Python Tutorial

Versión 3.14.0rc3

Guido van Rossum and the Python development team

octubre 01, 2025

Python Software Foundation
Email: docs@python.org

Índice general

1 Abriendo el apetito **3**

2 Usando el intérprete de Python **5**
2.1 Invocar el intérprete . . . 5
2.1.1 Paso de argumentos . . . 6
2.1.2 Modo interactivo . . . 6
2.2 El intérprete y su entorno . . . 6
2.2.1 Codificación del código fuente . . . 6

3 Una introducción informal a Python **9**
3.1 Usando Python como una calculadora . . . 9
3.1.1 Números . . . 9
3.1.2 Texto . . . 11
3.1.3 Listas . . . 15
3.2 Primeros pasos hacia la programación . . . 16

4 Más herramientas para control de flujo **19**
4.1 La sentencia `if` . . . 19
4.2 La sentencia `for` . . . 19
4.3 La función `range()` . . . 20
4.4 `break` and `continue` Statements . . . 21
4.5 `else` Clauses on Loops . . . 22
4.6 La sentencia `pass` . . . 22
4.7 La sentencia `match` . . . 23
4.8 Definir funciones . . . 26
4.9 Más sobre definición de funciones . . . 27
4.9.1 Argumentos con valores por omisión . . . 27
4.9.2 Palabras claves como argumentos . . . 28
4.9.3 Parámetros especiales . . . 30
4.9.4 Listas de argumentos arbitrarios . . . 32
4.9.5 Desempaquetando una lista de argumentos . . . 33
4.9.6 Expresiones lambda . . . 33
4.9.7 Cadenas de texto de documentación . . . 33
4.9.8 Anotación de funciones . . . 34
4.10 Intermezzo: Estilo de programación . . . 34

5 Estructuras de datos **37**
5.1 Más sobre listas . . . 37
5.1.1 Usar listas como pilas . . . 38
5.1.2 Usar listas como colas . . . 39
5.1.3 Comprensión de listas . . . 39

5.1.4 Listas por comprensión anidadas . . . 40
5.2 La instrucción `del` . . . 41
5.3 Tuplas y secuencias . . . 42
5.4 Conjuntos . . . 43
5.5 Diccionarios . . . 43
5.6 Técnicas de iteración . . . 44
5.7 Más acerca de condiciones . . . 46
5.8 Comparando secuencias y otros tipos . . . 47

6 Módulos **49**
6.1 Más sobre los módulos . . . 50
6.1.1 Ejecutar módulos como scripts . . . 51
6.1.2 El camino de búsqueda de los módulos . . . 51
6.1.3 Archivos «compilados» de Python . . . 52
6.2 Módulos estándar . . . 52
6.3 La función `dir()` . . . 53
6.4 Paquetes . . . 54
6.4.1 Importar * desde un paquete . . . 56
6.4.2 Referencias internas en paquetes . . . 56
6.4.3 Paquetes en múltiples directorios . . . 57

7 Entrada y salida **59**
7.1 Formateo elegante de la salida . . . 59
7.1.1 Formatear cadenas literales . . . 60
7.1.2 El método format() de cadenas . . . 61
7.1.3 Formateo manual de cadenas . . . 62
7.1.4 Viejo formateo de cadenas . . . 63
7.2 Leyendo y escribiendo archivos . . . 63
7.2.1 Métodos de los objetos Archivo . . . 64
7.2.2 Guardar datos estructurados con `json` . . . 65

8 Errores y excepciones **67**
8.1 Errores de sintaxis . . . 67
8.2 Excepciones . . . 67
8.3 Gestionando excepciones . . . 68
8.4 Lanzando excepciones . . . 70
8.5 Encadenamiento de excepciones . . . 71
8.6 Excepciones definidas por el usuario . . . 72
8.7 Definiendo acciones de limpieza . . . 72
8.8 Acciones predefinidas de limpieza . . . 74
8.9 Lanzando y gestionando múltiples excepciones no relacionadas . . . 74
8.10 Enriqueciendo excepciones con notas . . . 76

9 Clases **79**
9.1 Unas palabras sobre nombres y objetos . . . 79
9.2 Ámbitos y espacios de nombres en Python . . . 80
9.2.1 Ejemplo de ámbitos y espacios de nombre . . . 81
9.3 Un primer vistazo a las clases . . . 82
9.3.1 Sintaxis de definición de clases . . . 82
9.3.2 Objetos clase . . . 82
9.3.3 Objetos instancia . . . 83
9.3.4 Objetos método . . . 83
9.3.5 Variables de clase y de instancia . . . 84
9.4 Algunas observaciones . . . 85
9.5 Herencia . . . 86
9.5.1 Herencia múltiple . . . 87
9.6 Variables privadas . . . 88
9.7 Detalles y Cuestiones Varias . . . 89
9.8 Iteradores . . . 89

9.9 Generadores . . . 90
9.10 Expresiones generadoras . . . 91

10 Breve recorrido por la Biblioteca Estándar **93**
10.1 Interfaz del sistema operativo . . . 93
10.2 Comodines de archivos . . . 94
10.3 Argumentos de Líneas de Comandos . . . 94
10.4 Redirigir la salida de error y finalización del programa . . . 94
10.5 Coincidencia de patrones de cadena . . . 94
10.6 Matemáticas . . . 95
10.7 Acceso a Internet . . . 95
10.8 Fechas y tiempos . . . 96
10.9 Compresión de datos . . . 96
10.10 Medición de rendimiento . . . 97
10.11 Control de calidad . . . 97
10.12 Pilas incluidas . . . 98

11 Pequeño paseo por la Biblioteca Estándar— Parte II **99**
11.1 Formato de salida . . . 99
11.2 Plantillas . . . 100
11.3 Trabajar con registros estructurados conteniendo datos binarios . . . 101
11.4 Multi-hilos . . . 101
11.5 Registrando . . . 102
11.6 Referencias débiles . . . 103
11.7 Herramientas para trabajar con listas . . . 103
11.8 Decimal Floating-Point Arithmetic . . . 104

12 Entornos virtuales y paquetes **107**
12.1 Introducción . . . 107
12.2 Creando entornos virtuales . . . 107
12.3 Manejando paquetes con pip . . . 108

13 ¿Y ahora qué? **111**

14 Edición de entrada interactiva y sustitución de historial **113**
14.1 Autocompletado con tab e historial de edición . . . 113
14.2 Alternativas al intérprete interactivo . . . 113

15 Floating-Point Arithmetic: Issues and Limitations **115**
15.1 Error de Representación . . . 118

16 Apéndice **121**
16.1 Modo interactivo . . . 121
16.1.1 Manejo de errores . . . 121
16.1.2 Programas ejecutables de Python . . . 121
16.1.3 El archivo de inicio interactivo . . . 122
16.1.4 Los módulos de customización . . . 122

A Glosario **125**

B About this documentation **145**
B.1 Contributors to the Python documentation . . . 145

C Historia y Licencia **147**
C.1 Historia del software . . . 147
C.2 Términos y condiciones para acceder o usar Python . . . 148
C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2 . . . 148
C.2.2 ACUERDO DE LICENCIA DE BEOPEN.COM PARA PYTHON 2.0 . . . 149
C.2.3 ACUERDO DE LICENCIA CNRI PARA PYTHON 1.6.1 . . . 150
C.2.4 ACUERDO DE LICENCIA CWI PARA PYTHON 0.9.0 HASTA 1.2 . . . 151

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION . 151
C.3 Licencias y reconocimientos para software incorporado . . . 151
C.3.1 Mersenne Twister . . . 151
C.3.2 Sockets . . . 152
C.3.3 Servicios de socket asincrónicos . . . 153
C.3.4 Gestión de cookies . . . 153
C.3.5 Seguimiento de ejecución . . . 154
C.3.6 funciones UUencode y UUdecode . . . 154
C.3.7 Llamadas a procedimientos remotos XML . . . 155
C.3.8 test_epoll . . . 155
C.3.9 Seleccionar kqueue . . . 156
C.3.10 SipHash24 . . . 156
C.3.11 strtod y dtoa . . . 157
C.3.12 OpenSSL . . . 157
C.3.13 expat . . . 161
C.3.14 libffi . . . 161
C.3.15 zlib . . . 162
C.3.16 cfuhash . . . 162
C.3.17 libmpdec . . . 163
C.3.18 Conjunto de pruebas W3C C14N . . . 163
C.3.19 mimalloc . . . 164
C.3.20 asyncio . . . 164
C.3.21 Global Unbounded Sequences (GUS) . . . 165
C.3.22 Zstandard bindings . . . 165

D Derechos de autor **167**

Índice **169**

Truco

This tutorial is designed for *programmers* that are new to the Python language, **not** *beginners* who are new to programming.

Python es un lenguaje de programación potente y fácil de aprender. Tiene estructuras de datos de alto nivel eficientes y un simple pero efectivo sistema de programación orientado a objetos. La elegante sintaxis de Python y su tipado dinámico, junto a su naturaleza interpretada lo convierten en un lenguaje ideal para scripting y desarrollo rápido de aplicaciones en muchas áreas, para la mayoría de plataformas.

El intérprete de Python y la extensa librería estándar se encuentran disponibles libremente en código fuente y de forma binaria para la mayoría de las plataformas desde la Web de Python, https://www.python.org/, y se pueden distribuir libremente. El mismo sitio también contiene distribuciones y referencias a muchos módulos libres de Python de terceros, programas, herramientas y documentación adicional.

El intérprete de Python es fácilmente extensible con funciones y tipos de datos implementados en C o C++ (u otros lenguajes que permitan ser llamados desde C). Python también es apropiado como un lenguaje para extender aplicaciones modificables.

This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. Be aware that it expects you to have a basic understanding of programming in general. It helps to have a Python interpreter handy for hands-on experience, but all examples are self-contained, so the tutorial can be read off-line as well.

Para una descripción de los objetos estándar y de los módulos, ver library-index. reference-index dónde se ofrece una definición más formal del lenguaje. Para escribir extensiones en C o C++, leer extending-index y c-api-index. Existen diversos libros que cubren Python en detalle.

Este tutorial no pretende ser exhaustivo ni cubrir cada una de las características del lenguaje, ni siquiera las más utilizadas. En vez de eso, pretende introducir muchas de las funcionalidades más notables y brindar una idea clara acerca del estilo y el tipo de lenguaje que es Python. Después de leerlo podrás leer y escribir módulos y programas en Python, y estarás listo para aprender más acerca de las diversas librerías y módulos descritos en library-index.

Es interesante leer el *Glosario*.

CAPÍTULO 1

Abriendo el apetito

Si trabajas mucho con ordenadores, en algún momento encontrarás que hay alguna tarea que quieres automatizar. Por ejemplo, quizás quieres buscar y remplazar un texto en muchos ficheros o renombrar y reordenar un montón de imágenes de forma complicada. Quizás lo que quieres es escribir una pequeña base de datos personalizada, una interfaz gráfica o un juego simple.

Si eres un desarrollador profesional, quizás quieres trabajar con varias librerías de C/C++/Java pero encuentras el ciclo de escribir/compilar/probar/recompilar bastante lento. Quizás estás escribiendo una serie de pruebas para éstas librerías y te parece tedioso escribir el código de pruebas. O quizás has escrito un programa que puede utilizar un lenguaje como extensión y no quieres diseñar e implementar un lenguaje entero para tu aplicación.

Python es justo el lenguaje para ti.

Puedes escribir un script de shell de Unix o archivos por lotes de Windows para algunas de estas tareas. Los scripts de shell son mejores para mover archivos y cambiar datos de texto, pero no son adecuados para juegos o aplicaciones GUI. Podrías escribir un programa C / C ++ / Java, pero llevaría mucho tiempo de desarrollo obtener incluso un primer programa de borrador. Python es más fácil de usar, está disponible en los sistemas operativos Windows, macOS y Unix, y te ayudará a hacer el trabajo más rápidamente.

Python es fácil de utilizar y es un verdadero lenguaje de programación ofreciendo mucha más estructura y soporte para programas grandes que la que ofrecen scripts de shell o archivos por lotes. Por otro lado, Python también ofrece mayor comprobación de errores que C y siendo un *lenguaje de muy alto nivel* tiene tipos de datos de alto nivel incorporados como listas flexibles y diccionarios. Debido a sus tipos de datos más generales, Python es aplicable a más dominios que Awk o Perl, aunque hay muchas cosas que son tan sencillas en Python como en esos lenguajes.

Python te permite dividir tu programa en módulos que pueden reutilizarse en otros programas de Python. Tiene una gran colección de módulos estándar que puedes utilizar como la base de tus programas o como ejemplos para empezar a aprender Python. Algunos de estos módulos proporcionan cosas como entrada/salida de ficheros, llamadas a sistema, sockets e incluso interfaces a herramientas de interfaz gráfica como Tk.

Python es un lenguaje interpretado, lo cual puede ahorrarte mucho tiempo durante el desarrollo ya que no es necesario compilar ni enlazar. El intérprete puede usarse interactivamente, lo que facilita experimentar con características del lenguaje, escribir programas desechables o probar funciones cuando se hace desarrollo de programas de abajo hacia arriba. Es también una calculadora de escritorio práctica.

Python permite escribir programas compactos y legibles. Los programas en Python son típicamente más cortos que sus programas equivalentes en C, C++ o Java por varios motivos:

- los tipos de datos de alto nivel permiten expresar operaciones complejas en una sola instrucción;
- la agrupación de instrucciones se hace mediante indentación en vez de llaves de apertura y cierre;

- no es necesario declarar variables ni argumentos.

Python es *extensible*: si ya sabes programar en C es fácil añadir nuevas funciones o módulos al intérprete, ya sea para realizar operaciones críticas a velocidad máxima, o para enlazar programas de Python con bibliotecas que tal vez sólo estén disponibles de forma binaria (por ejemplo bibliotecas gráficas específicas de un fabricante). Una vez estés realmente entusiasmado, puedes enlazar el intérprete Python en una aplicación hecha en C y usarlo como lenguaje de extensión o de comando para esa aplicación.

Por cierto, el lenguaje recibe su nombre del programa de televisión de la BBC «Monty Python's Flying Circus» y no tiene nada que ver con reptiles. Hacer referencias sobre Monty Python en la documentación no sólo esta permitido, ¡sino que también está bien visto!

Ahora que estás emocionado con Python, querrás verlo en más detalle. Como la mejor forma de aprender un lenguaje es usarlo, el tutorial te invita a que juegues con el intérprete de Python a medida que vas leyendo.

En el próximo capítulo se explicará la mecánica de uso del intérprete. Esta es información bastante mundana, pero es esencial para poder probar los ejemplos que aparecerán más adelante.

El resto del tutorial introduce varias características del lenguaje y el sistema Python a través de ejemplos, empezando con expresiones, instrucciones y tipos de datos simples, pasando por funciones y módulos, y finalmente tocando conceptos avanzados como excepciones y clases definidas por el usuario.

CAPÍTULO 2

Usando el intérprete de Python

2.1 Invocar el intérprete

The Python interpreter is usually installed as `/usr/local/bin/python3.14` on those machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command:

```
python3.14
```

en el terminal[1]. Ya que la elección del directorio dónde vivirá el intérprete es una opción del proceso de instalación, puede estar en otros lugares; consulta a tu gurú de Python local o administrador de sistemas. (Por ejemplo, `/usr/local/python` es una alternativa popular).

On Windows machines where you have installed Python from the Microsoft Store, the `python3.14` command will be available. If you have the py.exe launcher installed, you can use the `py` command. See setting-envvars for other ways to launch Python.

Se puede salir del intérprete con estado de salida cero ingresando el carácter de fin de archivo (`Control-D` en Unix, `Control-Z` en Windows). Si eso no funciona, puedes salir del intérprete escribiendo el comando: `quit()`.

