.

```
await_expr  ::=  "await" primary
```

Nuevo en la versión 3.5.

6.5 El operador de potencia

El operador de potencia se vincula más estrechamente que los operadores unarios a su izquierda; se vincula con menos fuerza que los operadores unarios a su derecha. La sintaxis es:

```
power  ::=  (await_expr | primary) ["**" u_expr]
```

Por lo tanto, en una secuencia sin paréntesis de operadores unarios y de potencia, los operadores son evaluados desde la derecha a la izquierda (este no se constriñe al orden de evaluación para los operandos): `-1**2` resulta en `-1`.

El operador de potencia tiene las mismas semánticas que la función incorporada `pow()` cuando se invoca con dos argumentos: este produce su argumento de la izquierda elevado a la potencia de su argumento de la derecha. Los argumentos numéricos se convierten primero en un tipo común y el resultado es de ese tipo.

Para operandos int, el resultado tiene el mismo tipo que los operandos a no ser que el segundo argumento sea negativo; en ese caso, todos los argumentos son convertidos a float y se entrega un resultado float. Por ejemplo, `10**2` retorna `100`, pero `10**-2` retorna `0.01`.

Elevar `0.0` a una potencia negativa resulta en un `ZeroDivisionError`. Elevar un número negativo a una potencia fraccional resulta en un número `complex`. (En versiones anteriores se genera un `ValueError`.)

6.6 Aritmética unaria y operaciones bit a bit

Toda la aritmética unaria y las operaciones bit a bit tienen la misma prioridad:

```
u_expr  ::=  power | "-" u_expr | "+" u_expr | "~" u_expr
```

El operador unario − (menos) produce la negación de su argumento numérico.

El operador unario + (más) produce su argumento numérico sin cambios.

El operador unario ~ (invertir) produce la inversión bit a bit de su argumento entero. La inversión bit a bit de `x` se define como `-(x+1)`. Sólo aplica a números integrales.

En todos los tres casos, si el argumento no tiene el tipo apropiado, se genera una excepción `TypeError`.

6.7 Operaciones aritméticas binarias

Las operaciones aritméticas binarias tienen los niveles convencionales de prioridad. Tenga en cuenta que algunas de esas operaciones también aplican a ciertos tipos no numéricos. Aparte del operador de potencia, hay sólo dos niveles, uno para operadores multiplicativos y uno para aditivos:

```
m_expr  ::=  u_expr | m_expr "*" u_expr | m_expr "@" m_expr |
             m_expr "//" u_expr | m_expr "/" u_expr |
             m_expr "%" u_expr
a_expr  ::=  m_expr | a_expr "+" m_expr | a_expr "-" m_expr
```

El operador `*` (multiplicación) produce el producto de sus argumentos. Los argumentos pueden ser ambos números, o un argumento debe ser un entero y el otro debe ser una secuencia. En el primer caso, los números se convierten a un tipo común y luego son multiplicados. En el segundo caso, se realiza una repetición de secuencia; un factor de repetición negativo produce una secuencia vacía.

El operador `@` (en) está destinado a ser usado para multiplicación de matrices. Ningún tipo incorporado en Python implementa este operador.

Nuevo en la versión 3.5.

Los operadores `/` (división) y `//` (división de redondeo) producen el cociente de sus argumentos. Los argumentos numéricos son primero convertidos a un tipo común. La división de enteros producen un número de punto flotante, mientras que la división redondeada de enteros resulta en un entero; el resultado es aquel de una división matemática con la función "floor" aplicada al resultado. Dividir entre 0 genera la excepción `ZeroDivisionError`.

El operador `%` (módulo) produce el resto de la división del primer argumento entre el segundo. Los argumentos numéricos son primero convertidos a un tipo común. Un argumento a la derecha cero genera la excepción `ZeroDivisionError`. Los argumentos pueden ser números de punto flotante, ej., `3.14%0.7` es igual a `0.34` (ya que `3.14` es igual a `4*0.7 + 0.34`.) El operador módulo siempre produce un resultado con el mismo signo que su segundo operando (o cero); el valor absoluto del resultado es estrictamente más pequeño que el valor absoluto del segundo operando[1].

[1] Mientras `abs(x%y) < abs(y)` es matemáticamente verdadero, para números de punto flotante puede no ser verdadero numéricamente debido al redondeo. Por ejemplo, y asumiendo una plataforma en la cual un número de punto flotante de Python es un número de doble precisión IEEE 754, a fin de que `-1e-100 % 1e100` tenga el mismo signo que `1e100`, el resultado calculado es `-1e-100 + 1e100`, el cual es numéricamente

Los operadores de división de redondeo y módulo están conectados por la siguiente identidad: `x == (x//y)*y + (x%y)`. La división de redondeo y el módulo también están conectadas por la función incorporada `divmod()`: `divmod(x, y) == (x//y, x%y)`.[2].

Adicionalmente a realizar la operación módulo en números, el operador `%` también está sobrecargado por objetos cadena de caracteres para realizar formateo de cadenas al estilo antiguo (también conocido como interpolación). La sintaxis para el formateo de cadenas está descrita en la Referencia de la Biblioteca de Python, sección old-string-formatting.

El operador de división de redondeo, el operador módulo y la función `divmod()` no están definidas para números complejos. En su lugar, convierta a un número de punto flotante usando la función `abs()` si es apropiado.

El operador + (adición) produce la suma de sus argumentos. Los argumentos deben ser ambos números o ambos secuencias del mismo tipo. En el primer caso, los números son convertidos a un tipo común y luego sumados. En el segundo caso, las secuencias son concatenadas.

El operador − (resta) produce la diferencia de sus argumentos. Los argumentos numéricos son primero convertidos a un tipo común.

6.8 Operaciones de desplazamiento

Las operaciones de desplazamiento tienen menos prioridad que las operaciones aritméticas:

```
shift_expr  ::=  a_expr | shift_expr ("<<" | ">>") a_expr
```

Estos operadores aceptan enteros como argumentos. Ellos desplazan el primer argumento a la izquierda o derecha el número de dígitos dados por el segundo argumento.

Un desplazamiento de *n* bits hacia la derecha está definido como una división de redondeo entre `pow(2,n)`. Un desplazamiento de *n* bits hacia la izquierda está definido como una multiplicación por `pow(2,n)`.

6.9 Operaciones bit a bit binarias

Cada una de las tres operaciones de bits binarias tienen diferente nivel de prioridad:

```
and_expr  ::=  shift_expr | and_expr "&" shift_expr
xor_expr  ::=  and_expr | xor_expr "^" and_expr
or_expr   ::=  xor_expr | or_expr "|" xor_expr
```

El operador `&` produce el AND bit a bit de sus argumentos, los cuales deben ser enteros.

El operador `^` produce el XOR (OR exclusivo) bit a bit de sus argumentos, los cuales deben ser enteros.

El operador `|` produce el OR (inclusivo) bit a bit de sus argumentos, los cuales deben ser enteros.

exactamente igual a `1e100`. La función `math.fmod()` retorna un resultado cuyo signo concuerda con el signo del primer argumento en su lugar, y por ello retorna `-1e-100` en este caso. La aproximación más apropiada depende de su aplicación.

[2] Si x está muy cerca de un entero exacto múltiple de y, es posible para `x//y` que sea uno mayor que `(x-x%y)//y` debido al redondeo. En tales casos, Python retorna el último resultado, a fin de preservar que `divmod(x,y)[0] * y + x % y` sea muy cercano a `x`.

6.10 Comparaciones

A diferencia de C, todas las operaciones de comparación en Python tienen la misma prioridad, la cual es menor que la de cualquier operación aritmética, de desplazamiento o bit a bit. También, a diferencia de C, expresiones como `a < b < c` tienen la interpretación convencional en matemáticas:

```
comparison    ::=  or_expr (comp_operator or_expr)*
comp_operator ::=  "<" | ">" | "==" | ">=" | "<=" | "!="
                   | "is" ["not"] | ["not"] "in"
```

Comparaciones producen valores booleanos: " True" o `False`.

Las comparaciones pueden ser encadenadas arbitrariamente, ej., `x < y <= z` es equivalente a `x < y and y <= z`, excepto que `y` es evaluado sólo una vez (pero en ambos casos `z` no es evaluado para nada cuando `x < y` se encuentra que es falso).

Formalmente, si *a*, *b*, *c*, …, *y*, *z* son expresiones y *op1*, *op2*, …, *opN* son operadores de comparación, entonces `a op1 b op2 c ... y opN z` es equivalente a `a op1 b and b op2 c and ... y opN z`, excepto que cada expresión es evaluada como mucho una vez.

Tenga en cuenta que `a op1 b op2 c` no implica ningún tipo de comparación entre *a* y *c*, por lo que, por ejemplo, `x < y > z` es perfectamente legal (aunque quizás no es bonito).

6.10.1 Comparaciones de valor

Los operadores <, >, ==, >=, <=, y != comparan los valores de dos objetos. Los objetos no necesitan ser del mismo tipo.

El capítulo *Objetos, valores y tipos* afirma que los objetos tienen un valor (en adición al tipo e identidad). El valor de un objeto es una noción bastante abstracta en Python: Por ejemplo, no existe un método de acceso canónico para el valor de un objeto. Además, no se requiere que el valor de un objeto deba ser construido de una forma particular, ej. compuesto de todos sus atributos de datos. Los operadores de comparación implementan una noción particular de lo que es el valor de un objeto. Uno puede pensar en ellos definiendo el valor de un objeto indirectamente, mediante su implementación de comparación.

Debido a que todos los tipos son subtipos (directos o indirectos) de `object`, ellos heredan el comportamiento de comparación predeterminado desde `object`. Los tipos pueden personalizar su comportamiento de comparación implementando *rich comparison methods* como `__lt__()`, descritos en *Basic customization*.

El comportamiento predeterminado para comparación de igualdad (== y !=) se basa en la identidad de los objetos. Por lo tanto, la comparación de instancias con la misma identidad resulta en igualdad, y la comparación de igualdad de instancias con diferentes entidades resulta en desigualdad. Una motivación para este comportamiento predeterminado es el deseo de que todos los objetos sean reflexivos (ej. `x is y` implica `x == y`).

No se provee un orden de comparación por defecto (<, >, <=, and >=); un intento genera `TypeError`. Una motivación para este comportamiento predeterminado es la falta de una invariante similar como para la igualdad.

El comportamiento de la comparación de igualdad predeterminado, que instancias con diferentes identidades siempre son desiguales, puede estar en contraste a que los tipos que necesitarán que tengan una definición sensata de valor de objeto e igualdad basada en el valor. Tales tipos necesitarán personalizar su comportamiento de comparación y, de hecho, un número de tipos incorporados lo han realizado.

La siguiente lista describe el comportamiento de comparación de los tipos incorporados más importantes.

- Números de tipos numéricos incorporadas (typesnumeric) y tipos de la biblioteca estándar `fractions.Fraction` y `decimal.Decimal` pueden ser comparados consigo mismos y entre sus tipos, con la restricción

de que números complejos no soportan orden de comparación. Dentro de los límites de los tipos involucrados, se comparan matemáticamente (algorítmicamente) correctos sin pérdida de precisión.

Los valores no-un-número `float('NaN')` y `decimal.Decimal('NaN')` son especiales. Cualquier comparación ordenada de un número a un no-un-número es falsa. Una implicación contraintuitiva es que los valores no-un-número son son iguales a sí mismos. Por ejemplo, si `x = float('NaN')`, `3 < x`, `x < 3` y `x == x` son todos falso, mientras `x != x` es verdadero. Este comportamiento cumple con IEEE 754.

- `None` y `NotImplemented` son singletons. **PEP 8** avisa que las comparaciones para singletons deben ser realizadas siempre con `is` o `is not`, nunca los operadores de igualdad.
- Las secuencias binarias (instancias de `bytes` o `bytearray`) pueden ser comparadas entre sí y con otros tipos. Ellas comparan lexicográficamente utilizando los valores numéricos de sus elementos.
- Las cadenas de caracteres (instancias de `str`) comparan lexicográficamente usando los puntos de códigos numéricos Unicode (el resultado de la función incorporada `ord()`) o sus caracteres.[3]

  Las cadenas de caracteres y las secuencias binarias no pueden ser comparadas directamente.
- Las secuencias (instancias de `tuple`, `list`, o `range`) pueden ser comparadas sólo entre cada uno de sus tipos, con la restricción de que los rangos no soportan comparación de orden. Comparación de igualdad entre esos tipos resulta en desigualdad y la comparación de orden entre esos tipos genera `TypeError`.

  Las secuencias comparan lexicográficamente usando comparación de sus correspondientes elementos. Los contenedores incorporados asumen que los objetos idénticos son iguales a sí mismos. Eso les permite omitir las pruebas de igualdad para objetos idénticos para mejorar el rendimiento y mantener sus invariantes internos.

  La comparación lexicográfica entre colecciones incorporadas funciona de la siguiente forma:

  - Para que dos colecciones sean comparadas iguales, ellas deben ser del mismo tipo, tener el mismo largo, y cada para de elementos correspondientes deben comparar iguales (por ejemplo, `[1,2] == (1,2)` es falso debido a que el tipo no es el mismo).
  - Las colecciones que soportan comparación de orden son ordenadas igual que sus primeros elementos desiguales (por ejemplo, `[1,2,x] <= [1,2,y]` tiene el mismo valor que `x <= y`). Si un elemento correspondiente no existe, la colección más corta es ordenada primero (por ejemplo, `[1,2] < [1,2,3]` es verdadero).
- Los mapeos (instancias de `dict`) comparan igual si y sólo si tienen pares *(clave, valor)* iguales. La comparación de igualdad de claves y valores refuerza la reflexibilidad.

  Comparaciones de orden (<, >, <=, and >=) generan `TypeError`.
- Conjuntos (instancias de `set` o `frozenset`) pueden ser comparadas entre sí y entre sus tipos.