Las características para edición de líneas del intérprete incluyen edición interactiva, sustitución de historial y completado de código en sistemas que soportan la biblioteca GNU Readline . Quizás la forma más rápida para comprobar si las características de edición se encuentran disponibles es presionar `Control-P` en el primer prompt de Python que aparezca. Si se escucha un sonido, tienes edición de línea de comandos; ver Apéndice *Edición de entrada interactiva y sustitución de historial* para una introducción a las teclas. Si parece que no ocurre nada, o si se muestra `^P`, estas características no están disponibles; solo vas a poder usar la tecla de retroceso (*backspace*) para borrar los caracteres de la línea actual.

El intérprete funciona de manera similar al shell de Unix: cuando se le llama con una entrada estándar conectada a un terminal, lee y ejecuta comandos de manera interactiva; cuando se le llama con un argumento de nombre de archivo o con un archivo como entrada estándar, lee y ejecuta un *script* desde ese archivo.

Una segunda forma de iniciar el intérprete es `python -c comando [arg] ...`, que ejecuta las sentencias en *comando*, similar a la opción de shell `-c` . Como las sentencias de Python a menudo contienen espacios u otros caracteres que son especiales para el shell, generalmente se recomienda citar *comando* en su totalidad.

Algunos módulos de Python también son útiles como scripts. Estos pueden invocarse utilizando `python -m módulo [arg] ...`, que ejecuta el archivo fuente para *módulo* como si se hubiera escrito el nombre completo en la línea de comandos.

[1] En Unix, el intérprete de Python 3.x no está instalado por defecto con el ejecutable llamado `python`, por lo que no entra en conflicto con un ejecutable de Python 2.x instalado simultáneamente.

Cuando se usa un script, a veces es útil poder ejecutar el script y luego ingresar al modo interactivo. Esto se puede hacer pasando la `-i` antes del nombre del script.

Todas las opciones de la línea de comandos se describen en using-on-general.

2.1.1 Paso de argumentos

Cuando son conocidos por el intérprete, el nombre del script y los argumentos adicionales se convierten a una lista de cadenas de texto asignada a la variable `argv` del módulo `sys`. Puedes acceder a esta lista haciendo `import sys`. La longitud de la lista es al menos uno; cuando no se utiliza ningún script o argumento, `sys.argv[0]` es una cadena vacía. Cuando se pasa el nombre del script con `'-'` (lo que significa la entrada estándar), `sys.argv[0]` vale `'-'`. Cuando se usa `-c` *comando*, `sys.argv[0]` vale `'-c'`. Cuando se usa `-m` *módulo*, `sys.argv[0]` contiene el valor del nombre completo del módulo. Las opciones encontradas después de `-c` *comando* o `-m` *módulo* no son consumidas por el procesador de opciones de Python pero de todas formas se almacenan en `sys.argv` para ser manejadas por el comando o módulo.

2.1.2 Modo interactivo

Cuando se leen los comandos desde un terminal, se dice que el intérprete está en *modo interactivo*. En este modo, espera el siguiente comando con el *prompt primario*, generalmente tres signos de mayor que (`>>>`); para las líneas de continuación, aparece el *prompt secundario*, por defecto tres puntos (`...`). El intérprete imprime un mensaje de bienvenida que indica su número de versión y un aviso de copyright antes de imprimir el primer *prompt primario*:

```
$ python3.14
Python 3.14 (default, April 4 2024, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Las líneas de continuación son necesarias cuando se ingresa una construcción multilínea. Como ejemplo, echa un vistazo a la sentencia `if`:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

Para más información sobre el modo interactivo, ver *Modo interactivo*.

2.2 El intérprete y su entorno

2.2.1 Codificación del código fuente

De forma predeterminada, los archivos fuente de Python se tratan como codificados en UTF-8. En esa codificación, los caracteres de la mayoría de los idiomas del mundo se pueden usar simultáneamente en literales, identificadores y comentarios, aunque la biblioteca estándar solo usa caracteres ASCII para los identificadores, una convención que debería seguir cualquier código que sea portable.Para mostrar todos estos caracteres correctamente, tu editor debe reconocer que el archivo es UTF-8, y debe usar una fuente que admita todos los caracteres del archivo.

Para declarar una codificación que no sea la predeterminada, se debe agregar una línea de comentario especial como la *primera* línea del archivo. La sintaxis es la siguiente:

```
# -*- coding: encoding -*-
```

donde *encoding* es uno de los `codecs` soportados por Python.

Por ejemplo, para declarar que se utilizará la codificación de Windows-1252, la primera línea del archivo de código fuente debe ser:

```
# -*- coding: cp1252 -*-
```

Una excepción a la regla de *primera línea* es cuando el código fuente comienza con una *linea UNIX «shebang»*. En ese caso, la declaración de codificación debe agregarse como la segunda línea del archivo. Por ejemplo:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

Notas al pie

CAPÍTULO 3

Una introducción informal a Python

En los siguientes ejemplos, la entrada y la salida se distinguen por la presencia o ausencia de prompts (»> y …): para repetir el ejemplo, escribe todo después del prompt, cuando aparece; las líneas que no comienzan con un prompt son emitidas desde el intérprete. Ten en cuenta que en un ejemplo un prompt secundario en una linea en solitario significa que debes escribir una línea en blanco. Esto se utiliza para finalizar un comando multilínea.

Muchos de los ejemplos de este manual, incluso aquellos ingresados en el prompt interactivo, incluyen comentarios. Los comentarios en Python comienzan con el carácter numeral, `#`, y se extienden hasta el final visible de la línea. Un comentario quizás aparezca al comienzo de la línea o seguido de espacios en blanco o código, pero no dentro de una cadena de caracteres. Un carácter numeral dentro de una cadena de caracteres es sólo un carácter numeral. Ya que los comentarios son para aclarar código y no son interpretados por Python, pueden omitirse cuando se escriben los ejemplos.

Algunos ejemplos:

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1 Usando Python como una calculadora

Probemos algunos comandos simples de Python. Inicia el intérprete y espera el prompt primario, >>>. (No debería tardar mucho.)

3.1.1 Números

El intérprete funciona como una simple calculadora: puedes introducir una expresión en él y este escribirá los valores. La sintaxis es sencilla: los operadores `+`, `-`, `*` y `/` se pueden usar para realizar operaciones aritméticas; los paréntesis (`()`) pueden ser usados para agrupar. Por ejemplo:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> 8 / 5  # division always returns a floating-point number
1.6
```

Los números enteros (ej. `2`, `4`, `20`) tienen tipo `int`, los que tienen una parte fraccionaria (por ejemplo `5.0`, `1.6`) tiene el tipo `float`. Vamos a ver más acerca de los tipos numéricos más adelante en el tutorial.

La división (`/`) siempre retorna un número decimal de punto flotante. Para hacer *floor division* y obtener un número entero como resultado puede usarse el operador `//`; para calcular el resto puedes usar `%`:

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # floored quotient * divisor + remainder
17
```

Con Python, es posible usar el operador `**` para calcular potencias[1]:

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

El signo igual (=) se usa para asignar un valor a una variable. Después, no se muestra ningún resultado antes del siguiente prompt interactivo:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Si una variable no está «definida» (no se le ha asignado un valor), al intentar usarla dará un error:

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Hay soporte completo de punto flotante; operadores con operando mezclados convertirán los enteros a punto flotante:

```
>>> 4 * 3.75 - 1
14.0
```

En el modo interactivo, la última expresión impresa se asigna a la variable _. Esto significa que cuando se está utilizando Python como calculadora, es más fácil seguir calculando, por ejemplo:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
```

(continúe en la próxima página)

[1] Debido a que `**` tiene una prioridad mayor que `-`, `-3**2` se interpretará como `-(3**2)`, por lo tanto dará como resultado `-9`. Para evitar esto y obtener `9`, puedes usar `(-3)**2`.

(proviene de la página anterior)

```
>>> round(_, 2)
113.06
```

Esta variable debe ser tratada como de sólo lectura por el usuario. No le asignes explícitamente un valor; crearás una variable local independiente con el mismo nombre enmascarando la variable con el comportamiento mágico.

Además de `int` y `float`, Python admite otros tipos de números, como `Decimal` y `Fraction`. Python también tiene soporte incorporado para complex numbers, y usa el sufijo `j` o `J` para indicar la parte imaginaria (por ejemplo, `3+5j`).

3.1.2 Texto

Python puede manipular texto (representado por el tipo `str`, conocido como «cadenas de caracteres») al igual que números. Esto incluye caracteres «`!`», palabras «`conejo`», nombres «`París`», oraciones «`¡Te tengo a la vista!`», etc. «`Yay! :)`». Se pueden encerrar en comillas simples (`'...'`) o comillas dobles (`"..."`) con el mismo resultado[2].

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!"  # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975'  # digits and numerals enclosed in quotes are also strings
'1975'
```

Para citar una cita, debemos «escapar» la cita precediéndola con `\`. Alternativamente, podemos usar el otro tipo de comillas:

```
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

En el intérprete de Python, la definición de cadena y la cadena de salida pueden verse diferentes. La función `print()` produce una salida más legible, omitiendo las comillas de encuadre e imprimiendo caracteres escapados y especiales:

```
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), special characters are included in the string
'First line.\nSecond line.'
>>> print(s)  # with print(), special characters are interpreted, so \n produces␣
↪new line
First line.
Second line.
```

Si no quieres que los caracteres precedidos por `\` se interpreten como caracteres especiales, puedes usar *cadenas sin formato* agregando una `r` antes de la primera comilla:

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
```

(continúe en la próxima página)

[2] A diferencia de otros lenguajes, caracteres especiales como `\n` tienen el mismo significado con simple(`'...'`) y dobles (`"..."`) comillas. La única diferencia entre las dos es que dentro de las comillas simples no existe la necesidad de escapar `"` (pero tienes que escapar `\'`) y viceversa.

(proviene de la página anterior)

```
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

Hay un aspecto sutil en las cadenas sin formato: una cadena sin formato no puede terminar en un número impar de caracteres `\`; consultar en preguntas frequentes para obtener más información y soluciones.

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''`. End-of-line characters are automatically included in the string, but it's possible to prevent this by adding a `\` at the end of the line. In the following example, the initial newline is not included:

```
>>> print("""\
... Usage: thingy [OPTIONS]
...      -h                        Display this usage message
...      -H hostname               Hostname to connect to
... """)
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to

>>>
```

Las cadenas se pueden concatenar (pegar juntas) con el operador `+` y se pueden repetir con `*`:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'ununununium'
```

Dos o más *cadenas literales* (es decir, las encerradas entre comillas) una al lado de la otra se concatenan automáticamente.

```
>>> 'Py' 'thon'
'Python'
```

Esta característica es particularmente útil cuando quieres dividir cadenas largas:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

Esto solo funciona con dos literales, no con variables ni expresiones:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
  File "<stdin>", line 1
    prefix 'thon'
           ^^^^^^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
               ^^^^^
SyntaxError: invalid syntax
```

Si quieres concatenar variables o una variable y un literal, usa `+`:

```
>>> prefix + 'thon'
'Python'
```

Las cadenas de texto se pueden *indexar* (subíndices), el primer carácter de la cadena tiene el índice 0. No hay un tipo de dato diferente para los caracteres; un carácter es simplemente una cadena de longitud uno:

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

Los índices también pueden ser números negativos, para empezar a contar desde la derecha:

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

Nótese que -0 es lo mismo que 0, los índice negativos comienzan desde -1.

Además de los índices, las *rebanadas* (slicing) también están soportadas. Mientras que la indexación se utiliza para obtener caracteres individuales, *rebanar* te permite obtener una subcadena:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

Los índices de las rebanadas tienen valores por defecto útiles; el valor por defecto para el primer índice es cero, el valor por defecto para el segundo índice es la longitud de la cadena a rebanar.

```
>>> word[:2]   # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to the end
'on'
```

Nótese cómo el inicio siempre se incluye y el final siempre se excluye. Esto asegura que `s[:i] + s[i:]` siempre sea igual a `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Una forma de recordar cómo funcionan las rebanadas es pensar que los índices apuntan *entre* caracteres, con el borde izquierdo del primer carácter numerado 0. Luego, el punto derecho del último carácter de una cadena de *n* caracteres tiene un índice *n*, por ejemplo

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

La primera fila de números da la posición de los índices 0…6 en la cadena; La segunda fila da los correspondientes indices negativos. La rebanada desde *i* hasta *j* consta de todos los caracteres entre los bordes etiquetados *i* y *j*, respectivamente.

Para índices no negativos, la longitud de la rebanada es la diferencia de los índices, si ambos están dentro de los límites. Por ejemplo, la longitud de `word[1:3]` es 2.

Intentar usar un índice que es muy grande resultará en un error:

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Sin embargo, los índices de rebanadas fuera de rango se manejan satisfactoriamente cuando se usan para rebanar:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Las cadenas de Python no se pueden modificar, son *immutable*. Por eso, asignar a una posición indexada de la cadena resulta en un error:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Si necesitas una cadena diferente, deberías crear una nueva:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

La función incorporada `len()` retorna la longitud de una cadena:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Ver también

textseq
Las cadenas de texto son ejemplos de *tipos secuencias* y soportan las operaciones comunes para esos tipos.

string-methods
Las cadenas de texto soportan una gran cantidad de métodos para transformaciones básicas y búsqueda.

f-strings
Literales de cadena que tienen expresiones embebidas.

formatstrings
Aquí se da información sobre formateo de cadenas de texto con `str.format()`.

old-string-formatting
Aquí se describen con más detalle las antiguas operaciones para formateo utilizadas cuando una cadena de texto está a la izquierda del operador `%`.

3.1.3 Listas

Python tiene varios tipos de datos *compuestos*, utilizados para agrupar otros valores. El más versátil es la *lista*, la cual puede ser escrita como una lista de valores separados por coma (ítems) entre corchetes. Las listas pueden contener ítems de diferentes tipos, pero usualmente los ítems son del mismo tipo.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Al igual que las cadenas (y todas las demás tipos integrados *sequence*), las listas se pueden indexar y segmentar:

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

Las listas también admiten operaciones como concatenación:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

A diferencia de las cadenas, que son *immutable*, las listas son de tipo *mutable*, es decir, es posible cambiar su contenido:

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `list.append()` *method* (we will see more about methods later):

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Simple assignment in Python never copies data. When you assign a list to a variable, the variable refers to the *existing list*. Any changes you make to the list through one variable will be seen through all other variables that refer to it.:

```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba)  # they reference the same object
True
>>> rgba.append("Alph")
>>> rgb
["Red", "Green", "Blue", "Alph"]
```

Todas las operaciones de rebanado retornan una nueva lista que contiene los elementos pedidos. Esto significa que la siguiente rebanada retorna una shallow copy de la lista:

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
["Red", "Green", "Blue", "Alpha"]
```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> rgba
["Red", "Green", "Blue", "Alph"]
```

También es posible asignar a una rebanada, y esto incluso puede cambiar la longitud de la lista o vaciarla totalmente:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

La función predefinida `len()` también sirve para las listas

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

Es posible anidar listas (crear listas que contengan otras listas), por ejemplo:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2 Primeros pasos hacia la programación

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

```
>>> # Fibonacci series:
>>> # the sum of two elements defines the next
>>> a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
1
1
2
3
```

(continúe en la próxima página)

(proviene de la página anterior)

```
5
8
```

Este ejemplo introduce varias características nuevas.

- La primera línea contiene una *asignación múltiple*: las variables `a` y `b` obtienen simultáneamente los nuevos valores 0 y 1. En la última línea esto se usa nuevamente, demostrando que las expresiones de la derecha son evaluadas primero antes de que se realice cualquiera de las asignaciones. Las expresiones del lado derecho se evalúan de izquierda a derecha.
- El bucle `while` se ejecuta mientras la condición (aquí: `a < 10`) sea verdadera. En Python, como en C, cualquier valor entero que no sea cero es verdadero; cero es falso. La condición también puede ser una cadena de texto o una lista, de hecho, cualquier secuencia; cualquier cosa con una longitud distinta de cero es verdadera, las secuencias vacías son falsas. La prueba utilizada en el ejemplo es una comparación simple. Los operadores de comparación estándar se escriben igual que en C: < (menor que), > (mayor que), == (igual a), <= (menor que o igual a), >= (mayor que o igual a) y `!=` (distinto a).
- El cuerpo del bucle está *indentado*: la indentación es la forma que usa Python para agrupar declaraciones. En el intérprete interactivo debes teclear un tabulador o espacio(s) para cada línea indentada. En la práctica vas a preparar entradas más complicadas para Python con un editor de texto; todos los editores de texto modernos tienen la facilidad de agregar la indentación automáticamente. Cuando se ingresa una instrucción compuesta de forma interactiva, se debe finalizar con una línea en blanco para indicar que está completa (ya que el analizador no puede adivinar cuando tecleaste la última línea). Nota que cada línea de un bloque básico debe estar sangrada de la misma forma.
- The `print()` function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating-point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

  ```
  >>> i = 256*256
  >>> print('The value of i is', i)
  The value of i is 65536
  ```

  El parámetro nombrado *end* puede usarse para evitar el salto de linea al final de la salida, o terminar la salida con una cadena diferente:

  ```
  >>> a, b = 0, 1
  >>> while a < 1000:
  ...     print(a, end=',')
  ...     a, b = b, a+b
  ...
  0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
  ```

Notas al pie

CAPÍTULO 4

Más herramientas para control de flujo

Además de la sentencia `while` que acabamos de introducir, Python utiliza algunas más que encontraremos en este capítulo.

4.1 La sentencia `if`

Tal vez el tipo más conocido de sentencia sea el `if`. Por ejemplo:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

Puede haber cero o más bloques `elif`, y el bloque `else` es opcional. La palabra reservada `elif` es una abreviación de "else if", y es útil para evitar un sangrado excesivo. Una secuencia `if` … `elif` … `elif` … sustituye las sentencias `switch` o `case` encontradas en otros lenguajes.

Si necesitas comparar un mismo valor con muchas constantes, o comprobar que tenga un tipo o atributos específicos puede que encuentres útil la sentencia `match`. Para más detalles véase *La sentencia match*.

4.2 La sentencia `for`

La sentencia `for` en Python difiere un poco de lo que uno puede estar acostumbrado en lenguajes como C o Pascal. En lugar de siempre iterar sobre una progresión aritmética de números (como en Pascal) o darle al usuario la posibilidad de definir tanto el paso de la iteración como la condición de fin (como en C), la sentencia `for` de Python itera sobre los ítems de cualquier secuencia (una lista o una cadena de texto), en el orden que aparecen en la secuencia. Por ejemplo:

```
>>> # Measure some strings:
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Código que modifica una colección mientras se itera sobre la misma colección puede ser complejo de hacer bien. Sin embargo, suele ser más directo iterar sobre una copia de la colección o crear una nueva colección:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '□□□': 'active'}

# Strategy:  Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy:  Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3 La función `range()`

Si se necesita iterar sobre una secuencia de números, es apropiado utilizar la función integrada `range()`, la cual genera progresiones aritméticas:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

El valor final dado nunca es parte de la secuencia; `range(10)` genera 10 valores, los índices correspondientes para los ítems de una secuencia de longitud 10. Es posible hacer que el rango empiece con otro número, o especificar un incremento diferente (incluso negativo; algunas veces se lo llama “paso”):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

Para iterar sobre los índices de una secuencia, puedes combinar `range()` y `len()` así:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

En la mayoría de los casos, sin embargo, conviene usar la función `enumerate()`, ver *Técnicas de iteración*.

Algo extraño sucede si tan sólo muestras un `range`:

```
>>> range(10)
range(0, 10)
```

El objeto retornado por `range()` se comporta de muchas maneras como si fuera una lista, pero no lo es. Es un objeto que retorna los ítems sucesivos de la secuencia deseada cuando iteras sobre él, pero realmente no construye la lista, ahorrando entonces espacio.

Decimos que tal objeto es *iterable*; esto es, que se puede usar en funciones y construcciones que esperan algo de lo cual obtener ítems sucesivos hasta que se termine. Hemos visto que la declaración `for` es una de esas construcciones, mientras que un ejemplo de función que toma un iterable es la función `sum()`:

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

Más adelante veremos otras funciones que aceptan iterables cómo argumentos o retornan iterables. En el capítulo *Estructuras de datos*, discutiremos en más detalle sobre la `list()`.

4.4 `break` and `continue` Statements

The `break` statement breaks out of the innermost enclosing `for` or `while` loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

The `continue` statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
```

(continúe en la próxima página)

(proviene de la página anterior)

```
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5 `else` Clauses on Loops

In a `for` or `while` loop the `break` statement may be paired with an `else` clause. If the loop finishes without executing the `break`, the `else` clause executes.

In a `for` loop, the `else` clause is executed after the loop finishes its final iteration, that is, if no break occurred.

En un bucle `while`, se ejecuta después de que la condición del bucle se vuelva falsa.

In either kind of loop, the `else` clause is **not** executed if the loop was terminated by a `break`. Of course, other ways of ending the loop early, such as a `return` or a raised exception, will also skip execution of the `else` clause.

Esto se ejemplifica en el siguiente bucle `for`, que busca números primos:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for` loop, **not** the `if` statement.)

One way to think of the else clause is to imagine it paired with the `if` inside the loop. As the loop executes, it will run a sequence like if/if/if/else. The `if` is inside the loop, encountered a number of times. If the condition is ever true, a `break` will happen. If the condition is never true, the `else` clause outside the loop will execute.

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does with that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see *Gestionando excepciones*.

4.6 La sentencia `pass`

La sentencia `pass` no hace nada. Se puede usar cuando una sentencia es requerida por la sintaxis pero el programa no requiere ninguna acción. Por ejemplo:

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

Se usa normalmente para crear clases en su mínima expresión:

```
>>> class MyEmptyClass:
...     pass
...
```

Otro lugar donde se puede usar `pass` es como una marca de lugar para una función o un cuerpo condicional cuando estás trabajando en código nuevo, lo cual te permite pensar a un nivel de abstracción mayor. El `pass` se ignora silenciosamente:

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

For this last case, many people use the ellipsis literal `...` instead of `pass`. This use has no special meaning to Python, and is not part of the language definition (you could use any constant expression here), but `...` is used conventionally as a placeholder body as well. See bltin-ellipsis-object.

4.7 La sentencia `match`

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java or JavaScript (and many other languages), but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables. If no case matches, none of the branches is executed.

La forma más simple compara un valor expuesto con uno o más literales:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Note the last block: the «variable name» _ acts as a *wildcard* and never fails to match.

Se pueden combinar varios literales en un solo patrón usando `|` («o»):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Los patrones pueden también verse como asignaciones que desempaquetan, y pueden usarse para ligar variables:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

¡Observa éste caso con cuidado! El primer patrón tiene dos literales y puede considerarse una extensión del patrón literal que se mostró anteriormente. Pero los siguientes dos patrones combinan un literal y una variable, y la variable *liga* uno de los elementos del sujeto (`point`). El cuarto patrón captura ambos elementos, lo que lo hace conceptualmente similar a la asignación que desempaqueta `(x, y) = point`.

Si estás usando clases para estructurar tus datos, puedes usar el nombre de la clase seguida de una lista de argumentos similar a la de un constructor, pero con la capacidad de capturar atributos en variables:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

Puedes usar argumentos posicionales en algunas clases incorporadas que proveen un orden para sus atributos (por ej. dataclasses). También puedes definir una posición especifica para los atributos de los patrones si asignas en tu clase el atributo especial `__match_args__`. Si le asignas («x», «y»), los siguientes patrones son todos equivalentes entre sí (y todos ligan el atributo `y` a la variable `var`):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

Una recomendación para leer patrones es verlos como una forma extendida de lo que pondrías en el lado izquierdo de una asignación, para así entender cuáles variables tomarían qué valores. Sólo los nombres que aparecen por si solos (cómo `var` arriba) son asignados por una sentencia match. Nunca se asigna a los nombres con puntos (como `foo.bar`), nombres de atributos (los `x=` e `y=` arriba) o nombres de clases (reconocidos por los «(…)» junto a ellos, como `Point` arriba).

Los patrones pueden anidarse arbitrariamente. Por ejemplo, si tuviéramos una lista corta de puntos, con `__match_args__` añadido, podríamos aplicar match así:

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
```

(continúe en la próxima página)

(proviene de la página anterior)

```
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

Podemos añadir una clausula `if` a un patrón, conocida como «guarda». Si la guarda es falsa, `match` pasa a intentar el siguiente bloque case. Obsérvese que la captura de valores sucede antes de que la guarda sea evaluada:

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

Algunas otras propiedades importantes de esta sentencia:

- Al igual que las asignaciones con desempaquetado, los patrones de lista o tupla tienen exactamente el mismo sentido y realmente coinciden con cualquier secuencia arbitraria. Una excepción importante es que no coinciden ni con iteradores ni con cadenas de caracteres.
- Los patrones de secuencia soportan desempaquetado extendido: `[x, y, *otros]` y `(x, y, *otros)` funcionan de manera similar a las asignaciones con desempaquetado. El nombre luego de `*` también puede ser _, con lo cual `(x, y, *_)` coincide con cualquier secuencia de al menos dos elementos, sin ligar ninguno de los demás elementos.
- Los patrones de mapeo: `{"ancho_de_banda": c, "latencia": l}` capturan los valores `"ancho_de_banda"` y `"latencia"` de un diccionario. A diferencia de los patrones de secuencia, las claves adicionales son ignoradas. Puede usarse un desempaquetado como `**otros` . (Aunque `**_` sería redundante, con lo cual no está permitido)
- Pueden capturarse subpatrones usando la palabra clave `as`:

  ```
  case (Point(x1, y1), Point(x2, y2) as p2): ...
  ```

  capturará el segundo elemento de la entrada en `p2` (siempre y cuando la entrada sea una secuencia de dos puntos)
- La mayoría de los literales se comparan por igualdad, pero las instancias únicas `True`, `False` y `None` se comparan por identidad.
- En un patrón pueden usarse constantes con nombres. Los nombres deben tener puntos para impedir que sean interpretados como variables a capturar:

  ```
  from enum import Enum
  class Color(Enum):
      RED = 'red'
      GREEN = 'green'
      BLUE = 'blue'

  color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

  match color:
      case Color.RED:
          print("I see red!")
      case Color.GREEN:
          print("Grass is green")
      case Color.BLUE:
          print("I'm feeling the blues :(")
  ```

Para una explicación más detallada y más ejemplos, puede leerse **PEP 636** que está escrita en un formato de tutorial.

4.8 Definir funciones

Podemos crear una función que escriba la serie de Fibonacci hasta un límite determinado:

```
>>> def fib(n):    # write Fibonacci series less than n
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

La palabra reservada `def` se usa para definir funciones. Debe seguirle el nombre de la función y la lista de parámetros formales entre paréntesis. Las sentencias que forman el cuerpo de la función empiezan en la línea siguiente, y deben estar con sangría.

La primera sentencia del cuerpo de la función puede ser opcionalmente una cadena de texto literal; esta es la cadena de texto de documentación de la función, o *docstring*. (Puedes encontrar más acerca de docstrings en la sección *Cadenas de texto de documentación*.). Existen herramientas que usan las `docstrings` para producir documentación imprimible o disponible en línea, o para dejar que los usuarios busquen interactivamente a través del código; es una buena práctica incluir `docstrings` en el código que escribes, así que acostúmbrate a hacerlo.

La *ejecución* de una función introduce una nueva tabla de símbolos usada para las variables locales de la función. Más precisamente, todas las asignaciones de variables en la función almacenan el valor en la tabla de símbolos local; así mismo la referencia a variables primero mira la tabla de símbolos local, luego en la tabla de símbolos local de las funciones externas, luego la tabla de símbolos global, y finalmente la tabla de nombres predefinidos. Así, a variables globales y a variables de funciones que engloban a una función no se les puede asignar directamente un valor dentro de una función (a menos que se las nombre en la sentencia `global`, o mediante la sentencia `nonlocal` para variables de funciones que engloban la función local), aunque si pueden ser referenciadas.

Los parámetros reales (argumentos) para una llamada de función se introducen en la tabla de símbolos local de la función llamada cuando ésta se llama; por lo tanto, los argumentos se pasan usando *llamada por valor* (donde el *valor* es siempre una *referencia* al objeto, no el valor del objeto).[1] Cuando una función llama a otra función, o se llama a sí misma de forma recursiva, se crea una nueva tabla de símbolos locales para esa llamada.

Una definición de función asocia el nombre de la función con el objeto de función en la tabla de símbolos actual. El intérprete reconoce el objeto al que apunta ese nombre como una función definida por el usuario. Otros nombres también pueden apuntar a ese mismo objeto de función y también se pueden usar para acceder a la función:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Viniendo de otros lenguajes, puedes objetar que `fib` no es una función, sino un procedimiento, porque no retorna un valor. De hecho, técnicamente hablando, los procedimientos sin `return` sí retornan un valor, aunque uno bastante aburrido. Este valor se llama `None` (es un nombre predefinido). El intérprete por lo general no escribe el valor `None` si va a ser el único valor escrito. Puede verlo si realmente lo desea utilizando `print()`:

```
>>> fib(0)
>>> print(fib(0))
None
```

[1] En realidad, *llamadas por referencia de objeto* sería una mejor descripción, ya que si se pasa un objeto mutable, quien realiza la llamada verá cualquier cambio que se realice sobre el mismo (por ejemplo ítems insertados en una lista).