  Ellas definen operadores de comparación de orden con la intención de comprobar subconjuntos y superconjuntos. Tales relaciones no definen ordenaciones completas (por ejemplo, los dos conjuntos `{1,2}` y `{2,3}` no son iguales, ni subconjuntos ni superconjuntos uno de otro). Acordemente, los conjuntos no son argumentos apropiados para funciones que dependen de ordenación completa (por ejemplo, `min()`, `max()` y `sorted()` producen resultados indefinidos dados una lista de conjuntos como entradas).

  La comparación de conjuntos refuerza la reflexibilidad de sus elementos.

[3] El estándar Unicode distingue entre *code points* (ej. U+0041) y *abstract characters* (ej. «LETRA MAYÚSCULA LATINA A»). Mientras la mayoría de caracteres abstractos en Unicode sólo son representados usando un punto de código, hay un número de caracteres abstractos que pueden adicionalmente ser representados usado una secuencia de más de un punto de código. Por ejemplo, el caracter abstracto «LETRA MAYÚSCULA C LATINA CON CEDILLA» puede ser representado como un único *precomposed character* en la posición de código U+00C7, o como una secuencia de un *base character* en la posición de código U+0043 (LETRA MAYÚSCULA C LATINA), seguida de un *combining character* en la posición de código U+0327 (CEDILLA COMBINADA).

Los operadores de comparación comparan en cadenas de caracteres al nivel de puntos de código Unicode. Esto puede ser contraintuitivo para humanos. Por ejemplo, `"\u00C7" == "\u0043\u0327"` es `False`, incluso aunque ambas cadenas presenten el mismo caracter abstracto «LETRA MAYÚSCULA C LATINA CON CEDILLA».

Para comparar cadenas al nivel de caracteres abstractos (esto es, de una forma intuitiva para humanos), usa `unicodedata.normalize()`.

- La mayoría de los otros tipos incorporados no tienen métodos de comparación implementados, por lo que ellos heredan el comportamiento de comparación predeterminado.

Las clases definidas por el usuario que personalizan su comportamiento de comparación deben seguir algunas reglas de consistencia, si es posible:

- La comparación de igualdad debe ser reflexiva. En otras palabras, los objetos idénticos deben comparar iguales:

  `x is y` implica `x == y`

- La comparación debe ser simétrica. En otras palabras, las siguientes expresiones deben tener el mismo resultado:

  `x == y` y `y == x`

  `x != y` y `y != x`

  `x < y` y `y > x`

  `x <= y` y `y >= x`

- La comparación debe ser transitiva. Los siguientes ejemplos (no exhaustivos) ilustran esto:

  `x > y and y > z` implica `x > z`

  `x < y and y <= z` implica `x < z`

- La comparación inversa debe resultar en la negación booleana. En otras palabras, las siguientes expresiones deben tener el mismo resultado:

  `x == y` y `not x != y`

  `x < y` y `not x >= y` (para ordenación completa)

  `x > y` y `not x <= y` (para ordenación completa)

  Las últimas dos expresiones aplican a colecciones completamente ordenadas (ej. a secuencias, pero no a conjuntos o mapeos). Vea también el decorador `total_ordering()`.

- La función `hash()` debe ser consistente con la igualdad. Los objetos que son iguales deben tener el mismo valor de hash o ser marcados como inhashables.

Python no fuerza a cumplir esas reglas de coherencia. De hecho, los valores no-un-número son u ejemplo para no seguir esas reglas.

6.10.2 Operaciones de prueba de membresía

Los operadores *in* y *not in* comprueban membresía. `x in s` evalúa a `True` si *x* es un miembro de *s* y `False` en caso contrario. `x not in s` retorna la negación de `x in s`. Todas las secuencias incorporadas y tipos conjuntos soportan esto, así como diccionarios, para los cuales `in` comprueba si un diccionario tiene una clave dada. Para tipos contenedores como list, tuple, set, frozenset, dict o collections.deque, la expresión `x in y` es equivalente a `any(x is e or x == e for e in y)`.

Para los tipos cadenas de caracteres y bytes, `x in y` es `True` si y sólo si *x* es una subcadena de *y*. Una comprobación equivalente es `y.find(x) != -1`. Las cadenas de caracteres vacías siempre son consideradas como subcadenas de cualquier otra cadena de caracteres, por lo que `"" in "abc"` retornará `True`.

Para clases definidas por el usuario las cuales definen el método *__contains__()*, `x in y` retorna `True` si `y.__contains__(x)` retorna un valor verdadero y `False` si no.

Para clases definidas por el usuario las cuales no definen *__contains__()* pero definen *__iter__()*, `x in y` es `True` si algún valor `z`, para el cual la expresión `x is z or x == z` es verdadera, es producido iterando sobre `y`. Si una excepción es generada durante la iteración, es como si *in* hubiera generado esa excepción.

Por último, se intenta el protocolo de iteración al estilo antiguo: si una clase define *__getitem__()*, `x in y` es `True` si y sólo si hay un índice entero no negativo *i* tal que `x is y[i] or x == y[i]` y ningún entero menor genera la excepción `IndexError`. (Si cualquier otra excepción es generada, es como si *in* hubiera generado esa excepción).

El operador *not in* es definido para tener el valor de veracidad inverso de *in*.

6.10.3 Comparaciones de identidad

Los operadores *is* y *is not* comprueban la identidad de un objeto. `x is y` es verdadero si y sólo si *x* e *y* son el mismo objeto. La identidad de un Objeto se determina usando la función `id()`. `x is not y` produce el valor de veracidad inverso.[4]

6.11 Operaciones booleanas

```
or_test   ::=   and_test | or_test "or" and_test
and_test  ::=   not_test | and_test "and" not_test
not_test  ::=   comparison | "not" not_test
```

En el contexto de las operaciones booleanas y también cuando sentencias de control de flujo usan expresiones, los siguientes valores se interpretan como falsos: `False`, `None`, ceros numéricos de todos los tipos y cadenas de caracteres y contenedores vacíos (incluyendo cadenas de caracteres, tuplas, diccionarios, conjuntos y conjuntos congelados). Todos los otros valores son interpretados como verdaderos. Los objetos definidos por el usuario pueden personalizar su valor de veracidad proveyendo un método *__bool__()*.

El operador *not* produce `True` si su argumento es falso, `False` si no.

La expresión `x and y` primero evalúa *x*; si *x* es falso, se retorna su valor; de otra forma, *y* es evaluado y se retorna el valor resultante.

La expresión `x or y` primero evalúa *x*; si *x* es verdadero, se retorna su valor; de otra forma, *y* es evaluado y se retorna el valor resultante.

Tenga en cuenta que ni *and* ni *or* restringen el valor y el tipo que retornan a `False` y `True`, sino retornan el último argumento evaluado. Esto es útil a veces, ej., si `s` es una cadena de caracteres que debe ser remplazada por un valor predeterminado si está vacía, la expresión `s or 'foo'` produce el valor deseado. Debido a que *not* tiene que crear un nuevo valor, retorna un valor booleano indiferentemente del tipo de su argumento (por ejemplo, `not 'foo'` produce `False` en lugar de `''`.)

6.12 Expresiones de asignación

```
assignment_expression  ::=   [identifier ":="] expression
```

An assignment expression (sometimes also called a «named expression» or «walrus») assigns an *expression* to an *identifier*, while also returning the value of the *expression*.

One common use case is when handling matched regular expressions:

[4] Debido a la recolección automática de basura, listas libres y a la naturaleza dinámica de los descriptores, puede notar un comportamiento aparentemente inusual en ciertos usos del operador *is*, como aquellos involucrando comparaciones entre métodos de instancia, o constantes. Compruebe su documentación para más información.

```
if matching := pattern.search(data):
    do_something(matching)
```

Or, when processing a file stream in chunks:

```
while chunk := file.read(9000):
    process(chunk)
```

Nuevo en la versión 3.8: Vea **PEP 572** para más detalles sobre las expresiones de asignación.

6.13 Expresiones condicionales

```
conditional_expression  ::=  or_test ["if" or_test "else" expression]
expression              ::=  conditional_expression | lambda_expr
expression_nocond       ::=  or_test | lambda_expr_nocond
```

Las expresiones condicionales (a veces denominadas un «operador ternario») tienen la prioridad más baja que todas las operaciones de Python.

La expresión `x if C else y` primero evalúa la condición, *C* en lugar de *x*. Si *C* es verdadero, *x* es evaluado y se retorna su valor; en caso contrario, *y* es evaluado y se retorna su valor.

Vea **PEP 308** para más detalles sobre expresiones condicionales.

6.14 Lambdas

```
lambda_expr         ::=  "lambda" [parameter_list] ":" expression
lambda_expr_nocond  ::=  "lambda" [parameter_list] ":" expression_nocond
```

Las expresiones lambda (a veces denominadas formas lambda) son usadas para crear funciones anónimas. La expresión `lambda parameters: expression` produce un objeto de función. El objeto sin nombre se comporta como un objeto función con:

```
def <lambda>(parameters):
    return expression
```

Vea la sección *Definiciones de funciones* para la sintaxis de listas de parámetros. Tenga en cuenta que las funciones creadas con expresiones lambda no pueden contener sentencias ni anotaciones.

6.15 Listas de expresiones

```
expression_list      ::=  expression ("," expression)* [","]
starred_list         ::=  starred_item ("," starred_item)* [","]
starred_expression   ::=  expression | (starred_item ",")* [starred_item]
starred_item         ::=  assignment_expression | "*" or_expr
```

Excepto cuando son parte de un despliegue de lista o conjunto, una lista de expresión conteniendo al menos una coma produce una tupla. El largo de la tupla es el número de expresiones en la lista. Las expresiones son evaluadas de izquierda a derecha.

Un asterisco `*` denota *iterable unpacking*. Su operando deben ser un *iterable*. El iterable es expandido en una secuencia de elementos, los cuales son incluidos en la nueva tupla, lista o conjunto en el lugar del desempaquetado.

Nuevo en la versión 3.5: Desempaquetado iterable en listas de expresiones, originalmente propuesto por **PEP 488**.

La coma final sólo es requerida para crear una tupla única (también denominada un *singleton*); es opcional en todos los otros casos. Una única expresión sin una coma final no crea una tupla, si no produce el valor de esa expresion. (Para crear una tupla vacía, usa un par de paréntesis vacío: `()`.)

6.16 Orden de evaluación

Python evalúa las expresiones de izquierda a derecha. Note que mientras se evalúa una asignación, la parte derecha es evaluada antes que la parte izquierda.

En las siguientes líneas, las expresiones serán evaluadas en el orden aritmético de sus sufijos:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
expr1(expr2, expr3, *expr4, **expr5)
expr3, expr4 = expr1, expr2
```

6.17 Prioridad de operador

La siguiente tabla resume la prioridad de operador en Python, desde la prioridad más baja (menos vinculante) a la prioridad más alta (más vinculante). Operadores en la misma caja tienen la misma prioridad. A no ser que la sintaxis sea dada explícitamente, los operadores son binarios. Los operadores en la misma caja, de izquierda a derecha (excepto para exponenciación, cuyos grupos de derecha a izquierda).

Tenga en cuenta que las comparaciones, comprobaciones de membresía y las comprobaciones de identidad tienen la misma prioridad y una característica de encadenado de izquierda a derecha como son descritas en la sección *Comparaciones*.

Operador	Descripción
`:=`	Expresión de asignación
lambda	Expresión lambda
if – `else`	Expresión condicional
or	Booleano OR
and	Booleano AND
not `x`	Booleano NOT
in, *not in*, *is*, *is not*, <, <=, >, >=, !=, ==	Comparaciones, incluyendo comprobaciones de membresía y de identidad
`|`	OR bit a bit
`^`	XOR bit a bit
`&`	AND bit a bit
<<, >>	Desplazamientos
+, –	Adición y sustracción
*, `@`, /, //, %	Multiplicación, multiplicación de matrices, división, división de redondeo, resto[5]
+x, –x, ~x	NOT positivo, negativo, bit a bit
**	Exponenciación[6]
await `x`	Expresión await
`x[index]`, `x[index:index]`, `x(arguments...)`, `x.attribute`	Subscripción, segmentación, invocación, referencia de atributo
`(expressions...)`, `[expressions...]`, `{key: value...}`, `{expressions...}`	Expresión de enlace o entre paréntesis, despliegues de lista, diccionario y conjunto

Notas al pie

[5] El operador `%` también es usado para formateo de cadenas; aplica la misma prioridad.

[6] El operador de potencia `**` vincula con menos fuerza que un operador unario aritmético uno bit a bit en su derecha, esto significa que `2**-1` is `0.5`.

CAPÍTULO 7

Simple statements

A simple statement is comprised within a single logical line. Several simple statements may occur on a single line separated by semicolons. The syntax for simple statements is:

```
simple_stmt  ::=  expression_stmt
                | assert_stmt
                | assignment_stmt
                | augmented_assignment_stmt
                | annotated_assignment_stmt
                | pass_stmt
                | del_stmt
                | return_stmt
                | yield_stmt
                | raise_stmt
                | break_stmt
                | continue_stmt
                | import_stmt
                | future_stmt
                | global_stmt
                | nonlocal_stmt
```

7.1 Expression statements

Expression statements are used (mostly interactively) to compute and write a value, or (usually) to call a procedure (a function that returns no meaningful result; in Python, procedures return the value `None`). Other uses of expression statements are allowed and occasionally useful. The syntax for an expression statement is:

```
expression_stmt  ::=  starred_expression
```

An expression statement evaluates the expression list (which may be a single expression).

In interactive mode, if the value is not `None`, it is converted to a string using the built-in `repr()` function and the resulting string is written to standard output on a line by itself (except if the result is `None`, so that procedure calls do not cause any output.)

7.2 Assignment statements

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