Es simple escribir una función que retorne una lista con los números de la serie de Fibonacci en lugar de imprimirlos:

```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Este ejemplo, como es usual, demuestra algunas características más de Python:

- La sentencia `return` retorna un valor en una función. `return` sin una expresión como argumento retorna `None`. Si se alcanza el final de una función, también se retorna `None`.
- The statement `result.append(a)` calls a *method* of the list object `result`. A method is a function that "belongs" to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*, see *Clases*) The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

4.9 Más sobre definición de funciones

También es posible definir funciones con un número variable de argumentos. Hay tres formas que pueden ser combinadas.

4.9.1 Argumentos con valores por omisión

La forma más útil es especificar un valor por omisión para uno o más argumentos. Esto crea una función que puede ser llamada con menos argumentos que los que permite. Por ejemplo:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

Esta función puede ser llamada de distintas maneras:

- pasando sólo el argumento obligatorio: `ask_ok('Do you really want to quit?')`
- pasando uno de los argumentos opcionales: `ask_ok('OK to overwrite the file?', 2)`
- o pasando todos los argumentos: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

Este ejemplo también introduce la palabra reservada `in`, la cual prueba si una secuencia contiene o no un determinado valor.

Los valores por omisión son evaluados en el momento de la definición de la función en el ámbito de la definición, entonces:

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

imprimirá `5`.

Advertencia importante: El valor por omisión es evaluado solo una vez. Existe una diferencia cuando el valor por omisión es un objeto mutable como una lista, diccionario, o instancia de la mayoría de las clases. Por ejemplo, la siguiente función acumula los argumentos que se le pasan en subsiguientes llamadas:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

Imprimirá

```
[1]
[1, 2]
[1, 2, 3]
```

Si no se quiere que el valor por omisión sea compartido entre subsiguientes llamadas, se pueden escribir la función así:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.9.2 Palabras claves como argumentos

Las funciones también puede ser llamadas usando *argumentos de palabras clave* (o argumentos nombrados) de la forma `kwarg=value`. Por ejemplo, la siguiente función:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

…acepta un argumento obligatorio (`voltage`)) y tres argumentos opcionales (`state`, `action`, y `type`). Esta función puede llamarse de cualquiera de las siguientes maneras:

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
```

(continúe en la próxima página)

(proviene de la página anterior)

```
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

…pero estas otras llamadas serían todas inválidas:

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

En una llamada a una función, los argumentos nombrados deben seguir a los argumentos posicionales. Cada uno de los argumentos nombrados pasados deben coincidir con un argumento aceptado por la función (por ejemplo, `actor` no es un argumento válido para la función `parrot`), y el orden de los mismos no es importante. Esto también se aplica a los argumentos obligatorios (por ejemplo, `parrot(voltage=1000)` también es válido). Ningún argumento puede recibir más de un valor al mismo tiempo. Aquí hay un ejemplo que falla debido a esta restricción:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

Cuando un parámetro formal de la forma `**nombre` está presente al final, recibe un diccionario (ver typesmapping) conteniendo todos los argumentos nombrados excepto aquellos correspondientes a un parámetro formal. Esto puede ser combinado con un parámetro formal de la forma `*nombre` (descrito en la siguiente sección) que recibe una *tupla* conteniendo los argumentos posicionales además de la lista de parámetros formales. (`*nombre` debe ocurrir antes de `**nombre`). Por ejemplo, si definimos una función así:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

Puede ser llamada así:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

…y por supuesto imprimirá:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Se debe notar que el orden en el cual los argumentos nombrados son impresos está garantizado para coincidir con el orden en el cual fueron provistos en la llamada a la función.

4.9.3 Parámetros especiales

Por defecto, los argumentos pueden enviarse a una función Python o bien por posición o explícitamente por clave. Para legibilidad y rendimiento tiene sentido restringir como se pueden enviar los argumentos, así un desarrollador necesitará mirar solamente la definición de la función para determinar si los argumentos se deben enviar por posición, por posición o clave, o por clave.

La definición de una función puede ser como la siguiente:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                - Keyword only
         -- Positional only
```

donde `/` y `*` son posicionales. Si se utilizan, esos símbolos indican el tipo de parámetro según la forma en que los argumentos deben enviarse a la función: solo por posición (*positional-only*), por posición o clave (*positional-or-keyword*) y solo por clave (*keyword-only*). Parámetros por clave pueden también denominarse parámetros por nombre o nombrados.

Argumentos posicionales o de palabras claves

Si `/` y `*` no están presentes en la definición de la función, los parámetros pueden ser pasados a una función posicionalmente o por palabra clave.

Parámetros únicamente posicionales

Mirando esto con un poco más de detalle, es posible señalar algunos parámetros como *únicamente posicionales*. En ese caso el orden de los parámetros es importante, y los parámetros no pueden ser indicados utilizando palabras claves. Parámetros únicamente posicionales son ubicados antes de una `/` (barra). La `/` es utilizada para separar lógicamente parámetros únicamente posicionales del resto. Si no existe una `/` en la definición de la función, no existen parámetros únicamente posicionales.

Los parámetros luego de una `/` pueden ser *únicamente posicionales* o *unicamente de palabras claves*.

Argumentos únicamente de palabras clave

Para señalar parámetros como *unicamente de palabras clave*, indicando que los parámetros deben ser pasados con una palabra clave, indiqué un `*` en la lista de argumentos antes del primer parámetro *únicamente de palabras clave*.

Ejemplos de Funciones

Considere el siguiente ejemplo de definiciones de funciones prestando especial atención a los marcadores `/` y `*`:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

La primer definición de función, `standard_arg`, la forma mas familiar, no indica ninguna restricción en las condiciones para llamarla y los parámetros deben ser pasados por posición o utilizando palabras clave:

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

La segunda función `pos_only_arg` está restringida a utilizar únicamente parámetros posicionales ya que existe una `/` en la definición de la función:

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments passed as keyword␣
↪arguments: 'arg'
```

The third function `kwd_only_arg` only allows keyword arguments as indicated by a `*` in the function definition:

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given

>>> kwd_only_arg(arg=3)
3
```

La última utiliza las tres convenciones en una misma definición de función:

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got some positional-only arguments passed as keyword␣
↪arguments: 'pos_only'
```

Finalmente, considere esta definición de función que contiene una colisión potencial entre los parámetros posicionales `name` y `**kwds` que incluye `name` como una clave:

```
def foo(name, **kwds):
    return 'name' in kwds
```

No hay una llamada posible que lo haga retornar `True` ya que la palabra clave `'name'` siempre se vinculará al primer parámetro. Por ejemplo:

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

Pero utilizando `/` (parámetros únicamente posicionales), es posible ya que permite utilizar `name` como un parámetro posicional y `name` como un parámetro de palabras clave:

```
>>> def foo(name, /, **kwds):
...     return 'name' in kwds
...
>>> foo(1, **{'name': 2})
True
```

En otras palabras, los nombres de parámetros únicamente posicionales pueden ser utilizados en `**kwds` sin ambigüedad.

Resumen

El caso de uso determinará qué parámetros utilizar en una definición de función:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

A modo de guía:

- Utilice únicamente posicionales si quiere que el nombre del parámetro esté disponible para el usuario. Esto es útil cuando el nombre del parámetro no tiene un significado real, si se quiere imponer el orden de los parámetros cuando una función es llamada o si necesita tomar algunos parámetros posicionales y palabras claves arbitrarias.
- Utilice parámetros únicamente de palabras clave cuando los nombres de los parámetros tienen un significado y la definición de la función será más entendible usando nombres explícitos o cuando desea evitar que los usuarios dependan de la posición de los parámetros que se pasan.
- En el caso de una API, use solo posicional para evitar que se rompan los cambios de la API si el nombre del parámetro se modifica en el futuro.

4.9.4 Listas de argumentos arbitrarios

Finalmente, la opción menos frecuentemente usada es especificar que una función puede ser llamada con un número arbitrario de argumentos. Estos argumentos serán organizados en una tupla (ver *Tuplas y secuencias*). Antes del número variable de argumentos, cero o más argumentos normales pueden estar presentes.:

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normalmente estos argumentos *variádicos* serán los últimos en la lista de parámetros formales, porque toman todo el remanente de argumentos que se pasan a la función. Cualquier parámetro que suceda luego del `*args` será “sólo de palabra clave”, o sea que sólo se pueden usar como argumentos nombrados y no como posicionales.

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

4.9.5 Desempaquetando una lista de argumentos

La situación inversa ocurre cuando los argumentos ya están en una lista o tupla pero necesitan ser desempaquetados para llamar a una función que requiere argumentos posicionales separados. Por ejemplo, la función predefinida `range()` espera los parámetros *inicio* y *fin*. Si estos no están disponibles en forma separada, se puede escribir la llamada a la función con el operador `*` para desempaquetar argumentos desde una lista o una tupla:

```
>>> list(range(3, 6))            # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))            # call with arguments unpacked from a list
[3, 4, 5]
```

Del mismo modo, los diccionarios pueden entregar argumentos nombrados con el operador `**`:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin
↪' demised !
```

4.9.6 Expresiones lambda

Pequeñas funciones anónimas pueden ser creadas con la palabra reservada `lambda`. Esta función retorna la suma de sus dos argumentos: `lambda a, b: a+b` Las funciones Lambda pueden ser usadas en cualquier lugar donde sea requerido un objeto de tipo función. Están sintácticamente restringidas a una sola expresión. Semánticamente, son solo azúcar sintáctica para definiciones normales de funciones. Al igual que las funciones anidadas, las funciones lambda pueden hacer referencia a variables desde el ámbito que la contiene:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument. For instance, `list.sort()` takes a sorting key function *key* which can be a lambda function:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.9.7 Cadenas de texto de documentación

Acá hay algunas convenciones sobre el contenido y formato de las cadenas de texto de documentación.

La primera línea debe ser siempre un resumen corto y conciso del propósito del objeto. Para ser breve, no se debe mencionar explícitamente el nombre o tipo del objeto, ya que estos están disponibles de otros modos (excepto si el nombre es un verbo que describe el funcionamiento de la función). Esta línea debe empezar con una letra mayúscula y terminar con un punto.

Si hay más líneas en la cadena de texto de documentación, la segunda línea debe estar en blanco, separando visualmente el resumen del resto de la descripción. Las líneas siguientes deben ser uno o más párrafos describiendo las convenciones para llamar al objeto, efectos secundarios, etc.

El analizador de Python no quita el sangrado de las cadenas de texto literales multi-líneas, entonces las herramientas que procesan documentación tienen que quitarlo si así lo desean. Esto se hace mediante la siguiente convención. La primera línea que no está en blanco *siguiente* a la primer línea de la cadena determina la cantidad de sangría para toda la cadena de documentación. (No podemos usar la primer línea ya que generalmente es adyacente a las comillas de apertura de la cadena y el sangrado no se nota en la cadena de texto). Los espacios en blanco «equivalentes» a este sangrado son luego quitados del comienzo de cada línea en la cadena. No deberían haber líneas con una sangría menor, pero si las hay todos los espacios en blanco del comienzo deben ser quitados. La equivalencia de espacios en blanco debe ser verificada luego de la expansión de tabuladores (a 8 espacios, normalmente).

Este es un ejemplo de un `docstring` multi-línea:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.

No, really, it doesn't do anything.
```

4.9.8 Anotación de funciones

Las anotaciones de funciones son información completamente opcional sobre los tipos usadas en funciones definidas por el usuario (ver **PEP 3107** y **PEP 484** para más información).

Annotations are stored in the `__annotations__` attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal `->`, followed by an expression, between the parameter list and the colon denoting the end of the `def` statement. The following example has a required argument, an optional argument, and the return value annotated:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.10 Intermezzo: Estilo de programación

Ahora que estás a punto de escribir piezas de Python más largas y complejas, es un buen momento para hablar sobre *estilo de programación*. La mayoría de los lenguajes pueden ser escritos (o mejor dicho, *formateados*) con diferentes estilos; algunos son mas fáciles de leer que otros. Hacer que tu código sea más fácil de leer por otros es siempre una buena idea, y adoptar un buen estilo de programación ayuda tremendamente a lograrlo.

Para Python, **PEP 8** se erigió como la guía de estilo a la que más proyectos adhirieron; promueve un estilo de programación fácil de leer y visualmente agradable. Todos los desarrolladores Python deben leerlo en algún momento; aquí están extraídos los puntos más importantes:

- Usar sangrías de 4 espacios, no tabuladores.

  4 espacios son un buen compromiso entre una sangría pequeña (permite mayor nivel de sangrado)y una sangría grande (más fácil de leer). Los tabuladores introducen confusión y es mejor dejarlos de lado.

- Recortar las líneas para que no superen los 79 caracteres.

  Esto ayuda a los usuarios con pantallas pequeñas y hace posible tener varios archivos de código abiertos, uno al lado del otro, en pantallas grandes.

- Usar líneas en blanco para separar funciones y clases, y bloques grandes de código dentro de funciones.
- Cuando sea posible, poner comentarios en una sola línea.
- Usar `docstrings`.
- Usar espacios alrededor de operadores y luego de las comas, pero no directamente dentro de paréntesis: `a = f(1, 2) + g(3, 4)`.
- Nombrar las clases y funciones consistentemente; la convención es usar `NotacionCamello` para clases y `minusculas_con_guiones_bajos` para funciones y métodos. Siempre usa `self` como el nombre para el primer argumento en los métodos (ver *Un primer vistazo a las clases* para más información sobre clases y métodos).
- No uses codificaciones estrafalarias si esperas usar el código en entornos internacionales. El default de Python, UTF-8, o incluso ASCII plano funcionan bien en la mayoría de los casos.
- De la misma manera, no uses caracteres no-ASCII en los identificadores si hay incluso una pequeñísima chance de que gente que hable otro idioma tenga que leer o mantener el código.

Notas al pie

CAPÍTULO 5

Estructuras de datos

Este capítulo describe en más detalle algunas cosas que ya has aprendido y agrega algunas cosas nuevas también.

5.1 Más sobre listas

El tipo de dato lista tiene algunos métodos más. Aquí están todos los métodos de los objetos lista:

`list.`**`append`**`(`*x*`)`

Add an item to the end of the list. Similar to `a[len(a):] = [x]`.

`list.`**`extend`**`(`*iterable*`)`

Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.

`list.`**`insert`**`(`*i*, *x*`)`

Inserta un ítem en una posición dada. El primer argumento es el índice del ítem delante del cual se insertará, por lo tanto `a.insert(0, x)` inserta al principio de la lista y `a.insert(len(a), x)` equivale a `a.append(x)`.

`list.`**`remove`**`(`*x*`)`

Quita el primer ítem de la lista cuyo valor sea *x*. Lanza un `ValueError` si no existe tal ítem.

`list.`**`pop`**`(`[*i*]`)`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an `IndexError` if the list is empty or the index is outside the list range.

`list.`**`clear`**`()`

Remove all items from the list. Similar to `del a[:]`.

`list.`**`index`**`(`*x*[, *start*[, *end*]]`)`

Return zero-based index of the first occurrence of *x* in the list. Raises a `ValueError` if there is no such item.

Los argumentos opcionales *start* y *end* son interpretados como la notación de rebanadas y se usan para limitar la búsqueda a un segmento particular de la lista. El índice retornado se calcula de manera relativa al inicio de la secuencia completa en lugar de hacerlo con respecto al argumento *start*.

`list.`**`count`**`(`*x*`)`

Retorna el número de veces que *x* aparece en la lista.

`list.sort(*(Keyword-only parameters separator (PEP 3102)), key=None, reverse=False)`

Ordena los elementos de la lista in situ (los argumentos pueden ser usados para personalizar el orden de la lista, ver `sorted()` para su explicación).

`list.reverse()`

Invierte los elementos de la lista in situ.

`list.copy()`

Return a shallow copy of the list. Similar to `a[:]`.

Un ejemplo que usa la mayoría de los métodos de la lista:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting at position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

Quizás hayas notado que métodos como `insert`, `remove` o `sort` que únicamente modifican la lista no tienen un valor de retorno impreso – retornan el valor por defecto `None`.[1] Esto es un principio de diseño para todas las estructuras de datos mutables en Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings and `None` can't be compared to other types. Also, there are some types that don't have a defined ordering relation. For example, `3+4j < 5+7j` isn't a valid comparison.

5.1.1 Usar listas como pilas

Los métodos de lista hacen que resulte muy fácil usar una lista como una pila, donde el último elemento añadido es el primer elemento retirado («último en entrar, primero en salir»). Para agregar un elemento a la cima de la pila, utiliza `append()`. Para retirar un elemento de la cima de la pila, utiliza `pop()` sin un índice explícito. Por ejemplo:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
```

(continúe en la próxima página)

[1] Otros lenguajes podrían retornar un objeto mutado, que permite encadenamiento de métodos como `d->insert("a")->remove("b")->sort();`.

(proviene de la página anterior)

```
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2 Usar listas como colas

También es posible usar una lista como una cola, donde el primer elemento añadido es el primer elemento retirado («primero en entrar, primero en salir»); sin embargo, las listas no son eficientes para este propósito. Agregar y sacar del final de la lista es rápido, pero insertar o sacar del comienzo de una lista es lento (porque todos los otros elementos tienen que ser desplazados en uno).

Para implementar una cola, utiliza `collections.deque` el cual fue diseñado para añadir y quitar de ambas puntas de forma rápida. Por ejemplo:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 Comprensión de listas

Las comprensiones de listas ofrecen una manera concisa de crear listas. Sus usos comunes son para hacer nuevas listas donde cada elemento es el resultado de algunas operaciones aplicadas a cada miembro de otra secuencia o iterable, o para crear un segmento de la secuencia de esos elementos para satisfacer una condición determinada.

Por ejemplo, asumamos que queremos crear una lista de cuadrados, como:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Nótese que esto crea (o sobreescribe) una variable llamada `x` que sigue existiendo luego de que el bucle haya terminado. Podemos calcular la lista de cuadrados sin ningún efecto secundario haciendo:

```
squares = list(map(lambda x: x**2, range(10)))
```

o, un equivalente:

```
squares = [x**2 for x in range(10)]
```

que es más conciso y legible.

Una lista de comprensión consiste de corchetes rodeando una expresión seguida de la declaración `for` y luego cero o más declaraciones `for` o `if`. El resultado será una nueva lista que sale de evaluar la expresión en el contexto de los `for` o `if` que le siguen. Por ejemplo, esta lista de comprensión combina los elementos de dos listas si no son iguales:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

y es equivalente a:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Nótese como el orden de los `for` y `if` es el mismo en ambos pedacitos de código.

Si la expresión es una tupla (como el `(x, y)` en el ejemplo anterior), debe estar entre paréntesis.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Las comprensiones de listas pueden contener expresiones complejas y funciones anidadas:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4 Listas por comprensión anidadas

La expresión inicial de una comprensión de listas puede ser cualquier expresión arbitraria, incluyendo otra comprensión de listas.

Considerá el siguiente ejemplo de una matriz de 3x4 implementada como una lista de tres listas de largo 4:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

La siguiente comprensión de lista transpondrá las filas y columnas:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Como vimos en la sección anterior, la lista de comprensión anidada se evalúa en el contexto del `for` que lo sigue, por lo que este ejemplo equivale a:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

el cual, a la vez, es lo mismo que:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

En el mundo real, deberías preferir funciones predefinidas a declaraciones con flujo complejo. La función `zip()` haría un buen trabajo para este caso de uso:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Ver *Desempaquetando una lista de argumentos* para detalles sobre el asterisco de esta línea.

5.2 La instrucción `del`

Hay una manera de quitar un ítem de una lista dado su índice en lugar de su valor: la instrucción `del`. Esta es diferente del método `pop()`, el cual retorna un valor. La instrucción `del` también puede usarse para quitar secciones de una lista o vaciar la lista completa (lo que hacíamos antes asignando una lista vacía a la rebanada). Por ejemplo:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` puede usarse también para eliminar variables:

```
>>> del a
```

Hacer referencia al nombre `a` de aquí en más es un error (al menos hasta que se le asigne otro valor). Veremos otros usos para `del` más adelante.

5.3 Tuplas y secuencias

Vimos que las listas y cadenas tienen propiedades en común, como el indexado y las operaciones de rebanado. Estas son dos ejemplos de datos de tipo *secuencia* (ver typesseq). Como Python es un lenguaje en evolución, otros datos de tipo secuencia pueden agregarse. Existe otro dato de tipo secuencia estándar: la *tupla*.

Una tupla está formada por un número de valores separados por comas, por ejemplo:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
>>> t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Como puedes ver, en la salida las tuplas siempre se encierran entre paréntesis para que las tuplas anidadas puedan interpretarse correctamente; pueden ingresarse con o sin paréntesis, aunque a menudo los paréntesis son necesarios de todas formas (si la tupla es parte de una expresión más grande). No es posible asignar a los ítems individuales de una tupla, pero sin embargo sí se puede crear tuplas que contengan objetos mutables, como las listas.

A pesar de que las tuplas puedan parecerse a las listas, frecuentemente se utilizan en distintas situaciones y para distintos propósitos. Las tuplas son *immutable* y normalmente contienen una secuencia heterogénea de elementos que son accedidos al desempaquetar (ver más adelante en esta sección) o indizar (o incluso acceder por atributo en el caso de las `namedtuples`). Las listas son *mutable*, y sus elementos son normalmente homogéneos y se acceden iterando a la lista.

Un problema particular es la construcción de tuplas que contengan 0 o 1 ítem: la sintaxis presenta algunas peculiaridades para estos casos. Las tuplas vacías se construyen mediante un par de paréntesis vacío; una tupla con un ítem se construye poniendo una coma a continuación del valor (no alcanza con encerrar un único valor entre paréntesis). Feo, pero efectivo. Por ejemplo:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

La declaración `t = 12345, 54321, 'hola!'` es un ejemplo de *empaquetado de tuplas*: los valores `12345`, `54321` y `'hola!'` se empaquetan juntos en una tupla. La operación inversa también es posible:

```
>>> x, y, z = t
```

Esto se llama, apropiadamente, *desempaquetado de secuencias*, y funciona para cualquier secuencia en el lado derecho del igual. El desempaquetado de secuencias requiere que la cantidad de variables a la izquierda del signo igual sea el tamaño de la secuencia. Nótese que la asignación múltiple es en realidad sólo una combinación de empaquetado de tuplas y desempaquetado de secuencias.

5.4 Conjuntos

Python también incluye un tipo de dato para *conjuntos*. Un conjunto es una colección no ordenada y sin elementos repetidos. Los usos básicos de éstos incluyen verificación de pertenencia y eliminación de entradas duplicadas. Los conjuntos también soportan operaciones matemáticas como la unión, intersección, diferencia, y diferencia simétrica.

Las llaves o la función `set()` pueden usarse para crear conjuntos. Notá que para crear un conjunto vacío tenés que usar `set()`, no `{}`; esto último crea un diccionario vacío, una estructura de datos que discutiremos en la sección siguiente.

Una pequeña demostración:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
>>>
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

De forma similar a las *comprensiones de listas*, la comprensión de conjuntos está también soportada:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5 Diccionarios

Otro tipo de dato útil incluido en Python es el *diccionario* (ver typesmapping). Los diccionarios se encuentran a veces en otros lenguajes como «memorias asociativas» o «arreglos asociativos». A diferencia de las secuencias, que se indexan mediante un rango numérico, los diccionarios se indexan con *claves*, que pueden ser cualquier tipo inmutable; las cadenas y números siempre pueden ser claves. Las tuplas pueden usarse como claves si solamente contienen cadenas, números o tuplas; si una tupla contiene cualquier objeto mutable directa o indirectamente, no puede usarse

como clave. No podés usar listas como claves, ya que las listas pueden modificarse usando asignación por índice, asignación por sección, o métodos como `append()` y `extend()`.

Es mejor pensar en un diccionario como un conjunto de pares *clave:valor* con el requerimiento de que las claves sean únicas (dentro de un diccionario). Un par de llaves crean un diccionario vacío: `{}`. Colocar una lista de pares clave:valor separada por comas dentro de las llaves añade pares clave:valor iniciales al diccionario; esta es también la forma en que los diccionarios se muestran en la salida.

Las operaciones principales sobre un diccionario son guardar un valor con una clave y extraer ese valor dada la clave. También es posible borrar un par clave:valor con `del`. Si usás una clave que ya está en uso para guardar un valor, el valor que estaba asociado con esa clave se pierde. Es un error extraer un valor usando una clave inexistente.

Ejecutando `list(d)` en un diccionario retornará una lista con todas las claves usadas en el diccionario, en el orden de inserción (si deseas que esté ordenada simplemente usa `sorted(d)` en su lugar). Para comprobar si una clave está en el diccionario usa la palabra clave `in`.

Un pequeño ejemplo de uso de un diccionario:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

El constructor `dict()` crea un diccionario directamente desde secuencias de pares clave-valor:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Además, las comprensiones de diccionarios se pueden usar para crear diccionarios desde expresiones arbitrarias de clave y valor:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Cuando las claves son cadenas simples, a veces resulta más fácil especificar los pares usando argumentos por palabra clave:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6 Técnicas de iteración

Cuando iteramos sobre diccionarios, se pueden obtener al mismo tiempo la clave y su valor correspondiente usando el método `items()`.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

Cuando se itera sobre una secuencia, se puede obtener el índice de posición junto a su valor correspondiente usando la función `enumerate()`.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

Para iterar sobre dos o más secuencias al mismo tiempo, los valores pueden emparejarse con la función `zip()`.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Para iterar sobre una secuencia en orden inverso, se especifica primero la secuencia al derecho y luego se llama a la función `reversed()`.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Para iterar sobre una secuencia ordenada, se utiliza la función `sorted()` la cual retorna una nueva lista ordenada dejando a la original intacta.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

El uso de `set()` en una secuencia elimina los elementos duplicados. El uso de `sorted()` en combinación con `set()` sobre una secuencia es una forma idiomática de recorrer elementos únicos de la secuencia ordenada.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

A veces uno intenta cambiar una lista mientras la está iterando; sin embargo, a menudo es más simple y seguro crear una nueva lista:

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7 Más acerca de condiciones

Las condiciones usadas en las instrucciones `while` e `if` pueden contener cualquier operador, no sólo comparaciones.

Los operadores de comparación `in` y `not in` verifican si un valor ocurre (o no ocurre) en una secuencia. Los operadores `is` e `is not` comparan si dos objetos son realmente el mismo objeto. Todos los operadores de comparación tienen la misma prioridad, que es menor que la de todos los operadores numéricos.

Las comparaciones pueden encadenarse. Por ejemplo, `a < b == c` verifica si `a` es menor que `b` y además si `b` es igual a `c`.

Las comparaciones pueden combinarse mediante los operadores booleanos `and` y `or`, y el resultado de una comparación (o de cualquier otra expresión booleana) puede negarse con `not`. Estos tienen prioridades menores que los operadores de comparación; entre ellos `not` tiene la mayor prioridad y `or` la menor, o sea que `A and not B or C` equivale a `(A and (not B)) or C`. Como siempre, los paréntesis pueden usarse para expresar la composición deseada.

Los operadores booleanos `and` y `or` son los llamados operadores *cortocircuito*: sus argumentos se evalúan de izquierda a derecha, y la evaluación se detiene en el momento en que se determina su resultado. Por ejemplo, si `A` y `C` son verdaderas pero `B` es falsa, en `A and B and C` no se evalúa la expresión `C`. Cuando se usa como un valor general y no como un booleano, el valor retornado de un operador cortocircuito es el último argumento evaluado.

Es posible asignar el resultado de una comparación u otra expresión booleana a una variable. Por ejemplo,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Nótese que en Python, a diferencia de C, asignaciones dentro de expresiones deben realizarse explícitamente con el operador walrus `:=`. Esto soluciona algunos problemas comunes encontrados en C: escribiendo = en una expresión cuando se intentaba escribir ==.

5.8 Comparando secuencias y otros tipos

Las secuencias pueden compararse con otros objetos del mismo tipo de secuencia. La comparación usa orden *lexicográfico*: primero se comparan los dos primeros ítems, si son diferentes esto ya determina el resultado de la comparación; si son iguales, se comparan los siguientes dos ítems, y así sucesivamente hasta llegar al final de alguna de las secuencias. Si dos ítems a comparar son ambos secuencias del mismo tipo, la comparación lexicográfica es recursiva. Si todos los ítems de dos secuencias resultan iguales, se considera que las secuencias son iguales. Si una secuencia es la parte inicial de la otra, la secuencia más corta es la más pequeña. El orden lexicográfico de las cadenas de caracteres utiliza el punto de código Unicode para ordenar caracteres individuales. Algunos ejemplos de comparación entre secuencias del mismo tipo:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Observá que comparar objetos de diferentes tipos con < o > es legal siempre y cuando los objetos tenga los métodos de comparación apropiados. Por ejemplo, los tipos de números mezclados son comparados de acuerdo a su valor numérico, o sea 0 es igual a 0.0, etc. Si no es el caso, en lugar de proveer un ordenamiento arbitrario, el intérprete lanzará una excepción `TypeError`.

Notas al pie

CAPÍTULO 6

Módulos

Si sales del intérprete de Python y vuelves a entrar, las definiciones que habías hecho (funciones y variables) se pierden. Por lo tanto, si quieres escribir un programa más o menos largo, es mejor que utilices un editor de texto para preparar la entrada para el intérprete y ejecutarlo con ese archivo como entrada. Esto se conoce como crear un *script*. A medida que tu programa crezca, quizás quieras separarlo en varios archivos para que el mantenimiento sea más sencillo. Quizás también quieras usar una función útil que has escrito en distintos programas sin copiar su definición en cada programa.

Para soportar esto, Python tiene una manera de poner definiciones en un archivo y usarlos en un script o en una instancia del intérprete. Este tipo de ficheros se llama *módulo*; las definiciones de un módulo pueden ser *importadas* a otros módulos o al módulo *principal* (la colección de variables a las que tienes acceso en un script ejecutado en el nivel superior y en el modo calculadora).

Un módulo es un fichero conteniendo definiciones y declaraciones de Python. El nombre de archivo es el nombre del módulo con el sufijo `.py` agregado. Dentro de un módulo, el nombre del mismo módulo (como cadena) está disponible en el valor de la variable global `__name__`. Por ejemplo, utiliza tu editor de texto favorito para crear un archivo llamado `fibo.py` en el directorio actual, con el siguiente contenido:

```
# Fibonacci numbers module

def fib(n):
    """Write Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):
    """Return Fibonacci series up to n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Ahora entra en el intérprete de Python e importa este modulo con el siguiente comando:

```
>>> import fibo
```

Esto no añade los nombres de las funciones definidas en `fibo` directamente al actual *namespace* (ver *Ámbitos y espacios de nombres en Python* para más detalles); sólo añade el nombre del módulo `fibo` allí. Usando el nombre del módulo puedes acceder a las funciones:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Si pretendes utilizar una función frecuentemente puedes asignarla a un nombre local:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Más sobre los módulos

Un módulo puede contener tanto declaraciones ejecutables como definiciones de funciones. Estas declaraciones están pensadas para inicializar el módulo. Se ejecutan únicamente la *primera* vez que el módulo se encuentra en una declaración import.[1] (También se ejecutan si el archivo se ejecuta como script.)

Cada módulo tiene su propio espacio de nombres privado, que es utilizado como espacio de nombres global por todas las funciones definidas en el módulo. De este modo, el autor de un módulo puede utilizar variables globales en el módulo sin preocuparse por choques accidentales con las variables globales de un usuario. Por otro lado, si sabes lo que estás haciendo puedes tocar las variables globales de un módulo con la misma notación que se utiliza para referirse a sus funciones, `modname.itemname`.

Los módulos pueden importar otros módulos. Es costumbre pero no obligatorio ubicar todas las declaraciones `import` al principio del módulo (o script, para el caso). Los nombres de los módulos importados, si se colocan en el nivel superior de un módulo (fuera de cualquier función o clase), se añaden al espacio de nombres global del módulo.

Hay una variante de la declaración `import` que importa los nombres de un módulo directamente al espacio de nombres del módulo que hace la importación. Por ejemplo:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Esto no introduce en el espacio de nombres local el nombre del módulo desde el cual se está importando (por lo tanto, en el ejemplo, `fibo` no esta definido).

Hay incluso una variante para importar todos los nombres que un módulo define:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Esto importa todos los nombres excepto los que inician con un guión bajo (_). La mayoría de las veces los programadores de Python no usan esto ya que introduce en el intérprete un conjunto de nombres desconocido, posiblemente escondiendo algunas de las definiciones previas.

Nótese que en general la práctica de importar `*` de un módulo o paquete está muy mal vista, ya que frecuentemente genera código poco legible. Sin embargo, está bien usarlo para ahorrar tecleo en sesiones interactivas.

[1] De hecho, las definiciones de funciones también son «declaraciones» que se «ejecutan»; la ejecución de una definición de función a nivel de módulo, añade el nombre de la función en el espacio de nombres global del módulo.