```
assignment_stmt  ::=  (target_list "=")+ (starred_expression | yield_expression)
target_list      ::=  target ("," target)* [","]
target           ::=  identifier
                      | "(" [target_list] ")"
                      | "[" [target_list] "]"
                      | attributeref
                      | subscription
                      | slicing
                      | "*" target
```

(See section *Primarios* for the syntax definitions for *attributeref*, *subscription*, and *slicing*.)

An assignment statement evaluates the expression list (remember that this can be a single expression or a comma-separated list, the latter yielding a tuple) and assigns the single resulting object to each of the target lists, from left to right.

Assignment is defined recursively depending on the form of the target (list). When a target is part of a mutable object (an attribute reference, subscription or slicing), the mutable object must ultimately perform the assignment and decide about its validity, and may raise an exception if the assignment is unacceptable. The rules observed by various types and the exceptions raised are given with the definition of the object types (see section *Jerarquía de tipos estándar*).

Assignment of an object to a target list, optionally enclosed in parentheses or square brackets, is recursively defined as follows.

- If the target list is a single target with no trailing comma, optionally in parentheses, the object is assigned to that target.
- Else: The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets.
  - If the target list contains one target prefixed with an asterisk, called a «starred» target: The object must be an iterable with at least as many items as there are targets in the target list, minus one. The first items of the iterable are assigned, from left to right, to the targets before the starred target. The final items of the iterable are assigned to the targets after the starred target. A list of the remaining items in the iterable is then assigned to the starred target (the list can be empty).
  - Else: The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets.

Assignment of an object to a single target is recursively defined as follows.

- If the target is an identifier (name):
  - If the name does not occur in a *global* or *nonlocal* statement in the current code block: the name is bound to the object in the current local namespace.
  - Otherwise: the name is bound to the object in the global namespace or the outer namespace determined by *nonlocal*, respectively.

The name is rebound if it was already bound. This may cause the reference count for the object previously bound to the name to reach zero, causing the object to be deallocated and its destructor (if it has one) to be called.

- If the target is an attribute reference: The primary expression in the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, `TypeError` is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily `AttributeError`).

  Note: If the object is a class instance and the attribute reference occurs on both sides of the assignment operator, the right-hand side expression, `a.x` can access either an instance attribute or (if no instance attribute exists) a class attribute. The left-hand side target `a.x` is always set as an instance attribute, creating it if necessary. Thus, the two occurrences of `a.x` do not necessarily refer to the same attribute: if the right-hand side expression refers to a class attribute, the left-hand side creates a new instance attribute as the target of the assignment:

```
class Cls:
    x = 3             # class variable
inst = Cls()
inst.x = inst.x + 1   # writes inst.x as 4 leaving Cls.x as 3
```

  This description does not necessarily apply to descriptor attributes, such as properties created with `property()`.

- If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence object (such as a list) or a mapping object (such as a dictionary). Next, the subscript expression is evaluated.

  If the primary is a mutable sequence object (such as a list), the subscript must yield an integer. If it is negative, the sequence's length is added to it. The resulting value must be a nonnegative integer less than the sequence's length, and the sequence is asked to assign the assigned object to its item with that index. If the index is out of range, `IndexError` is raised (assignment to a subscripted sequence cannot add new items to a list).

  If the primary is a mapping object (such as a dictionary), the subscript must have a type compatible with the mapping's key type, and the mapping is then asked to create a key/datum pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

  For user-defined objects, the *`__setitem__()`* method is called with appropriate arguments.

- If the target is a slicing: The primary expression in the reference is evaluated. It should yield a mutable sequence object (such as a list). The assigned object should be a sequence object of the same type. Next, the lower and upper bound expressions are evaluated, insofar they are present; defaults are zero and the sequence's length. The bounds should evaluate to integers. If either bound is negative, the sequence's length is added to it. The resulting bounds are clipped to lie between zero and the sequence's length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the target sequence allows it.

CPython implementation detail: In the current implementation, the syntax for targets is taken to be the same as for expressions, and invalid syntax is rejected during the code generation phase, causing less detailed error messages.

Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are "simultaneous" (for example `a, b = b, a` swaps two variables), overlaps *within* the collection of assigned-to variables occur left-to-right, sometimes resulting in confusion. For instance, the following program prints `[0, 2]`:

```
x = [0, 1]
i = 0
i, x[i] = 1, 2          # i is updated, then x[i] is updated
print(x)
```

Ver también:

PEP 3132 - Extended Iterable Unpacking The specification for the `*target` feature.

7.2.1 Augmented assignment statements

Augmented assignment is the combination, in a single statement, of a binary operation and an assignment statement:

```
augmented_assignment_stmt  ::=  augtarget augop (expression_list | yield_expression)
augtarget                  ::=  identifier | attributeref | subscription | slicing
augop                      ::=  "+=" | "-=" | "*=" | "@=" | "/=" | "//=" | "%=" | "**="
                                | ">>=" | "<<=" | "&=" | "^=" | "|="
```

(See section *Primarios* for the syntax definitions of the last three symbols.)

An augmented assignment evaluates the target (which, unlike normal assignment statements, cannot be an unpacking) and the expression list, performs the binary operation specific to the type of assignment on the two operands, and assigns the result to the original target. The target is only evaluated once.

An augmented assignment expression like `x += 1` can be rewritten as `x = x + 1` to achieve a similar, but not exactly equal effect. In the augmented version, `x` is only evaluated once. Also, when possible, the actual operation is performed *in-place*, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

Unlike normal assignments, augmented assignments evaluate the left-hand side *before* evaluating the right-hand side. For example, `a[i] += f(x)` first looks-up `a[i]`, then it evaluates `f(x)` and performs the addition, and lastly, it writes the result back to `a[i]`.

With the exception of assigning to tuples and multiple targets in a single statement, the assignment done by augmented assignment statements is handled the same way as normal assignments. Similarly, with the exception of the possible *in-place* behavior, the binary operation performed by augmented assignment is the same as the normal binary operations.

For targets which are attribute references, the same *caveat about class and instance attributes* applies as for regular assignments.

7.2.2 Annotated assignment statements

Annotation assignment is the combination, in a single statement, of a variable or attribute annotation and an optional assignment statement:

```
annotated_assignment_stmt  ::=  augtarget ":" expression
                                ["=" (starred_expression | yield_expression)]
```

The difference from normal *Assignment statements* is that only single target is allowed.

For simple names as assignment targets, if in class or module scope, the annotations are evaluated and stored in a special class or module attribute `__annotations__` that is a dictionary mapping from variable names (mangled if private) to evaluated annotations. This attribute is writable and is automatically created at the start of class or module body execution, if annotations are found statically.

For expressions as assignment targets, the annotations are evaluated if in class or module scope, but not stored.

If a name is annotated in a function scope, then this name is local for that scope. Annotations are never evaluated and stored in function scopes.

If the right hand side is present, an annotated assignment performs the actual assignment before evaluating annotations (where applicable). If the right hand side is not present for an expression target, then the interpreter evaluates the target except for the last *__setitem__()* or *__setattr__()* call.

Ver también:

PEP 526 - Syntax for Variable Annotations The proposal that added syntax for annotating the types of variables (including class variables and instance variables), instead of expressing them through comments.

PEP 484 - Type hints The proposal that added the `typing` module to provide a standard syntax for type annotations that can be used in static analysis tools and IDEs.

Distinto en la versión 3.8: Now annotated assignments allow same expressions in the right hand side as the regular assignments. Previously, some expressions (like un-parenthesized tuple expressions) caused a syntax error.

7.3 The `assert` statement

Assert statements are a convenient way to insert debugging assertions into a program:

```
assert_stmt  ::=  "assert" expression ["," expression]
```

The simple form, `assert expression`, is equivalent to

```
if __debug__:
    if not expression: raise AssertionError
```

The extended form, `assert expression1, expression2`, is equivalent to

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

These equivalences assume that `__debug__` and `AssertionError` refer to the built-in variables with those names. In the current implementation, the built-in variable `__debug__` is `True` under normal circumstances, `False` when optimization is requested (command line option `-O`). The current code generator emits no code for an assert statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

Assignments to `__debug__` are illegal. The value for the built-in variable is determined when the interpreter starts.

7.4 The `pass` statement

```
pass_stmt  ::=  "pass"
```

`pass` is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```

7.5 The `del` statement

```
del_stmt  ::=  "del" target_list
```

Deletion is recursively defined very similar to the way assignment is defined. Rather than spelling it out in full details, here are some hints.

Deletion of a target list recursively deletes each target, from left to right.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a *global* statement in the same code block. If the name is unbound, a `NameError` exception will be raised.

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).

Distinto en la versión 3.2: Previously it was illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

7.6 The `return` statement

```
return_stmt  ::=  "return" [expression_list]
```

return may only occur syntactically nested in a function definition, not within a nested class definition.

If an expression list is present, it is evaluated, else `None` is substituted.

return leaves the current function call with the expression list (or `None`) as return value.

When *return* passes control out of a *try* statement with a *finally* clause, that `finally` clause is executed before really leaving the function.

In a generator function, the *return* statement indicates that the generator is done and will cause `StopIteration` to be raised. The returned value (if any) is used as an argument to construct `StopIteration` and becomes the `StopIteration.value` attribute.

In an asynchronous generator function, an empty *return* statement indicates that the asynchronous generator is done and will cause `StopAsyncIteration` to be raised. A non-empty `return` statement is a syntax error in an asynchronous generator function.

7.7 The `yield` statement

```
yield_stmt  ::=  yield_expression
```

A *yield* statement is semantically equivalent to a *yield expression*. The yield statement can be used to omit the parentheses that would otherwise be required in the equivalent yield expression statement. For example, the yield statements

```
yield <expr>
yield from <expr>
```

are equivalent to the yield expression statements

```
(yield <expr>)
(yield from <expr>)
```

Yield expressions and statements are only used when defining a *generator* function, and are only used in the body of the generator function. Using yield in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

For full details of `yield` semantics, refer to the *Expresiones yield* section.

7.8 The `raise` statement

```
raise_stmt  ::=  "raise" [expression ["from" expression]]
```

If no expressions are present, `raise` re-raises the last exception that was active in the current scope. If no exception is active in the current scope, a `RuntimeError` exception is raised indicating that this is an error.

Otherwise, `raise` evaluates the first expression as the exception object. It must be either a subclass or an instance of `BaseException`. If it is a class, the exception instance will be obtained when needed by instantiating the class with no arguments.

The *type* of the exception is the exception instance's class, the *value* is the instance itself.

A traceback object is normally created automatically when an exception is raised and attached to it as the `__traceback__` attribute, which is writable. You can create an exception and set your own traceback in one step using the `with_traceback()` exception method (which returns the same exception instance, with its traceback set to its argument), like so:

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

The `from` clause is used for exception chaining: if given, the second *expression* must be another exception class or instance. If the second expression is an exception instance, it will be attached to the raised exception as the `__cause__` attribute (which is writable). If the expression is an exception class, the class will be instantiated and the resulting exception instance will be attached to the raised exception as the `__cause__` attribute. If the raised exception is not handled, both exceptions will be printed:

```
>>> try:
...     print(1 / 0)
... except Exception as exc:
...     raise RuntimeError("Something bad happened") from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

A similar mechanism works implicitly if an exception is raised inside an exception handler or a `finally` clause: the previous exception is then attached as the new exception's `__context__` attribute:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Exception chaining can be explicitly suppressed by specifying `None` in the `from` clause:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened") from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Additional information on exceptions can be found in section *Excepciones*, and information about handling exceptions is in section *La sentencia try*.

Distinto en la versión 3.3: `None` is now permitted as `Y` in `raise X from Y`.

Nuevo en la versión 3.3: The `__suppress_context__` attribute to suppress automatic display of the exception context.

7.9 The `break` statement

```
break_stmt  ::=  "break"
```

break may only occur syntactically nested in a *for* or *while* loop, but not nested in a function or class definition within that loop.

It terminates the nearest enclosing loop, skipping the optional `else` clause if the loop has one.

If a *for* loop is terminated by *break*, the loop control target keeps its current value.

When *break* passes control out of a *try* statement with a *finally* clause, that `finally` clause is executed before really leaving the loop.

7.10 The `continue` statement

```
continue_stmt   ::=   "continue"
```

continue may only occur syntactically nested in a *for* or *while* loop, but not nested in a function or class definition within that loop. It continues with the next cycle of the nearest enclosing loop.

When *continue* passes control out of a *try* statement with a *finally* clause, that `finally` clause is executed before really starting the next loop cycle.

7.11 The `import` statement

```
import_stmt      ::=   "import" module ["as" identifier] ("," module ["as" identifier])*
                       | "from" relative_module "import" identifier ["as" identifier]
                       ("," identifier ["as" identifier])*
                       | "from" relative_module "import" "(" identifier ["as" identifier]
                       ("," identifier ["as" identifier])* [","] ")"
                       | "from" module "import" "*"
module           ::=   (identifier ".")* identifier
relative_module  ::=   "."* module | "."+
```

The basic import statement (no *from* clause) is executed in two steps:

1. find a module, loading and initializing it if necessary
2. define a name or names in the local namespace for the scope where the *import* statement occurs.

When the statement contains multiple clauses (separated by commas) the two steps are carried out separately for each clause, just as though the clauses had been separated out into individual import statements.

The details of the first step, finding and loading modules are described in greater detail in the section on the *import system*, which also describes the various types of packages and modules that can be imported, as well as all the hooks that can be used to customize the import system. Note that failures in this step may indicate either that the module could not be located, *or* that an error occurred while initializing the module, which includes execution of the module's code.

If the requested module is retrieved successfully, it will be made available in the local namespace in one of three ways:

- If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.
- If no other name is specified, and the module being imported is a top level module, the module's name is bound in the local namespace as a reference to the imported module
- If the module being imported is *not* a top level module, then the name of the top level package that contains the module is bound in the local namespace as a reference to the top level package. The imported module must be accessed using its full qualified name rather than directly

The *from* form uses a slightly more complex process:

1. find the module specified in the *from* clause, loading and initializing it if necessary;
2. for each of the identifiers specified in the *import* clauses:
    1. check if the imported module has an attribute by that name
    2. if not, attempt to import a submodule with that name and then check the imported module again for that attribute

3. if the attribute is not found, `ImportError` is raised.
4. otherwise, a reference to that value is stored in the local namespace, using the name in the `as` clause if it is present, otherwise using the attribute name

Examples:

```
import foo                 # foo imported and bound locally
import foo.bar.baz         # foo.bar.baz imported, foo bound locally
import foo.bar.baz as fbb  # foo.bar.baz imported and bound as fbb
from foo.bar import baz    # foo.bar.baz imported and bound as baz
from foo import attr       # foo imported and foo.attr bound as attr
```

If the list of identifiers is replaced by a star (`'*'`), all public names defined in the module are bound in the local namespace for the scope where the *import* statement occurs.