Si el nombre del módulo es seguido por `as`, el nombre siguiendo `as` queda ligado directamente al módulo importado.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Esto es básicamente importar el módulo de la misma forma que se haría con `import fibo`, con la única diferencia en que se encuentra accesible como `fib`.

También se puede utilizar cuando se utiliza `from` con efectos similares:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Nota

Por razones de eficiencia, cada módulo es importado solo una vez por sesión del intérprete. Por lo tanto, si cambias tus módulos, debes reiniciar el interprete – ó, si es un solo módulo que quieres probar de forma interactiva, usa `importlib.reload()`, por ejemplo: `import importlib; importlib.reload(modulename)`.

6.1.1 Ejecutar módulos como scripts

Cuando ejecutes un módulo de Python con

```
python fibo.py <arguments>
```

el código en el módulo será ejecutado, tal como si lo hubieses importado, pero con `__name__` con el valor de `"__main__"`. Eso significa que agregando este código al final de tu módulo:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

puedes hacer que el archivo sea utilizable tanto como script, como módulo importable, porque el código que analiza la línea de órdenes sólo se ejecuta si el módulo es ejecutado como archivo principal:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

Si el módulo se importa, ese código no se ejecuta:

```
>>> import fibo
>>>
```

Esto es frecuentemente usado para proveer al módulo una interfaz de usuario conveniente, o para fines de prueba (ejecutar el módulo como un script que ejecuta un conjunto de pruebas).

6.1.2 El camino de búsqueda de los módulos

Cuando se importa un módulo llamado `spam`, el intérprete busca primero por un módulo con ese nombre que esté integrado en el intérprete. Estos nombres de módulos están listados en `sys.builtin_module_names`. Si no lo encuentra, entonces busca un archivo llamado `spam.py` en una lista de directorios especificada por la variable `sys.path`. `sys.path` se inicializa con las siguientes ubicaciones:

- El directorio que contiene el script de entrada (o el directorio actual cuando no se especifica archivo).
- `PYTHONPATH` (una lista de nombres de directorios, con la misma sintaxis que la variable de la terminal `PATH`).

- El valor predeterminado dependiente de la instalación (por convención incluye un directorio `site-packages`, manejado por el módulo `site`).

Más detalles en sys-path-init.

> **Nota**
>
> En los sistemas de archivo que soportan enlaces simbólicos, el directorio que contiene el script de entrada es calculado luego de seguir el enlace simbólico. En otras palabras, el directorio que contiene el enlace simbólico **no** es agregado al camino de búsqueda del módulo.

Luego de la inicialización, los programas Python pueden modificar `sys.path`. El directorio que contiene el script que se está ejecutando se ubica al principio de la búsqueda, adelante de la biblioteca estándar. Esto significa que se cargarán scripts en ese directorio en lugar de módulos de la biblioteca estándar con el mismo nombre. Esto es un error a menos que se esté reemplazando intencionalmente. Mirá la sección *Módulos estándar* para más información.

6.1.3 Archivos «compilados» de Python

Para acelerar la carga de módulos, Python cachea las versiones compiladas de cada módulo en el directorio `__pycache__` bajo el nombre `module.`*`version`*`.pyc`, dónde la versión codifica el formato del archivo compilado; generalmente contiene el número de versión de Python. Por ejemplo, en CPython *release* 3.3 la versión compilada de spam.py sería cacheada como `__pycache__/spam.cpython-33.pyc`. Este convención de nombre permite compilar módulos desde diferentes *releases* y versiones de Python para coexistir.

Python chequea la fecha de modificación de la fuente contra la versión compilada para ver si esta es obsoleta y necesita ser recompilada. Esto es un proceso completamente automático. También, los módulos compilados son independientes de la plataforma, así que la misma biblioteca puede ser compartida a través de sistemas con diferentes arquitecturas.

Python no chequea el caché en dos circunstancias. Primero, siempre recompila y no graba el resultado del módulo que es cargado directamente desde la línea de comando. Segundo, no chequea el caché si no hay módulo fuente. Para soportar una distribución sin fuente (solo compilada), el módulo compilado debe estar en el directorio origen, y no debe haber un módulo fuente.

Algunos consejos para expertos:

- Puedes usar los modificadores `-O` o `-OO` en el comando de Python para reducir el tamaño del módulo compilado. El modificador `-O` remueve las declaraciones *assert*, el modificador `-OO` remueve declaraciones *assert* y cadenas *__doc__*. Dado que algunos programas pueden confiar en tenerlos disponibles, solo deberías usar esta opción si conoces lo que estás haciendo. Los módulos «optimizados» tienen una etiqueta `opt-` y generalmente son mas pequeños. *Releases* futuras pueden cambiar los efectos de la optimización.
- Un programa no se ejecuta mas rápido cuando es leído de un archivo `.pyc` que cuando es leído de un archivo `.py`; la única cosa que es mas rápida en los archivos `.pyc` es la velocidad con la cual son cargados.
- El módulo `compileall` puede crear archivos .pyc para todos los módulos en un directorio.
- Hay mas detalle de este proceso, incluyendo un diagrama de flujo de decisiones, en **PEP 3147**.

6.2 Módulos estándar

Python viene con una biblioteca de módulos estándar, descrita en un documento separado, la Referencia de la Biblioteca de Python (de aquí en más, «Referencia de la Biblioteca»). Algunos módulos se integran en el intérprete; estos proveen acceso a operaciones que no son parte del núcleo del lenguaje pero que sin embargo están integrados, tanto por eficiencia como para proveer acceso a primitivas del sistema operativo, como llamadas al sistema. El conjunto de tales módulos es una opción de configuración que también depende de la plataforma subyacente. Por ejemplo, el módulo `winreg` sólo se provee en sistemas Windows. Un módulo en particular merece algo de atención: `sys`, el que está integrado en todos los intérpretes de Python. Las variables `sys.ps1` y `sys.ps2` definen las cadenas usadas como cursores primarios y secundarios:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

Estas dos variables están solamente definidas si el intérprete está en modo interactivo.

La variable `sys.path` es una lista de cadenas que determinan el camino de búsqueda del intérprete para los módulos. Se inicializa por omisión a un camino tomado de la variable de entorno `PYTHONPATH`, o a un valor predefinido en el intérprete si `PYTHONPATH` no está configurada. Lo puedes modificar usando las operaciones estándar de listas:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 La función `dir()`

La función integrada `dir()` se usa para encontrar qué nombres define un módulo. Retorna una lista ordenada de cadenas:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
 'warnoptions']
```

Sin argumentos, `dir()` lista los nombres que tienes actualmente definidos:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Nótese que lista todos los tipos de nombres: variables, módulos, funciones, etc.

`dir()` no lista los nombres de las funciones y variables integradas. Si quieres una lista de esos, están definidos en el módulo estándar `builtins`:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
 '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
 'zip']
```

6.4 Paquetes

Los Paquetes son una forma de estructurar el espacio de nombres de módulos de Python usando «nombres de módulo con puntos». Por ejemplo, el nombre del módulo `A.B` designa un submódulo `B` en un paquete llamado `A`. Así como el uso de módulos salva a los autores de diferentes módulos de tener que preocuparse por los nombres de las variables globales de los demás, el uso de nombres de módulo con puntos evita que los autores de paquetes multimódulos, como NumPy o Pillow, tengan que preocuparse por los nombres de los módulos de los demás.

Supongamos que quieres designar una colección de módulos (un «paquete») para el manejo uniforme de archivos y datos de sonidos. Hay diferentes formatos de archivos de sonido (normalmente reconocidos por su extensión, por ejemplo: `.wav`, `.aiff`, `.au`), por lo que tienes que crear y mantener una colección siempre creciente de módulos para la conversión entre los distintos formatos de archivos. Hay muchas operaciones diferentes que quizás quieras ejecutar en los datos de sonido (como mezclarlos, añadir eco, aplicar una función ecualizadora, crear un efecto estéreo artificial), por lo que además estarás escribiendo una lista sin fin de módulos para realizar estas operaciones. Aquí hay una posible estructura para tu paquete (expresados en términos de un sistema jerárquico de archivos):

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
```

(continúe en la próxima página)

(proviene de la página anterior)

```
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Al importar el paquete, Python busca a través de los directorios en `sys.path`, buscando el sub-directorio del paquete.

The `__init__.py` files are required to make Python treat directories containing the file as packages (unless using a *namespace package*, a relatively advanced feature). This prevents directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Los usuarios del paquete pueden importar módulos individuales del mismo, por ejemplo:

```
import sound.effects.echo
```

Esto carga el submódulo `sound.effects.echo`. Debe hacerse referencia al mismo con el nombre completo.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Otra alternativa para importar el submódulo es:

```
from sound.effects import echo
```

Esto también carga el submódulo `echo`, y lo deja disponible sin su prefijo de paquete, por lo que puede usarse así:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Otra variación más es importar la función o variable deseadas directamente:

```
from sound.effects.echo import echofilter
```

De nuevo, esto carga el submódulo `echo`, pero deja directamente disponible a la función `echofilter()`:

```
echofilter(input, output, delay=0.7, atten=4)
```

Nótese que al usar `from package import item`, el ítem puede ser tanto un submódulo (o subpaquete) del paquete, o algún otro nombre definido en el paquete, como una función, clase, o variable. La declaración `import` primero verifica si el ítem está definido en el paquete; si no, asume que es un módulo y trata de cargarlo. Si no lo puede encontrar, se genera una excepción `ImportError`.

Por otro lado, cuando se usa la sintaxis como `import item.subitem.subsubitem`, cada ítem excepto el último debe ser un paquete; el mismo puede ser un módulo o un paquete pero no puede ser una clase, función o variable definida en el ítem previo.

6.4.1 Importar * desde un paquete

Ahora, ¿qué sucede cuando el usuario escribe `from sound.effects import *`? Idealmente, uno esperaría que esto de alguna manera vaya al sistema de archivos, encuentre cuales submódulos están presentes en el paquete, y los importe a todos. Esto puede tardar mucho y el importar sub-módulos puede tener efectos secundarios no deseados que sólo deberían ocurrir cuando se importe explícitamente el sub-módulo.

La única solución es que el autor del paquete provea un índice explícito del paquete. La declaración `import` usa la siguiente convención: si el código del `__init__.py` de un paquete define una lista llamada `__all__`, se toma como la lista de los nombres de módulos que deberían ser importados cuando se hace `from package import *`. Es tarea del autor del paquete mantener actualizada esta lista cuando se libera una nueva versión del paquete. Los autores de paquetes podrían decidir no soportarlo, si no ven un uso para importar * en sus paquetes. Por ejemplo, el archivo `sound/effects/__init__.py` podría contener el siguiente código:

```
__all__ = ["echo", "surround", "reverse"]
```

Esto significaría que `from sound.effects import *` importaría esos tres submódulos del paquete `sound.effects`.

Ten en cuenta que los submódulos pueden quedar ocultos por nombres definidos localmente. Por ejemplo, si agregaste una función llamada `reverse` al archivo `sound/effects/__init__.py`, `from sound.effects import *` solo importaría los dos submódulos `echo` y `surround`, pero *no* el submódulo `reverse` porque queda oculto por la función `reverse` definida localmente:

```
__all__ = [
    "echo",      # refers to the 'echo.py' file
    "surround",  # refers to the 'surround.py' file
    "reverse",   # !!! refers to the 'reverse' function now !!!
]

def reverse(msg: str):  # <-- this name shadows the 'reverse.py' submodule
    return msg[::-1]    #     in the case of a 'from sound.effects import *'
```

Si no se define `__all__`, la declaración `from sound.effects import *` *no* importa todos los submódulos del paquete `sound.effects` al espacio de nombres actual; sólo se asegura que se haya importado el paquete `sound.effects` (posiblemente ejecutando algún código de inicialización que haya en `__init__.py`) y luego importa aquellos nombres que estén definidos en el paquete. Esto incluye cualquier nombre definido (y submódulos explícitamente cargados) por `__init__.py`. También incluye cualquier submódulo del paquete que pudiera haber sido explícitamente cargado por declaraciones `import` previas. Considere este código:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

En este ejemplo, los módulos `echo` y `surround` se importan en el espacio de nombre actual porque están definidos en el paquete `sound.effects` cuando se ejecuta la declaración `from...import`. (Esto también funciona cuando se define `__all__`).

A pesar de que ciertos módulos están diseñados para exportar solo nombres que siguen ciertos patrones cuando uses `import *`, también se considera una mala práctica en código de producción.

Recuerda, ¡no hay nada malo al usar `from package import specific_submodule`! De hecho, esta es la notación recomendada a menos que el módulo que importamos necesite usar submódulos con el mismo nombre desde un paquete diferente.

6.4.2 Referencias internas en paquetes

Cuando se estructuran los paquetes en sub-paquetes (como en el ejemplo `sound`), puedes usar imports absolutos para referirte a submódulos de paquetes hermanos. Por ejemplo, si el módulo `sound.filters.vocoder` necesita usar el módulo `echo` en el paquete `sound.effects`, puede hacer `from sound.effects import echo`.

También puedes escribir imports relativos con la forma `from module import name`. Estos imports usan puntos adelante para indicar los paquetes actuales o paquetes padres involucrados en el import relativo. En el ejemplo `surround`, podrías hacer:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that relative imports are based on the name of the current module's package. Since the main module does not have a package, modules intended for use as the main module of a Python application must always use absolute imports.

6.4.3 Paquetes en múltiples directorios

Packages support one more special attribute, `__path__`. This is initialized to be a *sequence* of strings containing the name of the directory holding the package's `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

Aunque esta característica no se necesita frecuentemente, puede usarse para extender el conjunto de módulos que se encuentran en el paquete.

Notas al pie

CAPÍTULO 7

Entrada y salida

Hay diferentes métodos de presentar la salida de un programa; los datos pueden ser impresos de una forma legible por humanos, o escritos a un archivo para uso futuro. Este capítulo discutirá algunas de las posibilidades.

7.1 Formateo elegante de la salida

Hasta ahora encontramos dos maneras de escribir valores: *declaraciones de expresión* y la función `print()`. (Una tercera manera es usando el método `write()` de los objetos tipo archivo; el archivo de salida estándar puede referenciarse como `sys.stdout`. Mirá la Referencia de la Biblioteca para más información sobre esto).

A menudo se querrá tener más control sobre el formato de la salida, y no simplemente imprimir valores separados por espacios. Para ello, hay varias maneras de dar formato a la salida.

- Para usar *literales de cadena formateados*, comience una cadena con `f` o `F` antes de la comilla de apertura o comillas triples. Dentro de esta cadena, se puede escribir una expresión de Python entre los caracteres `{` y `}` que pueden hacer referencia a variables o valores literales.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted. In the following code block there are two examples of how to format variables:

```
>>> yes_votes = 42_572_654
>>> total_votes = 85_705_149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

  Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints `percentage` multiplied by 100, with 2 decimal places and followed by a percent sign (see formatspec for details).

- Por último, puede realizar todo el control de cadenas usted mismo mediante operaciones de concatenación y segmentación de cadenas para crear cualquier diseño que se pueda imaginar. El tipo de cadena tiene algunos métodos que realizan operaciones útiles para rellenar cadenas a un ancho de columna determinado.

Cuando no necesita una salida elegante, pero solo desea una visualización rápida de algunas variables con fines de depuración, puede convertir cualquier valor en una cadena con las funciones `repr()` o `str()`.

La función `str()` retorna representaciones de los valores que son bastante legibles por humanos, mientras que `repr()` genera representaciones que pueden ser leídas por el intérprete (o forzarían un `SyntaxError` si no hay sintaxis equivalente). Para objetos que no tienen una representación en particular para consumo humano, `str()` retornará el mismo valor que `repr()`. Muchos valores, como números o estructuras como listas y diccionarios, tienen la misma representación usando cualquiera de las dos funciones. Las cadenas, en particular, tienen dos representaciones distintas.

Algunos ejemplos:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
>>> repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

The `string` module contains support for a simple templating approach based upon regular expressions, via `string.Template`. This offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary. This syntax is easy to use, although it offers much less control for formatting.

7.1.1 Formatear cadenas literales

Literales de cadena formateados (también llamados f-strings para abreviar) le permiten incluir el valor de las expresiones de Python dentro de una cadena prefijando la cadena con `f` o `F` y escribiendo expresiones como `{expresion}`.

La expresión puede ir seguida de un especificador de formato opcional . Esto permite un mayor control sobre cómo se formatea el valor. En el ejemplo siguiente se redondea pi a tres lugares después del decimal:

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

Pasar un entero después de `':'` hará que ese campo sea un número mínimo de caracteres de ancho. Esto es útil para hacer que las columnas se alineen.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

Se pueden utilizar otros modificadores para convertir el valor antes de formatearlo. `'!a'` se aplica `ascii()`, `'!s'` se aplica `str()`, y `'!r'` se aplica `repr()`:

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

El especificador = puede utilizarse para expandir una expresión al texto de la expresión, un signo igual y, a continuación, la representación de la expresión evaluada:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

Véase expresiones auto-documentadas para más información en el especificador =. Para obtener una referencia sobre estas especificaciones de formato, consulte la guía de referencia para formatspec.

7.1.2 El método format() de cadenas

El uso básico del método `str.format()` es como esto:

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

Las llaves y caracteres dentro de las mismas (llamados campos de formato) son reemplazadas con los objetos pasados en el método `str.format()`. Un número en las llaves se refiere a la posición del objeto pasado en el método `str.format()`.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

Si se usan argumentos nombrados en el método `str.format()`, sus valores se referencian usando el nombre del argumento.

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

Se pueden combinar arbitrariamente argumentos posicionales y nombrados:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
...                                                   other='Georg'))
The story of Bill, Manfred, and Georg.
```

Si tiene una cadena de caracteres de formato realmente larga que no desea dividir, sería bueno si pudiera hacer referencia a las variables que se formatearán por nombre en lugar de por posición. Esto se puede hacer simplemente pasando el diccionario y usando corchetes `'[]'` para acceder a las claves.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Esto se podría hacer, también, pasando el diccionario `table` como argumentos por palabra clave con la notación "**".

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f'{k}: ' + '{' + k +'};' for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

Como ejemplo, las siguientes líneas producen un conjunto ordenado de columnas que dan enteros y sus cuadrados y cubos:

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Para una completa descripción del formateo de cadenas con `str.format()`, ver string-formatting.

7.1.3 Formateo manual de cadenas

Aquí está la misma tabla de cuadrados y cubos, formateados manualmente:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Nótese que el espacio existente entre cada columna es añadido debido a como funciona `print()`: siempre añade espacios entre sus argumentos.)

El método `str.rjust()` de los objetos cadena justifica a la derecha en un campo de anchura predeterminada rellenando con espacios a la izquierda. Métodos similares a este son `str.ljust()` y `str.center()`. Estos métodos no escriben nada, simplemente retornan una nueva cadena. Si la cadena de entrada es demasiado larga no la truncarán

sino que la retornarán sin cambios; esto desordenará la disposición de la columna que es, normalmente, mejor que la alternativa, la cual podría falsear un valor. (Si realmente deseas truncar siempre puedes añadir una operación de rebanado, como en `x.ljust(n)[:n]`.)

Hay otro método, `str.zfill()`, el cual rellena una cadena numérica a la izquierda con ceros. Entiende signos positivos y negativos:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

7.1.4 Viejo formateo de cadenas

The % operator (modulo) can also be used for string formatting. Given `format % values` (where *format* is a string), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. This operation is commonly known as string interpolation. For example:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

Podés encontrar más información en la sección old-string-formatting.

7.2 Leyendo y escribiendo archivos

La función `open()` retorna un *file object*, y se usa normalmente con dos argumentos posicionales y un argumento nombrado: `open(nombre_de_archivo, modo, encoding=None)`

```
>>> f = open('workfile', 'w', encoding="utf-8")
```

El primer argumento es una cadena que contiene el nombre del fichero. El segundo argumento es otra cadena que contiene unos pocos caracteres describiendo la forma en que el fichero será usado. *mode* puede ser `'r'` cuando el fichero solo se leerá, `'w'` para solo escritura (un fichero existente con el mismo nombre se borrará) y `'a'` abre el fichero para agregar; cualquier dato que se escribe en el fichero se añade automáticamente al final. `'r+'` abre el fichero tanto para lectura como para escritura. El argumento *mode* es opcional; se asume que se usará `'r'` si se omite.

Normalmente, los ficheros se abren en *modo texto*, es decir, se leen y escriben cadenas desde y hacia el fichero, que están codificadas en una *codificación* específica. Si no se especifica *codificación*, el valor por defecto depende de la plataforma (véase `open()`). Dado que UTF-8 es el estándar moderno de facto, se recomienda `encoding="utf-8"` a menos que sepa que necesita usar una codificación diferente. Añadiendo `'b'` al modo se abre el fichero en *modo binario*. Los datos en modo binario se leen y escriben como objetos `bytes`. No se puede especificar *codificación* al abrir un fichero en modo binario.

Cuando se lee en modo texto, por defecto se convierten los fin de lineas que son específicos a las plataformas (`\n` en Unix, `\r\n` en Windows) a solamente `\n`. Cuando se escribe en modo texto, por defecto se convierten los `\n` a los fin de linea específicos de la plataforma. Este cambio automático está bien para archivos de texto, pero corrompería datos binarios como los de archivos `JPEG` o `EXE`. Asegúrese de usar modo binario cuando lea y escriba tales archivos.

Es una buena práctica usar la declaración `with` cuando manejamos objetos archivo. Tiene la ventaja de que el archivo es cerrado apropiadamente luego de que el bloque termina, incluso si se generó una excepción. También es mucho más corto que escribir los equivalentes bloques `try-finally`

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

Si no está utilizando la palabra clave `with`, entonces debe llamar a `f.close()` para cerrar el archivo y liberar inmediatamente los recursos del sistema utilizados por él.

Advertencia

Llamar a `f.write()` sin usar la palabra clave `with` o sin llamar a `f.close()` **podría** dar como resultado que los argumentos de `f.write()` no se escriban completamente en disco, incluso si el programa se termina correctamente.

Después de que un objeto de archivo es cerrado, ya sea por `with` o llamando a `f.close()`, intentar volver a utilizarlo fallará automáticamente:

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

7.2.1 Métodos de los objetos Archivo

El resto de los ejemplos en esta sección asumirán que ya se creó un objeto archivo llamado `f`.

Para leer el contenido de una archivo utiliza `f.read(size)`, el cual lee alguna cantidad de datos y los retorna como una cadena (en modo texto) o un objeto de bytes (en modo binario). *size* es un argumento numérico opcional. Cuando se omite *size* o es negativo, el contenido entero del archivo será leído y retornado; es tu problema si el archivo es el doble de grande que la memoria de tu máquina. De otra manera, son leídos y retornados como máximo *size* caracteres (en modo texto) o *size* bytes (en modo binario). Si se alcanzó el fin del archivo, `f.read()` retornará una cadena vacía (`''`).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` lee una sola linea del archivo; el carácter de fin de linea (`\n`) se deja al final de la cadena, y sólo se omite en la última linea del archivo si el mismo no termina en un fin de linea. Esto hace que el valor de retorno no sea ambiguo; si `f.readline()` retorna una cadena vacía, es que se alcanzó el fin del archivo, mientras que una linea en blanco es representada por `'\n'`, una cadena conteniendo sólo un único fin de linea.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

Para leer líneas de un archivo, puedes iterar sobre el objeto archivo. Esto es eficiente en memoria, rápido, y conduce a un código más simple:

```
>>> for line in f:
...     print(line, end='')
...
```

(continúe en la próxima página)

(proviene de la página anterior)

```
This is the first line of the file.
Second line of the file
```

Si querés leer todas las líneas de un archivo en una lista también podés usar `list(f)` o `f.readlines()`.

`f.write(cadena)` escribe el contenido de la *cadena* al archivo, retornando la cantidad de caracteres escritos.

```
>>> f.write('This is a test\n')
15
```

Otros tipos de objetos necesitan ser convertidos – tanto a una cadena (en modo texto) o a un objeto de bytes (en modo binario) – antes de escribirlos:

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` retorna un entero que indica la posición actual en el archivo representada como número de bytes desde el comienzo del archivo en modo binario y un número opaco en modo texto.

Para cambiar la posición del objeto archivo, utiliza `f.seek(offset, whence)`. La posición es calculada agregando el *offset* a un punto de referencia; el punto de referencia se selecciona del argumento *whence*. Un valor *whence* de 0 mide desde el comienzo del archivo, 1 usa la posición actual del archivo, y 2 usa el fin del archivo como punto de referencia. *whence* puede omitirse, el valor por defecto es 0, usando el comienzo del archivo como punto de referencia.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

En los archivos de texto (aquellos que se abrieron sin una `b` en el modo), se permiten solamente desplazamientos con `seek` relativos al comienzo (con la excepción de ir justo al final con `seek(0, 2)`) y los únicos valores de *desplazamiento* válidos son aquellos retornados por `f.tell()`, o cero. Cualquier otro valor de *desplazamiento* produce un comportamiento indefinido.

Los objetos archivo tienen algunos métodos más, como `isatty()` y `truncate()` que son usados menos frecuentemente; consultá la Referencia de la Biblioteca para una guía completa sobre los objetos archivo.

7.2.2 Guardar datos estructurados con `json`

Las cadenas pueden fácilmente escribirse y leerse de un archivo. Los números toman algo más de esfuerzo, ya que el método `read()` sólo retorna cadenas, que tendrán que ser pasadas a una función como `int()`, que toma una cadena como `'123'` y retorna su valor numérico 123. Sin embargo, cuando querés guardar tipos de datos más complejos como listas, diccionarios, o instancias de clases, las cosas se ponen más complicadas.

En lugar de tener a los usuarios constantemente escribiendo y debugueando código para guardar tipos de datos complicados, Python te permite usar el popular formato intercambiable de datos llamado JSON (JavaScript Object Notation). El módulo estándar llamado `json` puede tomar datos de Python con una jerarquía, y convertirlo a representaciones de cadena de caracteres; este proceso es llamado *serialización*. Reconstruir los datos desde la representación de cadena de caracteres es llamado *deserialización*. Entre serialización y deserialización, la cadena de caracteres representando el objeto quizás haya sido guardado en un archivo o datos, o enviado a una máquina distante por una conexión de red.

Nota

El formato JSON es comúnmente usado por aplicaciones modernas para permitir el intercambio de datos. Muchos programadores ya están familiarizados con él, lo cual lo convierte en una buena opción para la interoperabilidad.

Si tienes un objeto `x`, puedes ver su representación JSON con una simple línea de código:

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

Otra variante de la función `dumps()`, llamada `dump()`, simplemente serializa el objeto a un *archivo de texto*. Así que, si `f` es un objeto *archivo de texto* abierto para escritura, podemos hacer:

```
json.dump(x, f)
```

Para decodificar un objeto nuevamente, si `f` es un objeto *binary file* o *text file* que fue abierto para lectura:

```
x = json.load(f)
```

Nota

Los archivos JSON deben estar codificados en UTF-8. Utilice `encoding="utf-8"` al abrir un archivo JSON como *text file* tanto para lectura como para escritura.

La simple técnica de serialización puede manejar listas y diccionarios, pero serializar instancias de clases arbitrarias en JSON requiere un poco de esfuerzo extra. La referencia del módulo `json` contiene una explicación de esto.

Ver también

`pickle` - El módulo *pickle*

Contrariamente a *JSON*, *pickle* es un protocolo que permite la serialización de objetos Python arbitrariamente complejos. Como tal, es específico de Python y no se puede utilizar para comunicarse con aplicaciones escritas en otros lenguajes. También es inseguro de forma predeterminada: deserializar los datos de *pickle* procedentes de un origen que no es de confianza puede ejecutar código arbitrario, si los datos fueron creados por un atacante experto.

CAPÍTULO 8

Errores y excepciones

Hasta ahora los mensajes de error apenas habían sido mencionados, pero si has probado los ejemplos anteriores probablemente hayas visto algunos. Hay (al menos) dos tipos diferentes de errores: *errores de sintaxis* y *excepciones*.

8.1 Errores de sintaxis

Los errores de sintaxis, también conocidos como errores de interpretación, son quizás el tipo de queja más común que tenés cuando todavía estás aprendiendo Python:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
               ^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function `print()`, since a colon (`':'`) is missing just before it.

The file name (`<stdin>` in our example) and line number are printed so you know where to look in case the input came from a file.

8.2 Excepciones

Incluso si una declaración o expresión es sintácticamente correcta, puede generar un error cuando se intenta ejecutar. Los errores detectados durante la ejecución se llaman *excepciones*, y no son incondicionalmente fatales: pronto aprenderás a gestionarlos en programas Python. Sin embargo, la mayoría de las excepciones no son gestionadas por el código, y resultan en mensajes de error como los mostrados aquí:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    10 * (1/0)
         ~^~
ZeroDivisionError: division by zero
>>> 4 + spam*3
```

(continúe en la próxima página)

(proviene de la página anterior)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    4 + spam*3
        ^^^^
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    '2' + 2
    ~~~~^~~
TypeError: can only concatenate str (not "int") to str
```

La última línea de los mensajes de error indica qué ha sucedido. Hay excepciones de diferentes tipos, y el tipo se imprime como parte del mensaje: los tipos en el ejemplo son: `ZeroDivisionError`, `NameError` y `TypeError`. La cadena mostrada como tipo de la excepción es el nombre de la excepción predefinida que ha ocurrido. Esto es válido para todas las excepciones predefinidas del intérprete, pero no tiene por que ser así para excepciones definidas por el usuario (aunque es una convención útil). Los nombres de las excepciones estándar son identificadores incorporados al intérprete (no son palabras clave reservadas).

El resto de la línea provee información basado en el tipo de la excepción y qué la causó.

La parte anterior del mensaje de error muestra el contexto donde ocurrió la excepción, en forma de seguimiento de pila. En general, contiene un seguimiento de pila que enumera las líneas de origen; sin embargo, no mostrará las líneas leídas desde la entrada estándar.

bltin-exceptions lista las excepciones predefinidas y sus significados.

8.3 Gestionando excepciones

Es posible escribir programas que gestionen determinadas excepciones. Véase el siguiente ejemplo, que le pide al usuario una entrada hasta que ingrese un entero válido, pero permite al usuario interrumpir el programa (usando `Control-C` o lo que soporte el sistema operativo); nótese que una interrupción generada por el usuario es señalizada generando la excepción `KeyboardInterrupt`.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

La sentencia `try` funciona de la siguiente manera.

- Primero, se ejecuta la cláusula *try* (la(s) linea(s) entre las palabras reservadas `try` y la `except`).
- Si no ocurre ninguna excepción, la cláusula *except* se omite y la ejecución de la cláusula `try` finaliza.
- Si ocurre una excepción durante la ejecución de la cláusula `try`, se omite el resto de la cláusula. Luego, si su tipo coincide con la excepción nombrada después de la palabra clave `except`, se ejecuta la *cláusula except*, y luego la ejecución continúa después del bloque try/except.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

Una declaración `try` puede tener más de una *cláusula except*, para especificar gestores para diferentes excepciones. Como máximo, se ejecutará un gestor. Los gestores solo manejan las excepciones que ocurren en la *cláusula try* correspondiente, no en otros gestores de la misma declaración `try`. Una *cláusula except* puede nombrar múltiples excepciones como una tupla entre paréntesis, por ejemplo:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an *except clause* listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Nótese que si las *cláusulas except* estuvieran invertidas (con `except B` primero), habría impreso B, B, B — se usa la primera *cláusula except* coincidente.