The *public names* defined by a module are determined by checking the module's namespace for a variable named `__all__`; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in `__all__` are all considered public and are required to exist. If `__all__` is not defined, the set of public names includes all names found in the module's namespace which do not begin with an underscore character (`'_'`). `__all__` should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

The wild card form of import — `from module import *` — is only allowed at the module level. Attempting to use it in class or function definitions will raise a `SyntaxError`.

When specifying what module to import you do not have to specify the absolute name of the module. When a module or package is contained within another package it is possible to make a relative import within the same top package without having to mention the package name. By using leading dots in the specified module or package after *from* you can specify how high to traverse up the current package hierarchy without specifying exact names. One leading dot means the current package where the module making the import exists. Two dots means up one package level. Three dots is up two levels, etc. So if you execute `from . import mod` from a module in the `pkg` package then you will end up importing `pkg.mod`. If you execute `from ..subpkg2 import mod` from within `pkg.subpkg1` you will import `pkg.subpkg2.mod`. The specification for relative imports is contained in the *Paquete Importaciones relativas* section.

`importlib.import_module()` is provided to support applications that determine dynamically the modules to be loaded.

Raises an auditing event `import` with arguments `module`, `filename`, `sys.path`, `sys.meta_path`, `sys.path_hooks`.

7.11.1 Future statements

A *future statement* is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python where the feature becomes standard.

The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

```
future_stmt   ::=   "from" "__future__" "import" feature ["as" identifier]
                    ("," feature ["as" identifier])*
                    | "from" "__future__" "import" "(" feature ["as" identifier]
                    ("," feature ["as" identifier])* [","] ")"
feature       ::=   identifier
```

A future statement must appear near the top of the module. The only lines that can appear before a future statement are:

- the module docstring (if any),
- comments,
- blank lines, and
- other future statements.

The only feature that requires using the future statement is `annotations` (see **PEP 563**).

All historical features enabled by the future statement are still recognized by Python 3. The list includes `absolute_import`, `division`, `generators`, `generator_stop`, `unicode_literals`, `print_function`, `nested_scopes` and `with_statement`. They are all redundant because they are always enabled, and only kept for backwards compatibility.

A future statement is recognized and treated specially at compile time: Changes to the semantics of core constructs are often implemented by generating different code. It may even be the case that a new feature introduces new incompatible syntax (such as a new reserved word), in which case the compiler may need to parse the module differently. Such decisions cannot be pushed off until runtime.

For any given release, the compiler knows which feature names have been defined, and raises a compile-time error if a future statement contains a feature not known to it.

The direct runtime semantics are the same as for any import statement: there is a standard module `__future__`, described later, and it will be imported in the usual way at the time the future statement is executed.

The interesting runtime semantics depend on the specific feature enabled by the future statement.

Note that there is nothing special about the statement:

```
import __future__ [as name]
```

That is not a future statement; it's an ordinary import statement with no special semantics or syntax restrictions.

Code compiled by calls to the built-in functions `exec()` and `compile()` that occur in a module `M` containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can be controlled by optional arguments to `compile()` — see the documentation of that function for details.

A future statement typed at an interactive interpreter prompt will take effect for the rest of the interpreter session. If an interpreter is started with the `-i` option, is passed a script name to execute, and the script includes a future statement, it will be in effect in the interactive session started after the script is executed.

Ver también:

PEP 236 - Back to the __future__ The original proposal for the __future__ mechanism.

7.12 The `global` statement

```
global_stmt  ::=  "global" identifier ("," identifier)*
```

The *global* statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as globals. It would be impossible to assign to a global variable without `global`, although free variables may refer to globals without being declared global.

Names listed in a *global* statement must not be used in the same code block textually preceding that `global` statement.

Names listed in a *global* statement must not be defined as formal parameters or in a *for* loop control target, *class* definition, function definition, *import* statement, or variable annotation.

CPython implementation detail: The current implementation does not enforce some of these restrictions, but programs should not abuse this freedom, as future implementations may enforce them or silently change the meaning of the program.

Programmer's note: *global* is a directive to the parser. It applies only to code parsed at the same time as the `global` statement. In particular, a `global` statement contained in a string or code object supplied to the built-in `exec()` function does not affect the code block *containing* the function call, and code contained in such a string is unaffected by `global` statements in the code containing the function call. The same applies to the `eval()` and `compile()` functions.

7.13 The `nonlocal` statement

```
nonlocal_stmt  ::=  "nonlocal" identifier ("," identifier)*
```

The *nonlocal* statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals. This is important because the default behavior for binding is to search the local namespace first. The statement allows encapsulated code to rebind variables outside of the local scope besides the global (module) scope.

Names listed in a *nonlocal* statement, unlike those listed in a *global* statement, must refer to pre-existing bindings in an enclosing scope (the scope in which a new binding should be created cannot be determined unambiguously).

Names listed in a *nonlocal* statement must not collide with pre-existing bindings in the local scope.

Ver también:

PEP 3104 - Access to Names in Outer Scopes The specification for the *nonlocal* statement.

CAPÍTULO 8

Sentencias compuestas

Las sentencias compuestas contienen (grupos de) otras sentencias; estas afectan o controlan la ejecución de esas otras sentencias de alguna manera. En general, las sentencias compuestas abarcan varias líneas, aunque en representaciones simples una sentencia compuesta completa puede estar contenida en una línea.

Las sentencias *if*, *while* y *for* implementan construcciones de control de flujo tradicionales. *try* especifica gestores de excepción o código de limpieza para un grupo de sentencias, mientras que las sentencias *with* permite la ejecución del código de inicialización y finalización alrededor de un bloque de código. Las definiciones de función y clase también son sentencias sintácticamente compuestas.

Una sentencia compuesta consta de una o más "cláusulas". Una cláusula consta de un encabezado y una "suite". Los encabezados de cláusula de una declaración compuesta particular están todos en el mismo nivel de indentación. Cada encabezado de cláusula comienza con una palabra clave de identificación única y termina con dos puntos. Una suite es un grupo de sentencias controladas por una cláusula. Una suite puede ser una o más sentencias simples separadas por punto y coma en la misma línea como el encabezado, siguiendo los dos puntos del encabezado, o puede ser una o puede ser una o más declaraciones indentadas en líneas posteriores. Solo la última forma de una suite puede contener sentencias compuestas anidadas; lo siguiente es ilegal, principalmente porque no estaría claro a qué cláusula *if* seguido de la cláusula *else* hace referencia:

```
if test1: if test2: print(x)
```

También tenga en cuenta que el punto y coma se une más apretado que los dos puntos en este contexto, de modo que en el siguiente ejemplo, todas o ninguna de las llamadas `print()` se ejecutan:

```
if x < y < z: print(x); print(y); print(z)
```

Resumiendo:

```
compound_stmt   ::=   if_stmt
                    | while_stmt
                    | for_stmt
                    | try_stmt
                    | with_stmt
```

```
                    | funcdef
                    | classdef
                    | async_with_stmt
                    | async_for_stmt
                    | async_funcdef
suite           ::=  stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement       ::=  stmt_list NEWLINE | compound_stmt
stmt_list       ::=  simple_stmt (";" simple_stmt)* [";"]
```

Tenga en cuenta que las sentencias siempre terminan en un `NEWLINE` posiblemente seguida de `DEDENT`. También tenga en cuenta que las cláusulas de continuación opcionales siempre comienzan con una palabra clave que no puede iniciar una sentencia, por lo tanto, no hay ambigüedades (el problema de "colgado *if*" se resuelve en Python al requerir que las sentencias anidadas *if* deben estar indentadas).

El formato de las reglas gramaticales en las siguientes secciones coloca cada cláusula en una línea separada para mayor claridad.

8.1 La sentencia `if`

La sentencia *if* se usa para la ejecución condicional:

```
if_stmt  ::=  "if" assignment_expression ":" suite
              ("elif" assignment_expression ":" suite)*
              ["else" ":" suite]
```

Selecciona exactamente una de las suites evaluando las expresiones una por una hasta que se encuentre una verdadera (vea la sección *Operaciones booleanas* para la definición de verdadero y falso); entonces esa suite se ejecuta (y ninguna otra parte de la sentencia *if* se ejecuta o evalúa). Si todas las expresiones son falsas, se ejecuta la suite de cláusulas *else*, si está presente.

8.2 La sentencia `while`

La sentencia *while* se usa para la ejecución repetida siempre que una expresión sea verdadera:

```
while_stmt  ::=  "while" assignment_expression ":" suite
                 ["else" ":" suite]
```

Esto prueba repetidamente la expresión y, si es verdadera, ejecuta la primera suite; si la expresión es falsa (que puede ser la primera vez que se prueba), se ejecuta el conjunto de cláusulas `else`, si está presente, y el bucle termina.

La sentencia *break* ejecutada en la primer suite termina el bucle sin ejecutar la suite de cláusulas `else`. La sentencia *continue* ejecutada en la primera suite omite el resto de la suite y vuelve a probar la expresión.

8.3 La sentencia `for`

La sentencia *for* se usa para iterar sobre los elementos de una secuencia (como una cadena de caracteres, tupla o lista) u otro objeto iterable:

```
for_stmt  ::=  "for" target_list "in" expression_list ":" suite
               ["else" ":" suite]
```

La lista de expresiones se evalúa una vez; debería producir un objeto iterable. Se crea un iterador para el resultado de la `expression_list`. La suite se ejecuta una vez para cada elemento proporcionado por el iterador, en el orden retornado por el iterador. Cada elemento a su vez se asigna a la lista utilizando las reglas estándar para las asignaciones (ver *Assignment statements*), y luego se ejecuta la suite. Cuando los elementos están agotados (que es inmediatamente cuando la secuencia está vacía o un iterador genera una excepción del tipo `StopIteration`), la suite en la cláusula `else`, si está presente, se ejecuta y el bucle termina.

La sentencia *break* ejecutada en la primera suite termina el bucle sin ejecutar el conjunto de cláusulas `else`. La sentencia *continue* ejecutada en la primera suite omite el resto de las cláusulas y continúa con el siguiente elemento, o con la cláusula `else` si no hay un elemento siguiente.

El bucle `for` realiza asignaciones a las variables en la lista. Esto sobrescribe todas las asignaciones anteriores a esas variables, incluidas las realizadas en la suite del bucle `for`:

```
for i in range(10):
    print(i)
    i = 5             # this will not affect the for-loop
                      # because i will be overwritten with the next
                      # index in the range
```

Los nombres en la lista no se eliminan cuando finaliza el bucle, pero si la secuencia está vacía, el bucle no les habrá asignado nada. Sugerencia: la función incorporada `range()` retorna un iterador de enteros adecuado para emular el efecto de Pascal `for i := a to b do`; por ejemplo, `list(range(3))` retorna la lista `[0, 1, 2]`.

Nota: Hay una sutileza cuando la secuencia está siendo modificada por el bucle (esto solo puede ocurrir para secuencias mutables, por ejemplo, listas). Se utiliza un contador interno para realizar un seguimiento de qué elemento se usa a continuación, y esto se incrementa en cada iteración. Cuando este contador ha alcanzado la longitud de la secuencia, el bucle termina. Esto significa que si la suite elimina el elemento actual (o anterior) de la secuencia, se omitirá el siguiente elemento (ya que obtiene el índice del elemento actual que ya ha sido tratado). Del mismo modo, si la suite inserta un elemento en la secuencia anterior al elemento actual, el elemento actual será tratado nuevamente la próxima vez a través del bucle. Esto puede conducir a errores graves que se pueden evitar haciendo una copia temporal usando una porción de la secuencia completa, por ejemplo,

```
for x in a[:]:
    if x < 0: a.remove(x)
```

8.4 La sentencia `try`

La sentencia *try* es especifica para gestionar excepciones o código de limpieza para un grupo de sentencias:

```
try_stmt   ::=  try1_stmt | try2_stmt
try1_stmt  ::=  "try" ":" suite
                ("except" [expression ["as" identifier]] ":" suite)+
                ["else" ":" suite]
                ["finally" ":" suite]
try2_stmt  ::=  "try" ":" suite
                "finally" ":" suite
```

The *except* clause(s) specify one or more exception handlers. When no exception occurs in the *try* clause, no exception handler is executed. When an exception occurs in the `try` suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last; it matches any exception. For an except clause with an expression, that expression is evaluated, and the clause matches the exception if the resulting object is «compatible» with the exception. An object is compatible with an exception if it is the class or a base class of the exception object, or a tuple containing an item that is the class or a base class of the exception object.

Si ninguna cláusula `except` coincide con la excepción, la búsqueda de un gestor de excepciones continúa en el código circundante y en la pila de invocación.[1]

Si la evaluación de una expresión en el encabezado de una cláusula `except` genera una excepción, la búsqueda original de un gestor se cancela y se inicia la búsqueda de la nueva excepción en el código circundante y en la pila de llamadas (se trata como si toda la sentencia *try* provocó la excepción).

Cuando se encuentra una cláusula `except` coincidente, la excepción se asigna al destino especificado después de la palabra clave `as` en esa cláusula `except`, si está presente, y se ejecuta la suite de cláusulas `except`. Todas las cláusulas `except` deben tener un bloque ejecutable. Cuando se alcanza el final de este bloque, la ejecución continúa normalmente después de toda la sentencia try. (Esto significa que si existen dos gestores de errores anidados para la misma excepción, y la excepción ocurre en la cláusula `try` del gestor interno, el gestor externo no gestionará la excepción).

Cuando se ha asignado una excepción usando `as target`, se borra al final de la cláusula `except`. Esto es como si

```
except E as N:
    foo
```

fue traducido a

```
except E as N:
    try:
        foo
    finally:
        del N
```

Esto significa que la excepción debe asignarse a un nombre diferente para poder referirse a ella después de la cláusula `except`. Las excepciones se borran porque con el seguimiento vinculado a ellas, forman un bucle de referencia con el marco de la pila, manteniendo activos todos los locales en esa pila hasta que ocurra la próxima recolección de basura.

Antes de que se ejecute un conjunto de cláusulas `except`, los detalles sobre la excepción se almacenan en el módulo `sys` y se puede acceder a través de `sys.exc_info()`. `sys.exc_info()` retorna 3 tuplas que consisten en la clase de excepción, la instancia de excepción y un objeto de rastreo (ver sección *Jerarquía de tipos estándar*) que identifica el

[1] La excepción se propaga a la pila de invocación a menos que haya una cláusula *finally* que provoque otra excepción. Esa nueva excepción hace que se pierda la anterior.

punto en el programa donde ocurrió la excepción. Lo valores `sys.exc_info()` se restauran a sus valores anteriores (antes de la llamada) al regresar de una función que manejó una excepción.

La cláusula opcional `else` se ejecuta si el flujo de control sale de la suite *try*, no se produjo ninguna excepción, y no se ejecutó la sentencia *return*, *continue* o *break*. Las excepciones en la cláusula `else` no se gestionaron con las cláusulas precedentes *except*.

Si está presente *finally*, esto especifica un gestor de "limpieza". La cláusula *try* se ejecuta, incluidas las cláusulas *except* y `else`. Si se produce una excepción en cualquiera de las cláusulas y no se maneja, la excepción se guarda temporalmente. Se ejecuta la cláusula `finally`. Si hay una excepción guardada, se vuelve a generar al final de la cláusula `finally`. Si la cláusula `finally` genera otra excepción, la excepción guardada se establece como el contexto de la nueva excepción. Si la cláusula `finally` ejecuta una sentencia *return*, *break* o *continue*, la excepción guardada se descarta:

```
>>> def f():
...     try:
...         1/0
...     finally:
...         return 42
...
>>> f()
42
```

La información de excepción no está disponible para el programa durante la ejecución de la cláusula *finally*.

Cuando se ejecuta una sentencia *return*, *break* o *continue* en la suite *try* de un `try`…la sentencia `finally`, la cláusula *finally* también se ejecuta "al salir".

El valor de retorno de una función está determinado por la última sentencia *return* ejecutada. Dado que la cláusula *finally* siempre se ejecuta, una sentencia `return` ejecutada en la cláusula `finally` siempre será la última ejecutada:

```
>>> def foo():
...     try:
...         return 'try'
...     finally:
...         return 'finally'
...
>>> foo()
'finally'
```

Se puede encontrar información adicional sobre las excepciones en la sección *Excepciones*, e información sobre el uso de la sentencia *raise*, para generar excepciones se puede encontrar en la sección *The raise statement*.

Distinto en la versión 3.8: Antes de Python 3.8, una sentencia *continue* era ilegal en la cláusula *finally* debido a un problema con la implementación.

8.5 La sentencia `with`

La sentencia *with* se usa para ajustar la ejecución de un bloque con métodos definidos por un administrador de contexto (ver sección *With Statement Context Managers*). Esto permite que los patrones de uso comunes *try*…*except*…*finally* se encapsulen para una reutilización conveniente.

```
with_stmt  ::=  "with" with_item ("," with_item)* ":" suite
with_item  ::=  expression ["as" target]
```

La ejecución de la sentencia *with* con un «item» se realiza de la siguiente manera:

1. La expresión de contexto (la expresión dada en *with_item*) se evalúa para obtener un administrador de contexto.
2. El administrador de contexto *__enter__()* se carga para su uso posterior.
3. El administrador de contexto *__exit__()* se carga para su uso posterior.
4. Se invoca el método del administrador de contexto *__enter__()*.
5. Si se incluyó el destino en la sentencia *with*, se le asigna el valor de retorno de *__enter__()*.

   Nota: La sentencia *with* garantiza que si el método *__enter__()* regresa sin error, entonces siempre se llamará a *__exit__()*. Por lo tanto, si se produce un error durante la asignación a la lista de destino, se tratará de la misma manera que si se produciría un error dentro de la suite. Vea el paso 6 a continuación.

6. La suite se ejecuta.
7. Se invoca el método del administrador de contexto *__exit__()*. Si una excepción causó la salida de la suite, su tipo, valor y rastreo se pasan como argumentos a *__exit__()*. De lo contrario, se proporcionan tres argumentos `None`.

   Si se salió de la suite debido a una excepción, y el valor de retorno del método *__exit__()* fue falso, la excepción se vuelve a plantear. Si el valor de retorno era verdadero, la excepción se suprime y la ejecución continúa con la sentencia que sigue a la sentencia *with*.

   Si se salió de la suite por cualquier motivo que no sea una excepción, el valor de retorno de *__exit__()* se ignora y la ejecución continúa en la ubicación normal para el tipo de salida que se tomó.

El siguiente código:

```
with EXPRESSION as TARGET:
    SUITE
```

es semánticamente equivalente a:

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

Con más de un elemento, los administradores de contexto se procesan como si varias sentencias *with* estuvieran anidadas:

```
with A() as a, B() as b:
    SUITE
```

es semánticamente equivalente a:

```
with A() as a:
    with B() as b:
        SUITE
```

Distinto en la versión 3.1: Soporte para múltiples expresiones de contexto.

Ver también:

PEP 343 - La sentencia «with» La especificación, antecedentes y ejemplos de la sentencia de Python *with*.

8.6 Definiciones de funciones

Una definición de función define una función objeto determinada por el usuario (consulte la sección *Jerarquía de tipos estándar*):

```
funcdef                   ::=  [decorators] "def" funcname "(" [parameter_list] ")"
                               ["->" expression] ":" suite
decorators                ::=  decorator+
decorator                 ::=  "@" dotted_name ["(" [argument_list [","]] ")"] NEWLINE
dotted_name               ::=  identifier ("." identifier)*
parameter_list            ::=  defparameter ("," defparameter)* "," "/" ["," [parameter_l
                               | parameter_list_no_posonly
parameter_list_no_posonly ::=  defparameter ("," defparameter)* ["," [parameter_list_sta
                               | parameter_list_starargs
parameter_list_starargs   ::=  "*" [parameter] ("," defparameter)* ["," ["**" parameter
                               | "**" parameter [","]
parameter                 ::=  identifier [":" expression]
defparameter              ::=  parameter ["=" expression]
funcname                  ::=  identifier
```

Una definición de función es una sentencia ejecutable. Su ejecución vincula el nombre de la función en el espacio de nombres local actual a un objeto de función (un contenedor alrededor del código ejecutable para la función). Este objeto de función contiene una referencia al espacio de nombres global actual como el espacio de nombres global que se utilizará cuando se llama a la función.

La definición de la función no ejecuta el cuerpo de la función; esto se ejecuta solo cuando se llama a la función.[2]

Una definición de función puede estar envuelta por una o más expresiones *decorator*. Las expresiones de decorador se evalúan cuando se define la función, en el ámbito que contiene la definición de la función. El resultado debe ser invocable, la cual se invoca con el objeto de función como único argumento. El valor retornado está vinculado al nombre de la función en lugar del objeto de la función. Se aplican múltiples decoradores de forma anidada. Por ejemplo, el siguiente código

```
@f1(arg)
@f2
def func(): pass
```

es más o menos equivalente a

```
def func(): pass
func = f1(arg)(f2(func))
```

[2] Una cadena de caracteres literal que aparece como la primera sentencia en el cuerpo de la función se transforma en el atributo `__doc__` de la función y, por lo tanto, en funciones *docstring*.

excepto que la función original no está vinculada temporalmente al nombre `func`.

Cuando uno o más *parameters* tienen la forma *parameter = expression*, se dice que la función tiene «valores de parámetros predeterminados». Para un parámetro con un valor predeterminado, el correspondiente *argument* puede omitirse desde una llamada, en cuyo caso se sustituye el valor predeterminado del parámetro. Si un parámetro tiene un valor predeterminado, todos los parámetros siguientes hasta el «`*`» también deben tener un valor predeterminado — esta es una restricción sintáctica que la gramática no expresa.

Los valores de los parámetros predeterminados se evalúan de izquierda a derecha cuando se ejecuta la definición de la función. Esto significa que la expresión se evalúa una vez, cuando se define la función, y que se utiliza el mismo valor «precalculado» para cada llamada . Esto es especialmente importante para entender cuando un parámetro predeterminado es un objeto mutable, como una lista o un diccionario: si la función modifica el objeto (por ejemplo, al agregar un elemento a una lista), el valor predeterminado está en efecto modificado. Esto generalmente no es lo que se pretendía. Una forma de evitar esto es usar `None` como valor predeterminado y probarlo explícitamente en el cuerpo de la función, por ejemplo:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section *Invocaciones*. A function call always assigns values to all parameters mentioned in the parameter list, either from positional arguments, from keyword arguments, or from default values. If the form «`*identifier`» is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form «`**identifier`» is present, it is initialized to a new ordered mapping receiving any excess keyword arguments, defaulting to a new empty mapping of the same type. Parameters after «`*`» or «`*identifier`» are keyword-only parameters and may only be passed by keyword arguments. Parameters before «`/`» are positional-only parameters and may only be passed by positional arguments.

Distinto en la versión 3.8: The `/` function parameter syntax may be used to indicate positional-only parameters. See **PEP 570** for details.

Los parámetros pueden tener *annotation* de la forma «`: expression`» que sigue al nombre del parámetro. Cualquier parámetro puede tener una anotación, incluso las de la forma `*identifier` o `** identifier`. Las funciones pueden tener una anotación «return» de la forma «`-> expression`» después de la lista de parámetros. Estas anotaciones pueden ser cualquier expresión válida de Python. La presencia de anotaciones no cambia la semántica de una función. Los valores de anotación están disponibles como valores de un diccionario con los nombres de los parámetros en el atributo `__annotations__` del objeto de la función. Si se usa `annotations` importada desde `__future__`, las anotaciones se conservan como cadenas de caracteres en tiempo de ejecución que permiten la evaluación pospuesta. De lo contrario, se evalúan cuando se ejecuta la definición de la función. En este caso, las anotaciones pueden evaluarse en un orden diferente al que aparecen en el código fuente.

También es posible crear funciones anónimas (funciones no vinculadas a un nombre), para uso inmediato en expresiones. Utiliza expresiones lambda, descritas en la sección *Lambdas*. Tenga en cuenta que la expresión lambda es simplemente una abreviatura para una definición de función simplificada; una función definida en una sentencia «`def`» puede pasarse o asignarse a otro nombre al igual que una función definida por una expresión lambda. La forma «`def`» es en realidad más poderosa ya que permite la ejecución de múltiples sentencias y anotaciones.

Nota del programador: Las funciones son objetos de la primera-clase. Una sentencia «`def`» ejecutada dentro de una definición de función define una función local que se puede retornar o pasar. Las variables libres utilizadas en la función anidada pueden acceder a las variables locales de la función que contiene el def. Vea la sección *Nombres y vínculos* para más detalles.

Ver también:

PEP 3107 - Anotaciones de funciones La especificación original para anotaciones de funciones.

PEP 484 - Sugerencias de tipo Definición de un significado estándar para anotaciones: sugerencias de tipo.

PEP 526 - Sintaxis para anotaciones variables Capacidad para escribir declaraciones de variables indirectas, incluidas variables de clase y variables de instancia

PEP 563 - Evaluación pospuesta de anotaciones Admite referencias directas dentro de las anotaciones conservando las anotaciones en forma de cadena de caracteres en tiempo de ejecución en lugar de una evaluación apresurada.

8.7 Definiciones de clase

Una definición de clase define un objeto de clase (ver sección *Jerarquía de tipos estándar*):

```
classdef    ::=  [decorators] "class" classname [inheritance] ":" suite
inheritance ::=  "(" [argument_list] ")"
classname   ::=  identifier
```

Una definición de clase es una sentencia ejecutable. La lista de herencia generalmente proporciona una lista de clases base (consulte *Metaclasses* para usos más avanzados), por lo que cada elemento de la lista debe evaluar a un objeto de clase que permita la subclasificación. Las clases sin una lista de herencia heredan, por defecto, de la clase base `object`; por lo tanto,

```
class Foo:
    pass
```

es equivalente a

```
class Foo(object):
    pass
```

La suite de la clase se ejecuta en un nuevo marco de ejecución (ver *Nombres y vínculos*), usando un espacio de nombres local recién creado y el espacio de nombres global original. (Por lo general, el bloque contiene principalmente definiciones de funciones). Cuando la suite de la clase finaliza la ejecución, su marco de ejecución se descarta pero se guarda su espacio de nombres local.[3] Luego se crea un objeto de clase utilizando la lista de herencia para las clases base y el espacio de nombres local guardado para el diccionario de atributos. El nombre de la clase está vinculado a este objeto de clase en el espacio de nombres local original.

El orden en que se definen los atributos en el cuerpo de la clase se conserva en el `__dict__` de la nueva clase. Tenga en cuenta que esto es confiable solo justo después de crear la clase y solo para las clases que se definieron utilizando la sintaxis de definición.

La creación de clases se puede personalizar en gran medida usando *metaclasses*.

Las clases también se pueden decorar: al igual que cuando se decoran funciones,

```
@f1(arg)
@f2
class Foo: pass
```

es más o menos equivalente a

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

[3] Una cadena de caracteres literal que aparece como la primera sentencia en el cuerpo de la clase se transforma en el elemento del espacio de nombre `__doc__` y, por lo tanto, de la clase *docstring*.

Las reglas de evaluación para las expresiones de decorador son las mismas que para los decoradores de funciones. El resultado se vincula al nombre de la clase.

** Nota del programador: ** Las variables definidas en la definición de la clase son atributos de clase; son compartidos por instancias. Los atributos de instancia se pueden establecer en un método con `self.name = value`. Se puede acceder a los atributos de clase e instancia a través de la notación «`self.name`», y un atributo de instancia oculta un atributo de clase con el mismo nombre cuando se accede de esta manera. Los atributos de clase se pueden usar como valores predeterminados para los atributos de instancia, pero el uso de valores mutables puede generar resultados inesperados. *Descriptors* se puede usar para crear variables de instancia con diferentes detalles de implementación.

Ver también:

PEP 3115 - Metaclases en Python 3000 La propuesta que cambió la declaración de metaclases a la sintaxis actual y la semántica de cómo se construyen las clases con metaclases.

PEP 3129 - Decoradores de clase La propuesta que agregó decoradores de clase. Los decoradores de funciones y métodos se introdujeron en **PEP 318**.

8.8 Corrutinas

Nuevo en la versión 3.5.

8.8.1 Definición de la función corrutina

```
async_funcdef ::= [decorators] "async" "def" funcname "(" [parameter_list] ")"
                  ["->" expression] ":" suite
```

La ejecución de las corrutinas de Python puede suspenderse y reanudarse en muchos puntos (ver *coroutine*). Dentro del cuerpo de una función de corrutina, los identificadores `await` y `async` se convierten en palabras claves reservadas; las expresiones *`await`*, *`async for`* y *`async with`* solo se puede usar en los cuerpos de funciones de corrutina.

Las funciones definidas con la sintaxis `async def` siempre son funciones de corrutina, incluso si no contienen palabras claves `await` o `async`.

Es un error del tipo `SyntaxError` usar una expresión `yield from` dentro del cuerpo de una función de corrutina.

Un ejemplo de una función corrutina:

```
async def func(param1, param2):
    do_stuff()
    await some_coroutine()
```

8.8.2 La sentencia `async for`

```
async_for_stmt ::= "async" for_stmt
```

Un *asynchronous iterable* es capaz de llamar código asincrónico en su implementación *iter*, y *asynchronous iterator* puede llamar a código asincrónico en su método *next*.

La sentencia `async for` permite una iteración apropiada sobre iteradores asincrónicos.

El siguiente código:

```
async for TARGET in ITER:
    SUITE
else:
    SUITE2
```

Es semánticamente equivalente a:

```
iter = (ITER)
iter = type(iter).__aiter__(iter)
running = True

while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        SUITE
else:
    SUITE2
```

Ver también *__aiter__()* y *__anext__()* para más detalles.

Es un error del tipo `SyntaxError` usar una sentencia `async for` fuera del cuerpo de una función de corrutina.

8.8.3 La sentencia `async with`

```
async_with_stmt  ::=  "async" with_stmt
```

Un *asynchronous context manager* es un *context manager* que puede suspender la ejecución en sus métodos *enter* y *exit*.

El siguiente código:

```
async with EXPRESSION as TARGET:
    SUITE
```

es semánticamente equivalente a:

```
manager = (EXPRESSION)
aexit = type(manager).__aexit__
aenter = type(manager).__aenter__
value = await aenter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not await aexit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        await aexit(manager, None, None, None)
```

Ver también *__aenter__()* y *__aexit__()* para más detalles.

Es un error del tipo `SyntaxError` usar una sentencia `async with` fuera del cuerpo de una función de corrutina.

Ver también:

PEP 492 - Corrutinas con sintaxis `async` y `await` La propuesta que convirtió a las corrutinas en un concepto independiente adecuado en Python, y agregó una sintaxis de soporte.

Notas al pie

CAPÍTULO 9

Componentes de nivel superior

El intérprete de Python puede obtener su entrada de varias fuentes: de un script que se le pasa como entrada estándar o como argumento del programa, escrito interactivamente, de un archivo fuente de módulo, etc. Este capítulo proporciona la sintaxis utilizada en estos casos.

9.1 Programas completos de Python

Si bien una especificación de lenguaje no necesita prescribir cómo se invoca al intérprete de lenguaje, es útil tener una noción de un programa completo de Python. Un programa completo de Python se ejecuta en un entorno mínimamente inicializado: todos los módulos estándar e integrados están disponibles, pero ninguno ha sido inicializado, excepto `sys` (varios servicios del sistema), `builtins` (funciones integradas, excepciones y `Ninguno`) y `__main__`. Este último se utiliza para proporcionar el espacio de nombres local y global para la ejecución del programa completo.

La sintaxis de un programa completo de Python es la entrada de archivos, que se describe en la siguiente sección.

El intérprete también puede invocarse en modo interactivo; en este caso, no lee ni ejecuta un programa completo, sino que lee y ejecuta una instrucción (posiblemente compuesta) a la vez. El entorno inicial es idéntico al de un programa completo; cada instrucción se ejecuta en el espacio de nombres de `__main__`.

Se puede pasar un programa completo al intérprete en tres formas: con la opción `-c` *string* de línea de comando, como un archivo pasado como primer argumento de línea de comando o como entrada estándar. Si el archivo o la entrada estándar es un dispositivo tty, el intérprete ingresa al modo interactivo; de lo contrario, ejecuta el archivo como un programa completo.

9.2 Entrada de archivo

Todas las entradas leídas de archivos no interactivos tienen la misma forma:

```
file_input   ::=   (NEWLINE | statement)*
```

Esta sintaxis se utiliza en las siguientes situaciones:

- al analizar un programa completo de Python (desde un archivo o desde una cadena);
- al analizar un módulo;
- al analizar una cadena pasada a la función: `exec()`;

9.3 Entrada interactiva

La entrada en modo interactivo se analiza utilizando la siguiente gramática:

```
interactive_input   ::=   [stmt_list] NEWLINE | compound_stmt NEWLINE
```

Tenga en cuenta que una declaración compuesta (de nivel superior) debe ir seguida de una línea en blanco en modo interactivo; esto es necesario para ayudar al analizador sintáctico a detectar el final de la entrada.

9.4 Entrada de expresión

`eval()` se utiliza para la entrada de expresiones. Ignora los espacios en blanco iniciales. El argumento de cadena para `eval()` debe tener la siguiente forma:

```
eval_input   ::=   expression_list NEWLINE*
```

CAPÍTULO 10

Especificación completa de la gramática

Esta es la gramática completa de Python, tal y como es leída por el generador de análisis sintáctico usado para analizar sintácticamente archivos fuentes de Python:

```
# Grammar for Python

# NOTE WELL: You should also follow all the steps listed at
# https://devguide.python.org/grammar/

# Start symbols for the grammar:
#       single_input is a single interactive statement;
#       file_input is a module or sequence of commands read from an input file;
#       eval_input is the input for the eval() functions.
#       func_type_input is a PEP 484 Python 2 function type comment
# NB: compound_stmt in single_input is followed by extra NEWLINE!
# NB: due to the way TYPE_COMMENT is tokenized it will always be followed by a NEWLINE
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER

decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
decorators: decorator+
decorated: decorators (classdef | funcdef | async_funcdef)

async_funcdef: ASYNC funcdef
funcdef: 'def' NAME parameters ['->' test] ':' [TYPE_COMMENT] func_body_suite

parameters: '(' [typedargslist] ')'

# The following definition for typedarglist is equivalent to this set of rules:
#
#     arguments = argument (',' [TYPE_COMMENT] argument)*
#     argument = tfpdef ['=' test]
#     kwargs = '**' tfpdef [','] [TYPE_COMMENT]
#     args = '*' [tfpdef]
```

(continué en la próxima página)

(proviene de la página anterior)

```
#     kwonly_kwargs = (',' [TYPE_COMMENT] argument)* (TYPE_COMMENT | [',' [TYPE_
↪COMMENT] [kwargs]])
#     args_kwonly_kwargs = args kwonly_kwargs | kwargs
#     poskeyword_args_kwonly_kwargs = arguments ( TYPE_COMMENT | [',' [TYPE_COMMENT]␣
↪[args_kwonly_kwargs]])
#     typedargslist_no_posonly  = poskeyword_args_kwonly_kwargs | args_kwonly_kwargs
#     typedarglist = (arguments ',' [TYPE_COMMENT] '/' [',' [[TYPE_COMMENT]␣
↪typedargslist_no_posonly]])|(typedargslist_no_posonly)"
#
# It needs to be fully expanded to allow our LL(1) parser to work on it.

typedargslist: (
  (tfpdef ['=' test] (',' [TYPE_COMMENT] tfpdef ['=' test])* ',' [TYPE_COMMENT] '/' [
↪',' [ [TYPE_COMMENT] tfpdef ['=' test] (
        ',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [TYPE_COMMENT] [
        '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',␣
↪[TYPE_COMMENT] ['**' tfpdef [','] [TYPE_COMMENT]]])
      | '**' tfpdef [','] [TYPE_COMMENT]]])
  | '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [TYPE_
↪COMMENT] ['**' tfpdef [','] [TYPE_COMMENT]]])
  | '**' tfpdef [','] [TYPE_COMMENT]] )
|  (tfpdef ['=' test] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',␣
↪[TYPE_COMMENT] [
   '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [TYPE_
↪COMMENT] ['**' tfpdef [','] [TYPE_COMMENT]]])
  | '**' tfpdef [','] [TYPE_COMMENT]])
  | '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [TYPE_
↪COMMENT] ['**' tfpdef [','] [TYPE_COMMENT]]])
  | '**' tfpdef [','] [TYPE_COMMENT])
)
tfpdef: NAME [':' test]

# The following definition for varargslist is equivalent to this set of rules:
#
#     arguments = argument (',' argument )*
#     argument = vfpdef ['=' test]
#     kwargs = '**' vfpdef [',']
#     args = '*' [vfpdef]
#     kwonly_kwargs = (',' argument )* [',' [kwargs]]
#     args_kwonly_kwargs = args kwonly_kwargs | kwargs
#     poskeyword_args_kwonly_kwargs = arguments [',' [args_kwonly_kwargs]]
#     varargslist_no_posonly = poskeyword_args_kwonly_kwargs | args_kwonly_kwargs
#     varargslist = arguments ',' '/' [','[(varargslist_no_posonly)]] | (varargslist_no_
↪posonly)
#
# It needs to be fully expanded to allow our LL(1) parser to work on it.

varargslist: vfpdef ['=' test ](',' vfpdef ['=' test])* ',' '/' [',' [ (vfpdef ['=␣
↪test] (',' vfpdef ['=' test])* [',' [
        '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
      | '**' vfpdef [',']]]
  | '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
  | '**' vfpdef [',']) ]] | (vfpdef ['=' test] (',' vfpdef ['=' test])* [',' [
        '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
      | '**' vfpdef [',']]]
  | '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
  | '**' vfpdef [',']
```

(continué en la próxima página)

(proviene de la página anterior)

```
)
vfpdef: NAME

stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | nonlocal_stmt | assert_stmt)
expr_stmt: testlist_star_expr (annassign | augassign (yield_expr|testlist) |
                     [('=' (yield_expr|testlist_star_expr))+ [TYPE_COMMENT]] )
annassign: ':' test ['=' (yield_expr|testlist_star_expr)]
testlist_star_expr: (test|star_expr) (',' (test|star_expr))* [',']
augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '&=' | '|=' | '^=' |
            '<<=' | '>>=' | '**=' | '//=')
# For normal and annotated assignments, additional restrictions enforced by the␣
↪interpreter
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist_star_expr]
yield_stmt: yield_expr
raise_stmt: 'raise' [test ['from' test]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from: ('from' (('.' | '...')* dotted_name | ('.' | '...')+)
              'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
nonlocal_stmt: 'nonlocal' NAME (',' NAME)*
assert_stmt: 'assert' test [',' test]

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef |␣
↪classdef | decorated | async_stmt
async_stmt: ASYNC (funcdef | with_stmt | for_stmt)
if_stmt: 'if' namedexpr_test ':' suite ('elif' namedexpr_test ':' suite)* ['else' ':'␣
↪suite]
while_stmt: 'while' namedexpr_test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' [TYPE_COMMENT] suite ['else' ':' suite]
try_stmt: ('try' ':' suite
           ((except_clause ':' suite)+
            ['else' ':' suite]
            ['finally' ':' suite] |
           'finally' ':' suite))
with_stmt: 'with' with_item (',' with_item)*  ':' [TYPE_COMMENT] suite
with_item: test ['as' expr]
# NB compile.c makes sure that the default except clause is last
except_clause: 'except' [test ['as' NAME]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

namedexpr_test: test [':=' test]
test: or_test ['if' or_test 'else' test] | lambdef
```

(continué en la próxima página)

(proviene de la página anterior)

```
test_nocond: or_test | lambdef_nocond
lambdef: 'lambda' [varargslist] ':' test
lambdef_nocond: 'lambda' [varargslist] ':' test_nocond
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
# <> isn't actually a valid comparison operator in Python. It's here for the
# sake of a __future__ import described in PEP 401 (which really works :-)
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
star_expr: '*' expr
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'@'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom_expr ['**' factor]
atom_expr: [AWAIT] atom trailer*
atom: ('(' [yield_expr|testlist_comp] ')' |
       '[' [testlist_comp] ']' |
       '{' [dictorsetmaker] '}' |
       NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')
testlist_comp: (namedexpr_test|star_expr) ( comp_for | (',' (namedexpr_test|star_
↪expr))* [','] )
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: (expr|star_expr) (',' (expr|star_expr))* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( ((test ':' test | '**' expr)
                   (comp_for | (',' (test ':' test | '**' expr))* [','])) |
                  ((test | star_expr)
                   (comp_for | (',' (test | star_expr))* [',']))  )

classdef: 'class' NAME ['(' [arglist] ')'] ':' suite

arglist: argument (',' argument)*  [',']

# The reason that keywords are test nodes instead of NAME is that using NAME
# results in an ambiguity. ast.c makes sure it's a NAME.
# "test '=' test" is really "keyword '=' test", but we have no such token.
# These need to be in a single rule to avoid grammar that is ambiguous
# to our LL(1) parser. Even though 'test' includes '*expr' in star_expr,
# we explicitly match '*' here, too, to give it proper precedence.
# Illegal combinations and orderings are blocked in ast.c:
# multiple (test comp_for) arguments are blocked; keyword unpackings
# that precede iterable unpackings are blocked; etc.
argument: ( test [comp_for] |
            test ':=' test |
            test '=' test |
            '**' test |
            '*' test )

comp_iter: comp_for | comp_if
```

(continué en la próxima página)

(proviene de la página anterior)

```
sync_comp_for: 'for' exprlist 'in' or_test [comp_iter]
comp_for: [ASYNC] sync_comp_for
comp_if: 'if' test_nocond [comp_iter]

# not used in grammar, but may appear in "node" passed from Parser to Compiler
encoding_decl: NAME

yield_expr: 'yield' [yield_arg]
yield_arg: 'from' test | testlist_star_expr

# the TYPE_COMMENT in suites is only parsed for funcdefs,
# but can't go elsewhere due to ambiguity
func_body_suite: simple_stmt | NEWLINE [TYPE_COMMENT NEWLINE] INDENT stmt+ DEDENT

func_type_input: func_type NEWLINE* ENDMARKER
func_type: '(' [typelist] ')' '->' test
# typelist is a modified typedargslist (see above)
typelist: (test (',' test)* [','
       ['*' [test] (',' test)* [',' '**' test] | '**' test]]
     |  '*' [test] (',' test)* [',' '**' test] | '**' test)
```

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-reference.

clase base abstracta Las clases base abstractas (ABC, por sus siglas en inglés *Abstract Base Class*) complementan al *duck-typing* brindando un forma de definir interfaces con técnicas como `hasattr()` que serían confusas o sutilmente erróneas (por ejemplo con *magic methods*). Las ABC introduce subclases virtuales, las cuales son clases que no heredan desde una clase pero aún así son reconocidas por `isinstance()` y `issubclass()`; vea la documentación del módulo `abc`. Python viene con muchas ABC incorporadas para las estructuras de datos(en el módulo `collections.abc`), números (en el módulo `numbers`) , flujos de datos (en el módulo `io`) , buscadores y cargadores de importaciones (en el módulo `importlib.abc`) . Puede crear sus propios ABCs con el módulo `abc`.

anotación Una etiqueta asociada a una variable, atributo de clase, parámetro de función o valor de retorno, usado por convención como un *type hint*.

Las anotaciones de variables no pueden ser accedidas en tiempo de ejecución, pero las anotaciones de variables globales, atributos de clase, y funciones son almacenadas en el atributo especial `__annotations__` de módulos, clases y funciones, respectivamente.

Vea *variable annotation*, *function annotation*, **PEP 484** y **PEP 526**, los cuales describen esta funcionalidad.

argumento Un valor pasado a una *function* (o *method*) cuando se llama a la función. Hay dos clases de argumentos:

- *argumento nombrado*: es un argumento precedido por un identificador (por ejemplo, `nombre=`) en una llamada a una función o pasado como valor en un diccionario precedido por `**`. Por ejemplo `3` y `5` son argumentos nombrados en las llamadas a `complex()`:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *argumento posicional* son aquellos que no son nombrados. Los argumentos posicionales deben aparecer al principio de una lista de argumentos o ser pasados como elementos de un *iterable* precedido por `*`. Por ejemplo, `3` y `5` son argumentos posicionales en las siguientes llamadas:

```
complex(3, 5)
complex(*(3, 5))
```

Los argumentos son asignados a las variables locales en el cuerpo de la función. Vea en la sección *Invocaciones* 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 Un valor asociado a un objeto que es referencias por el nombre usado expresiones de punto. Por ejemplo, si un objeto *o* tiene un atributo *a* sería referenciado como *o.a*.

a la espera Es un objeto a la espera (*awaitable*) que puede ser usado en una expresión `await`. Puede ser una *coroutine* o un objeto con un método `__await__()`. Vea también **PEP 492**.

BDFL Sigla de *Benevolent Dictator For Life*, benevolente dictador vitalicio, es decir Guido van Rossum, el creador de Python.

archivo binario Un *file object* capaz de leer y escribir *objetos tipo binarios*. Ejemplos de archivos binarios son los abiertos en modo binario (`'rb'`, `'wb'` o `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, e instancias de `io.BytesIO` y de `gzip.GzipFile`.

Vea también *text file* para un objeto archivo capaz de leer y escribir objetos `str`.

objetos tipo binarios Un objeto que soporta bufferobjects y puede exportar un búfer C-*contiguous*. Esto incluye todas los objetos `bytes`, `bytearray`, y `array.array`, así como muchos objetos comunes `memoryview`. Los objetos tipo binarios pueden ser usados para varias operaciones que usan datos binarios; éstas incluyen compresión, salvar a archivos binarios, y enviarlos a través de un socket.

Algunas operaciones necesitan que los datos binarios sean mutables. La documentación frecuentemente se refiere a éstos como «objetos tipo binario de lectura y escritura». Ejemplos de objetos de búfer mutables incluyen a `bytearray` y `memoryview` de la `bytearray`. Otras operaciones que requieren datos binarios almacenados en objetos inmutables («objetos tipo binario de sólo lectura»); ejemplos de éstos incluyen `bytes` y `memoryview` del objeto `bytes`.

bytecode El código fuente Python es compilado en *bytecode*, la representación interna de un programa python en el intérprete CPython. El *bytecode* también es guardado en caché en los archivos *.pyc* de tal forma que ejecutar el mismo archivo es más fácil la segunda vez (la recompilación desde el código fuente a *bytecode* puede ser evitada). Este «lenguaje intermedio» deberá 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.

callback A subroutine function which is passed as an argument to be executed at some point in the future.

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(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

El mismo concepto existe para clases, pero son menos usadas. Vea la documentación de *function definitions* y *class definitions* para mayor detalle sobre decoradores.

descriptor Cualquier objeto que define los métodos `__get__()`, `__set__()`, o `__delete__()`. Cuando un atributo de clase es un descriptor, su conducta enlazada especial es disparada durante la búsqueda del atributo. Normalmente, usando *a.b* para consultar, establecer o borrar un atributo busca el objeto llamado *b* en el diccionario de clase de *a*, pero si *b* es un descriptor, el respectivo método descriptor es llamado. Entender 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 mayor información sobre los métodos de los descriptores vea *Implementing Descriptors*.

diccionario Un arreglo asociativo, con claves arbitrarias que son asociadas a valores. Las claves pueden ser cualquier objeto con los métodos `__hash__()` y `__eq__()` . Son llamadas hash en Perl.

dictionary comprehension A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`. See *Despliegues para listas, conjuntos y diccionarios*.

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 dict-views.

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*.

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 *Definiciones de funciones*.

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 *Definiciones de funciones*.

Vea *variable annotation* y **PEP 484**, que describen esta funcionalidad.

__future__ Un pseudo-módulo que los programadores pueden usar para habilitar nuevas capacidades del lenguaje que no son compatibles con el intérprete actual.

Al importar el módulo `__future__` y evaluar sus variables, puede verse cuándo las nuevas capacidades fueron agregadas por primera vez al lenguaje y cuando se quedaron establecidas por defecto:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

recolección de basura El proceso de liberar la memoria de lo que ya no está en uso. Python realiza recolección de basura (*garbage collection*) llevando la cuenta de las referencias, y el recogedor de basura cíclico es capaz de detectar y romper las referencias cíclicas. El recogedor de basura puede ser controlado mediante el módulo `gc` .

generador Una función que retorna un *generator iterator*. Luce como una función normal excepto que contiene la expresión `yield` para producir series de valores utilizables en un bucle *for* o que pueden ser obtenidas una por una con la función `next()`.

Usualmente se refiere a una función generadora, pero puede referirse a un *iterador generador* en ciertos contextos. En aquellos casos en los que el significado no está claro, usar los términos completos evita la ambigüedad.

iterador generador Un objeto creado por una función *generator*.

Cada `yield` suspende temporalmente el procesamiento, recordando el estado de ejecución local (incluyendo las variables locales y las sentencias *try* pendientes). Cuando el «iterador generado» vuelve, retoma donde ha dejado, a diferencia de lo que ocurre con las funciones que comienzan nuevamente con cada invocación.

expresión generadora Una expresión que retorna un iterador. Luce como una expresión normal seguida por la cláusula `for` definiendo así una variable de bucle, un rango y una cláusula opcional `if`. La expresión combinada genera valores para la función contenedora:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

función genérica Una función compuesta de muchas funciones que implementan la misma operación para diferentes tipos. Qué implementación deberá ser usada durante la llamada a la misma es determinado por el algoritmo de despacho.

Vea también la entrada de glosario *single dispatch*, el decorador `functools.singledispatch()`, y **PEP 443**.

GIL Vea *global interpreter lock*.

bloqueo global del intérprete Mecanismo empleado por el intérprete *CPython* para asegurar que sólo un hilo ejecute el *bytecode* Python por vez. Esto simplifica la implementación de CPython haciendo que el modelo de objetos (incluyendo algunos críticos como `dict`) están implícitamente a salvo de acceso concurrente. Bloqueando el intérprete completo se simplifica hacerlo multi-hilos, a costa de mucho del paralelismo ofrecido por las máquinas con múltiples procesadores.

Sin embargo, algunos módulos de extensión, tanto estándar como de terceros, están diseñados para liberar el GIL cuando se realizan tareas computacionalmente intensivas como la compresión o el *hashing*. Además, el GIL siempre es liberado cuando se hace entrada/salida.

Esfuerzos previos hechos para crear un intérprete «sin hilos» (uno que bloquee los datos compartidos con una granularidad mucho más fina) no han sido exitosos debido a que el rendimiento sufrió para el caso más común de un solo procesador. Se cree que superar este problema de rendimiento haría la implementación mucho más compleja y por tanto, más costosa de mantener.

hash-based pyc Un archivo cache de *bytecode* que usa el *hash* en vez de usar el tiempo de la última modificación del archivo fuente correspondiente para determinar su validez. Vea *Invalidación del código de bytes en caché*.

hashable Un objeto es *hashable* si tiene un valor de hash que nunca cambiará durante su tiempo de vida (necesita un método *__hash__()*), y puede ser comparado con otro objeto (necesita el método *__eq__()*). Los objetos hashables que se comparan iguales deben tener el mismo número hash.

Ser *hashable* hace a un objeto utilizable como clave de un diccionario y miembro de un set, porque éstas estructuras de datos usan los valores de hash internamente.

La mayoría de los objetos inmutables incorporados en Python son *hashables*; los contenedores mutables (como las listas o los diccionarios) no lo son; los contenedores inmutables (como tuplas y conjuntos *frozensets*) son *hashables* si sus elementos son *hashables* . Los objetos que son instancias de clases definidas por el usuario son *hashables* por defecto. Todos se comparan como desiguales (excepto consigo mismos), y su valor de hash está derivado de su función `id()`.

IDLE El entorno integrado de desarrollo de Python, o *Integrated Development Environment for Python*. IDLE es un editor básico y un entorno de intérprete que se incluye con la distribución estándar de Python.

inmutable Un objeto con un valor fijo. Los objetos inmutables son números, cadenas y tuplas. Éstos objetos no pueden ser alterados. Un nuevo objeto debe ser creado si un valor diferente ha de ser guardado. Juegan un rol importante en lugares donde es necesario un valor de hash constante, por ejemplo como claves de un diccionario.

ruta de importación Una lista de las ubicaciones (o *entradas de ruta*) que son revisadas por *path based finder* al importar módulos. Durante la importación, ésta lista de localizaciones usualmente viene de `sys.path`, pero para los subpaquetes también puede incluir al atributo `__path__` del paquete padre.

importar El proceso mediante el cual el código Python dentro de un módulo se hace alcanzable desde otro código Python en otro módulo.

importador Un objeto que buscan y lee un módulo; un objeto que es tanto *finder* como *loader*.

interactivo Python tiene un intérprete interactivo, lo que significa que puede ingresar sentencias y expresiones en el prompt del intérprete, ejecutarlos de inmediato y ver sus resultados. Sólo ejecute `python` sin argumentos (podría seleccionarlo desde el menú principal de su computadora). Es una forma muy potente de probar nuevas ideas o inspeccionar módulos y paquetes (recuerde `help(x)`).

interpretado Python es un lenguaje interpretado, a diferencia de uno compilado, a pesar de que la distinción puede ser difusa debido al compilador a *bytecode*. Esto significa que los archivos fuente pueden ser corridos directamente, sin crear explícitamente un ejecutable que es corrido luego. Los lenguajes interpretados típicamente tienen ciclos de desarrollo y depuración más cortos que los compilados, sin embargo sus programas suelen correr más lentamente. Vea también *interactive*.

apagado del intérprete Cuando se le solicita apagarse, el intérprete Python ingresa a un fase especial en la cual gradualmente libera todos los recursos reservados, como módulos y varias estructuras internas críticas. También hace varias llamadas al *recolector de basura*. Esto puede disparar la ejecución de código de destructores definidos por el usuario o *weakref callbacks*. El código ejecutado durante la fase de apagado puede encontrar varias excepciones debido a que los recursos que necesita pueden no funcionar más (ejemplos comunes son los módulos de bibliotecas o los artefactos de advertencias *warnings machinery*)

La principal razón para el apagado del intérpreter es que el módulo `__main__` o el script que estaba corriendo termine su ejecución.

iterable An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an *__iter__()* method or with a *__getitem__()* method that implements *Sequence* semantics.

Los iterables pueden ser usados en el bucle *for* y en muchos otros sitios donde una secuencia es necesaria (`zip()`, `map()`, ...). Cuando un objeto iterable es pasado como argumento a la función incorporada `iter()`, retorna un iterador para el objeto. Este iterador pasa así el conjunto de valores. Cuando se usan iterables, normalmente no es necesario llamar a la función `iter()` o tratar con los objetos iteradores usted mismo. La sentencia `for` lo hace automáticamente por usted, creando un variable temporal sin nombre para mantener el iterador mientras dura el bucle. Vea también *iterator*, *sequence*, y *generator*.

iterador Un objeto que representa un flujo de datos. Llamadas repetidas al método `__next__()` del iterador (o al pasar la función incorporada `next()`) retorna ítems sucesivos del flujo. Cuando no hay más datos disponibles, una excepción `StopIteration` es disparada. En este momento, el objeto iterador está exhausto y cualquier llamada posterior al método `__next__()` sólo dispara otra vez `StopIteration`. Los iteradores necesitan tener un método *`__iter__()`* que retorna el objeto iterador mismo así cada iterador es también un iterable y puede ser usado en casi todos los lugares donde los iterables son aceptados. Una excepción importante es el código que intenta múltiples pases de iteración. Un objeto contenedor (como la `list`) produce un nuevo iterador cada vez que pasa a una función `iter()` o se usa en un bucle *`for`*. Intentar ésto con un iterador simplemente retornaría el mismo objeto iterador exhausto usado en previas iteraciones, haciéndolo aparecer como un contenedor vacío.

Puede encontrar más información en typeiter.

función clave Una función clave o una función de colación es un invocable que retorna un valor usado para el ordenamiento o clasificación. Por ejemplo, `locale.strxfrm()` es usada para producir claves de ordenamiento que se adaptan a las convenciones específicas de ordenamiento de un *locale*.

Cierta cantidad de herramientas de Python aceptan funciones clave para controlar como los elementos son ordenados o agrupados. Incluyendo a `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, y `itertools.groupby()`.

Hay varias formas de crear una función clave. Por ejemplo, el método `str.lower()` puede servir como función clave para ordenamientos que no distingan mayúsculas de minúsculas. Como alternativa, una función clave puede ser realizada con una expresión *`lambda`* como `lambda r: (r[0], r[2])`. También, el módulo `operator` provee tres constructores de funciones clave: `attrgetter()`, `itemgetter()`, y `methodcaller()`. Vea en Sorting HOW TO ejemplos de cómo crear y usar funciones clave.

argumento nombrado Vea *argument*.

lambda Una función anónima de una línea consistente en un sola *expression* que es evaluada cuando la función es llamada. La sintaxis para crear una función lambda es `lambda [parameters]: expression`

LBYL Del inglés *Look before you leap*, «mira antes de saltar». Es un estilo de codificación que prueba explícitamente las condiciones previas antes de hacer llamadas o búsquedas. Este estilo contrasta con la manera *EAFP* y está caracterizado por la presencia de muchas sentencias *`if`*.

En entornos multi-hilos, el método LBYL tiene el riesgo de introducir condiciones de carrera entre los hilos que están «mirando» y los que están «saltando». Por ejemplo, el código, *if key in mapping: return mapping[key]*‘ puede fallar si otro hilo remueve *key* de *mapping* después del test, pero antes de retornar el valor. Este problema puede ser resuelto usando bloqueos o empleando el método EAFP.

lista Es una *sequence* Python incorporada. A pesar de su nombre es más similar a un arreglo en otros lenguajes que a una lista enlazada porque el acceso a los elementos es O(1).

comprensión de listas Una forma compacta de procesar todos o parte de los elementos en una secuencia y retornar una lista como resultado. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` genera una lista de cadenas conteniendo números hexadecimales (0x..) entre 0 y 255. La cláusula *`if`* es opcional. Si es omitida, todos los elementos en `range(256)` son procesados.

cargador Un objeto que carga un módulo. Debe definir el método llamado `load_module()`. Un cargador es normalmente retornados por un *finder*. Vea **PEP 302** para detalles y `importlib.abc.Loader` para una *abstract base class*.

método mágico Una manera informal de llamar a un *special method*.

mapeado Un objeto contenedor que permite recupero de claves arbitrarias y que implementa los métodos especificados en la `Mapping` o `MutableMapping` abstract base classes. Por ejemplo, `dict`, `collections.defaultdict`, `collections.OrderedDict` y `collections.Counter`.

meta buscadores de ruta Un *finder* retornado por una búsqueda de `sys.meta_path`. Los meta buscadores de ruta están relacionados a *buscadores de entradas de rutas*, pero son algo diferente.

Vea en `importlib.abc.MetaPathFinder` los métodos que los meta buscadores de ruta implementan.

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
>>> 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 Un *module* Python que puede contener submódulos o recursivamente, subpaquetes. Técnicamente, un paquete es un módulo Python con un atributo `__path__`.

Vea también *regular package* y *namespace package*.

parámetro Una entidad nombrada en una definición de una *function* (o método) que especifica un *argument* (o en algunos casos, varios argumentos) que la función puede aceptar. Existen cinco tipos de argumentos:

- *posicional o nombrado*: especifica un argumento que puede ser pasado tanto como *posicional* o como *nombrado*. Este es el tipo por defecto de parámetro, como *foo* y *bar* en el siguiente ejemplo:

```
def func(foo, bar=None): ...
```

- *sólo posicional*: especifica un argumento que puede ser pasado sólo por posición. Los parámetros sólo posicionales pueden ser definidos incluyendo un carácter `/` en la lista de parámetros de la función después de ellos, como *posonly1* y *posonly2* en el ejemplo que sigue:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *sólo nombrado*: especifica un argumento que sólo puede ser pasado por nombre. Los parámetros sólo por nombre pueden ser definidos incluyendo un parámetro posicional de una sola variable o un simple `*` antes de ellos en la lista de parámetros en la definición de la función, como *kw_only1* y *kw_only2* en el ejemplo siguiente:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *variable posicional*: especifica una secuencia arbitraria de argumentos posicionales que pueden ser brindados (además de cualquier argumento posicional aceptado por otros parámetros). Este parámetro puede ser definido anteponiendo al nombre del parámetro `*`, como a *args* en el siguiente ejemplo:

```
def func(*args, **kwargs): ...
```

- *variable nombrado*: especifica que arbitrariamente muchos argumentos nombrados pueden ser brindados (además de cualquier argumento nombrado ya aceptado por cualquier otro parámetro). Este parámetro puede ser definido anteponiendo al nombre del parámetro con `**`, como *kwargs* en el ejemplo precedente.

Los parámetros puede especificar tanto argumentos opcionales como requeridos, así como valores por defecto para algunos argumentos opcionales.

Vea también el glosario de *argument*, la pregunta respondida en la diferencia entre argumentos y parámetros, la clase `inspect.Parameter`, la sección *Definiciones de funciones* , 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 provisoria Una API provisoria es aquella que deliberadamente fue excluida de las garantías de compatibilidad hacia atrás de la biblioteca estándar. Aunque no se esperan cambios fundamentales en dichas interfaces, como están marcadas como provisionales, los cambios incompatibles hacia atrás (incluso remover la misma interfaz) podrían ocurrir si los desarrolladores principales lo estiman. Estos cambios no se hacen gratuitamente – solo ocurrirán si fallas fundamentales y serias son descubiertas que no fueron vistas antes de la inclusión de la API.

Incluso para APIs 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.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just *__getitem__()* and *__len__()*, adding `count()`, `index()`, *__contains__()*, and *__reversed__()*. Types that implement this expanded interface can be registered explicitly using `register()`.

set comprehension A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See *Despliegues para listas, conjuntos y diccionarios*.

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 *Special method names*.

sentencia Una sentencia es parte de un conjunto (un «bloque» de código). Una sentencia tanto es una *expression* como alguna de las varias sintaxis usando una palabra clave, como *if*, *while* o *for*.

codificación de texto Un códec que codifica las cadenas Unicode a bytes.

archivo de texto Un *file object* capaz de leer y escribir objetos `str`. Frecuentemente, un archivo de texto también accede a un flujo de datos binario y maneja automáticamente el *text encoding*. Ejemplos de archivos de texto que son abiertos en modo texto (`'r'` o `'w'`), `sys.stdin`, `sys.stdout`, y las instancias de `io.StringIO`.

Vea también *binary file* por objeto de archivos capaces de leer y escribir *objeto tipo binario*.

cadena con triple comilla Una cadena que está enmarcada por tres instancias de comillas (») o apostrofes (‘). Aunque no brindan ninguna funcionalidad que no está disponible usando cadenas con comillas simple, son útiles por varias razones. Permiten incluir comillas simples o dobles sin escapar dentro de las cadenas y pueden abarcar múltiples líneas sin el uso de caracteres de continuación, haciéndolas particularmente útiles para escribir docstrings.

tipo El tipo de un objeto Python determina qué tipo de objeto es; cada objeto tiene un tipo. El tipo de un objeto puede ser accedido por su atributo `__class__` o puede ser conseguido usando `type(obj)`.

alias de tipos Un sinónimo para un tipo, creado al asignar un tipo a un identificador.

Los alias de tipos son útiles para simplificar los *indicadores de tipo*. Por ejemplo:

```
from typing import List, Tuple

def remove_gray_shades(
        colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

podría ser más legible así:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

Vea `typing` y **PEP 484**, que describen esta funcionalidad.

indicador de tipo Una *annotation* que especifica el tipo esperado para una variable, un atributo de clase, un parámetro para una función o un valor de retorno.

Los indicadores de tipo son opcionales y no son obligados por Python pero son útiles para las herramientas de análisis de tipos estático, y ayuda a las IDE en el completado del código y la refactorización.

Los indicadores de tipo de las variables globales, atributos de clase, y funciones, no de variables locales, pueden ser accedidos usando `typing.get_type_hints()`.

Vea `typing` y **PEP 484**, que describen esta funcionalidad.

saltos de líneas universales Una manera de interpretar flujos de texto en la cual son reconocidos como finales de línea todas siguientes formas: la convención de Unix para fin de línea `'\n'`, la convención de Windows `'\r\n'`, y

la vieja convención de Macintosh `'\r'`. Vea **PEP 278** y **PEP 3116**, además de `bytes.splitlines()` para usos adicionales.

anotación de variable Una *annotation* de una variable o un atributo de clase.

Cuando se anota una variable o un atributo de clase, la asignación es opcional:

```
class C:
    field: 'annotation'
```

Las anotaciones de variables son frecuentemente usadas para *type hints*: por ejemplo, se espera que esta variable tenga valores de clase `int`:

```
count: int = 0
```

La sintaxis de la anotación de variables está explicada en la sección *Annotated assignment statements*.

Vea *function annotation*, **PEP 484** y **PEP 526**, los cuales describen esta funcionalidad.

entorno virtual Un entorno cooperativamente aislado de ejecución que permite a los usuarios de Python y a las aplicaciones instalar y actualizar paquetes de distribución de Python sin interferir con el comportamiento de otras aplicaciones de Python en el mismo sistema.

Vea también `venv`.

máquina virtual Una computadora definida enteramente por software. La máquina virtual de Python ejecuta el *bytecode* generado por el compilador de *bytecode*.

Zen de Python Un listado de los principios de diseño y la filosofía de Python que son útiles para entender y usar el lenguaje. El listado puede encontrarse ingresando «`import this`» en la consola interactiva.

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;
- el proyecto Docutils para creación de reStructuredText y el juego de Utilidades de Documentación;
- Fredrik Lundh por su proyecto Referencia Alternativa de Python para la cual Sphinx tuvo muchas ideas.

B.1 Contribuidores de la documentación de Python

Muchas personas han contribuido para el lenguaje de Python, la librería estándar de Python, y la documentación de Python. Revisa Misc/ACKS la distribución de Python para una lista parcial de contribuidores.

Es solamente con la aportación y contribuciones de la comunidad de Python que Python tiene tan fantástica documentación – Muchas gracias!

APÉNDICE C

History and License

C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see https://www.zope.org/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 and above	2.1.1	2001-now	PSF	yes

Nota: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenses and Acknowledgements for Incorporated Software* for an incomplete list of these licenses.

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.8.20

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.8.20 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.8.20 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.8.20 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.8.20 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   3.8.20.

4. PSF is making Python 3.8.20 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 3.8.20 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
```

```
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.8.20
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.8.20, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach␣
↪of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This␣
↪License
   Agreement does not grant permission to use PSF trademarks or trade name in␣
↪a
   trademark sense to endorse or promote products or services of Licensee, or␣
↪any
   third party.

8. By copying, installing or otherwise using Python 3.8.20, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.
```

(continué en la próxima página)

(proviene de la página anterior)

```
6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   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.
```

(continué en la próxima página)

(proviene de la página anterior)

```
7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.8.20 DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
```

(continué en la próxima página)

(proviene de la página anterior)

```
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie management

The `http.cookies` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode and UUdecode functions

The `uu` module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML Remote Procedure Calls

The `xmlrpc.client` module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

The `test_epoll` module contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

The `select` module contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

(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.
```

C.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski" implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

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 and dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from http://www.netlib.org/fp/. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
```

(continué en la próxima página)

(proviene de la página anterior)

```
*
************************************************************/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 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/)"
   *
```

(continué en la próxima página)

(proviene de la página anterior)

```
    * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
    * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
    * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
    * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
    * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
    * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
    * OF THE POSSIBILITY OF SUCH DAMAGE.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
    * 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
    *    being used are not cryptographic related :-).
    * 4. If you include any Windows specific code (or a derivative thereof) from
    *    the apps directory (application code) you must include an acknowledgement:
    *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
    *
    * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
    * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
    * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
    * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
    * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
    * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
    * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
    * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
    * SUCH DAMAGE.
    *
    * The licence and distribution terms for any publically available version or
    * derivative of this code cannot be changed.  i.e. this code cannot simply be
    * copied and put under another distribution licence
    * [including the GNU Public Licence.]
    */
```

C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

The `_ctypes` extension is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

The `zlib` extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

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

The implementation of the hash table used by the `tracemalloc` is based on the cfuhash project:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

The `_decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2016 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
```

(continué en la próxima página)

(proviene de la página anterior)

```
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N test suite

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

> Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang), All Rights Reserved.
>
> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> - Redistributions of works must retain the original copyright notice, this list of conditions and the following disclaimer.
> - Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
> - Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work without specific prior written permission.
>
> THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS «AS IS» AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

APÉNDICE D

Copyright

See *History and License* for complete license and permissions information.

Índice

No alfabético

A

B

C

D

E

F

G

H

M

N

O

P

R

S

T