Cuando ocurre una excepción, puede tener un valor asociado, también conocido como el *argumento* de la excepción. La presencia y el tipo de argumento depende del tipo de excepción.

La *cláusula except* puede especificar una variable después del nombre de la excepción. La variable está ligada a la instancia de la excepción, que normalmente tiene un atributo `args` que almacena los argumentos. Por conveniencia, los tipos de excepción incorporados definen `__str__()` para imprimir todos los argumentos sin acceder explícitamente a `.args`.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception type
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

La salida `__str__()` de la excepción se imprime como la última parte ("detalle") del mensaje para las excepciones no gestionadas.

`BaseException` es la clase base común de todas las excepciones. Una de sus subclases, `Exception`, es la clase base de todas las excepciones no fatales. Las excepciones que no son subclases de `Exception` no se suelen manejar,

porque se utilizan para indicar que el programa debe terminar. Entre ellas se incluyen `SystemExit`, que es lanzada por `sys.exit()` y `KeyboardInterrupt`, que se lanza cuando un usuario desea interrumpir el programa.

`Exception` se puede utilizar como un comodín que atrapa (casi) todo. Sin embargo, es una buena práctica ser lo más específico posible con los tipos de excepciones que pretendemos manejar, y permitir que cualquier excepción inesperada se propague.

El patrón más común para gestionar `Exception` es imprimir o registrar la excepción y luego volver a re-lanzarla (permitiendo a un llamador manejar la excepción también):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

La declaración `try` … `except` tiene una *cláusula else* opcional, que, cuando está presente, debe seguir todas las *cláusulas except*. Es útil para el código que debe ejecutarse si la *cláusula try* no lanza una excepción. Por ejemplo:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

El uso de la cláusula `else` es mejor que agregar código adicional en la cláusula `try` porque evita capturar accidentalmente una excepción que no fue generada por el código que está protegido por la declaración `try` … `except`.

Los gestores de excepciones no sólo gestionan excepciones que ocurren inmediatamente en la *cláusula try*, sino también aquellas que ocurren dentro de funciones que son llamadas (incluso indirectamente) en la *cláusula try*. Por ejemplo:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

8.4 Lanzando excepciones

La declaración `raise` permite al programador forzar a que ocurra una excepción específica. Por ejemplo:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise NameError('HiThere')
NameError: HiThere
```

El único argumento de `raise` indica la excepción a lanzar. Debe ser una instancia de excepción o una clase de excepción (una clase que derive de `BaseException`, como `Exception` o una de sus subclases). Si se pasa una clase de excepción, se instanciará implícitamente llamando a su constructor sin argumentos:

```
raise ValueError  # shorthand for 'raise ValueError()'
```

Si es necesario determinar si una excepción fue lanzada pero sin intención de gestionarla, una versión simplificada de la instrucción `raise` te permite relanzarla:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise NameError('HiThere')
NameError: HiThere
```

8.5 Encadenamiento de excepciones

Si se produce una excepción no gestionada dentro de una sección `except`, se le adjuntará la excepción que se está gestionando y se incluirá en el mensaje de error:

```
>>> try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    open("database.sqlite")
    ~~~~^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

Para indicar que una excepción es consecuencia directa de otra, la sentencia `raise` permite una cláusula opcional `from`:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

Esto puede resultar útil cuando está transformando excepciones. Por ejemplo:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    func()
    ~~~~^^
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError('Failed to open database') from exc
RuntimeError: Failed to open database
```

También permite deshabilitar el encadenamiento automático de excepciones utilizando el modismo `from None`:

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError from None
RuntimeError
```

Para obtener más información sobre la mecánica del encadenamiento, consulte bltin-exceptions.

8.6 Excepciones definidas por el usuario

Los programas pueden nombrar sus propias excepciones creando una nueva clase excepción (mirá *Clases* para más información sobre las clases de Python). Las excepciones, típicamente, deberán derivar de la clase `Exception`, directa o indirectamente.

Las clases de Excepción pueden ser definidas de la misma forma que cualquier otra clase, pero es habitual mantenerlas lo más simples posible, a menudo ofreciendo solo un número de atributos con información sobre el error que leerán los gestores de la excepción.

La mayoría de las excepciones se definen con nombres acabados en «Error», de manera similar a la nomenclatura de las excepciones estándar.

Muchos módulos estándar definen sus propias excepciones para reportar errores que pueden ocurrir en funciones propias.

8.7 Definiendo acciones de limpieza

La declaración `try` tiene otra cláusula opcional cuyo propósito es definir acciones de limpieza que serán ejecutadas bajo ciertas circunstancias. Por ejemplo:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise KeyboardInterrupt
KeyboardInterrupt
```

Si una cláusula `finally` está presente, el bloque `finally` se ejecutará al final antes de que todo el bloque `try` se complete. La cláusula `finally` se ejecuta independientemente de que la cláusula `try` produzca o no una excepción. Los siguientes puntos explican casos más complejos en los que se produce una excepción:

- Si ocurre una excepción durante la ejecución de la cláusula `try`, la excepción podría ser gestionada por una cláusula `except`. Si la excepción no es gestionada por una cláusula `except`, la excepción es relanzada después de que se ejecute el bloque de la cláusula `finally`.
- Podría aparecer una excepción durante la ejecución de una cláusula `except` o `else`. De nuevo, la excepción será relanzada después de que el bloque de la cláusula `finally` se ejecute.
- If the `finally` clause executes a `break`, `continue` or `return` statement, exceptions are not re-raised. This can be confusing and is therefore discouraged. From version 3.14 the compiler emits a `SyntaxWarning` for it (see **PEP 765**).
- Si el bloque `try` llega a una sentencia `break`, `continue` o `return`, la cláusula `finally` se ejecutará justo antes de la ejecución de dicha sentencia.
- If a `finally` clause includes a `return` statement, the returned value will be the one from the `finally` clause's `return` statement, not the value from the `try` clause's `return` statement. This can be confusing and is therefore discouraged. From version 3.14 the compiler emits a `SyntaxWarning` for it (see **PEP 765**).

Por ejemplo:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

Un ejemplo más complicado:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    divide("2", "1")
    ~~~~~~^^^^^^^^^^
  File "<stdin>", line 3, in divide
    result = x / y
             ~~^~~
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Como se puede ver, la cláusula `finally` siempre se ejecuta. La excepción `TypeError` lanzada al dividir dos cadenas de texto no es gestionado por la cláusula `except` y por lo tanto es relanzada luego de que se ejecuta la cláusula `finally`.

En aplicaciones reales, la cláusula `finally` es útil para liberar recursos externos (como archivos o conexiones de red), sin importar si el uso del recurso fue exitoso.

8.8 Acciones predefinidas de limpieza

Algunos objetos definen acciones de limpieza estándar para llevar a cabo cuando el objeto ya no necesario, independientemente de que las operaciones sobre el objeto hayan sido exitosas o no. Véase el siguiente ejemplo, que intenta abrir un archivo e imprimir su contenido en la pantalla.

```
for line in open("myfile.txt"):
    print(line, end="")
```

El problema con este código es que deja el archivo abierto por un periodo de tiempo indeterminado luego de que esta parte termine de ejecutarse. Esto no es un problema en *scripts* simples, pero puede ser un problema en aplicaciones más grandes. La declaración `with` permite que los objetos como archivos sean usados de una forma que asegure que siempre se los libera rápido y en forma correcta.:

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

Una vez que la declaración se ejecuta, el fichero *f* siempre se cierra, incluso si aparece algún error durante el procesado de las líneas. Los objetos que, como los ficheros, posean acciones predefinidas de limpieza lo indicarán en su documentación.

8.9 Lanzando y gestionando múltiples excepciones no relacionadas

Hay situaciones en las que es necesario informar de varias excepciones que se han producido. Este es a menudo el caso en los marcos de concurrencia, cuando varias tareas pueden haber fallado en paralelo, pero también hay otros casos de uso en los que es deseable continuar la ejecución y recoger múltiples errores en lugar de lanzar la primera excepción.

El incorporado `ExceptionGroup` envuelve una lista de instancias de excepción para que puedan ser lanzadas juntas. Es una excepción en sí misma, por lo que puede capturarse como cualquier otra excepción.

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
...
>>> f()
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |     f()
  |     ~^^
  |   File "<stdin>", line 3, in f
  |     raise ExceptionGroup('there were problems', excs)
  | ExceptionGroup: there were problems (2 sub-exceptions)
  +-+---------------- 1 ----------------
    | OSError: error 1
    +---------------- 2 ----------------
    | SystemError: error 2
    +------------------------------------
>>> try:
...     f()
... except Exception as e:
...     print(f'caught {type(e)}: e')
...
caught <class 'ExceptionGroup'>: e
>>>
```

Utilizando `except*` en lugar de `except`, podemos manejar selectivamente sólo las excepciones del grupo que coincidan con un determinado tipo. En el siguiente ejemplo, que muestra un grupo de excepciones anidado, cada cláusula `except*` extrae del grupo las excepciones de un tipo determinado, mientras que deja que el resto de excepciones se propaguen a otras cláusulas y, finalmente, se vuelvan a lanzar.

```
>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
...
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |     f()
  |     ~^^
```

(continúe en la próxima página)

(proviene de la página anterior)

```
  |   File "<stdin>", line 2, in f
  |     raise ExceptionGroup(
  |     ...<12 lines>...
  |     )
  | ExceptionGroup: group1 (1 sub-exception)
  +-+---------------- 1 ----------------
    | ExceptionGroup: group2 (1 sub-exception)
    +-+---------------- 1 ----------------
      | RecursionError: 4
      +------------------------------------
>>>
```

Tenga en cuenta que las excepciones anidadas en un grupo de excepciones deben ser instancias, no tipos. Esto se debe a que en la práctica las excepciones serían típicamente las que ya han sido planteadas y capturadas por el programa, siguiendo el siguiente patrón:

```
>>> excs = []
... for test in tests:
...     try:
...         test.run()
...     except Exception as e:
...         excs.append(e)
...
>>> if excs:
...    raise ExceptionGroup("Test Failures", excs)
...
```

8.10 Enriqueciendo excepciones con notas

Cuando se crea una excepción para ser lanzada, normalmente se inicializa con información que describe el error que se ha producido. Hay casos en los que es útil añadir información después de que la excepción haya sido capturada. Para este propósito, las excepciones tienen un método `add_note(note)` que acepta una cadena y la añade a la lista de notas de la excepción. La representación estándar del rastreo incluye todas las notas, en el orden en que fueron añadidas, después de la excepción.

```
>>> try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise TypeError('bad type')
TypeError: bad type
Add some information
Add some more information
>>>
```

Por ejemplo, al recopilar excepciones en un grupo de excepciones, es posible que queramos añadir información de contexto para los errores individuales. A continuación, cada excepción del grupo tiene una nota que indica cuándo se ha producido ese error.

```
>>> def f():
...     raise OSError('operation failed')
```

(continúe en la próxima página)

(proviene de la página anterior)

```
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Happened in Iteration {i+1}')
...         excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |     raise ExceptionGroup('We have some problems', excs)
  | ExceptionGroup: We have some problems (3 sub-exceptions)
  +-+---------------- 1 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 1
    +---------------- 2 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 2
    +---------------- 3 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 3
    +------------------------------------
>>>
```

CAPÍTULO 9

Clases

Las clases proveen una forma de empaquetar datos y funcionalidad juntos. Al crear una nueva clase, se crea un nuevo *tipo* de objeto, permitiendo crear nuevas *instancias* de ese tipo. Cada instancia de clase puede tener atributos adjuntos para mantener su estado. Las instancias de clase también pueden tener métodos (definidos por su clase) para modificar su estado.

Comparado con otros lenguajes de programación, el mecanismo de clases de Python agrega clases con un mínimo de nuevas sintaxis y semánticas. Es una mezcla de los mecanismos de clases encontrados en C++ y Modula-3. Las clases de Python proveen todas las características normales de la Programación Orientada a Objetos: el mecanismo de la herencia de clases permite múltiples clases base, una clase derivada puede sobre escribir cualquier método de su(s) clase(s) base, y un método puede llamar al método de la clase base con el mismo nombre. Los objetos pueden tener una cantidad arbitraria de datos de cualquier tipo. Igual que con los módulos, las clases participan de la naturaleza dinámica de Python: se crean en tiempo de ejecución, y pueden modificarse luego de la creación.

En terminología de C++, normalmente los miembros de las clases (incluyendo los miembros de datos), son *públicos* (excepto ver abajo *Variables privadas*), y todas las funciones miembro son *virtuales*. Como en Modula-3, no hay atajos para hacer referencia a los miembros del objeto desde sus métodos: la función método se declara con un primer argumento explícito que representa al objeto, el cual se provee implícitamente por la llamada. Como en Smalltalk, las clases mismas son objetos. Esto provee una semántica para importar y renombrar. A diferencia de C++ y Modula-3, los tipos de datos integrados pueden usarse como clases base para que el usuario los extienda. También, como en C++ pero a diferencia de Modula-3, la mayoría de los operadores integrados con sintaxis especial (operadores aritméticos, de sub-índice, etc.) pueden volver a ser definidos por instancias de la clase.

(Sin haber una terminología universalmente aceptada sobre clases, haré uso ocasional de términos de Smalltalk y C++. Usaría términos de Modula-3, ya que su semántica orientada a objetos es más cercana a Python que C++, pero no espero que muchos lectores hayan escuchado hablar de él.)

9.1 Unas palabras sobre nombres y objetos

Los objetos tienen individualidad, y múltiples nombres (en muchos ámbitos) pueden vincularse al mismo objeto. Esto se conoce como *aliasing* en otros lenguajes. Normalmente no se aprecia esto a primera vista en Python, y puede ignorarse sin problemas cuando se maneja tipos básicos inmutables (números, cadenas, tuplas). Sin embargo, el *aliasing*, o renombrado, tiene un efecto posiblemente sorpresivo sobre la semántica de código Python que involucra objetos mutables como listas, diccionarios, y la mayoría de otros tipos. Esto se usa normalmente para beneficio del programa, ya que los renombres funcionan como punteros en algunos aspectos. Por ejemplo, pasar un objeto es barato ya que la implementación solamente pasa el puntero; y si una función modifica el objeto que fue pasado, el que la llama verá el cambio; esto elimina la necesidad de tener dos formas diferentes de pasar argumentos, como en Pascal.

9.2 Ámbitos y espacios de nombres en Python

Antes de ver clases, primero debo decirte algo acerca de las reglas de ámbito de Python. Las definiciones de clases hacen unos lindos trucos con los espacios de nombres, y necesitás saber cómo funcionan los alcances y espacios de nombres para entender por completo cómo es la cosa. De paso, los conocimientos en este tema son útiles para cualquier programador Python avanzado.

Comencemos con unas definiciones.

Un *espacio de nombres* es una relación de nombres a objetos. Muchos espacios de nombres están implementados en este momento como diccionarios de Python, pero eso no se nota para nada (excepto por el desempeño), y puede cambiar en el futuro. Como ejemplos de espacios de nombres tenés: el conjunto de nombres incluidos (conteniendo funciones como `abs()`, y los nombres de excepciones integradas); los nombres globales en un módulo; y los nombres locales en la invocación a una función. Lo que es importante saber de los espacios de nombres es que no hay relación en absoluto entre los nombres de espacios de nombres distintos; por ejemplo, dos módulos diferentes pueden tener definidos los dos una función `maximizar` sin confusión; los usuarios de los módulos deben usar el nombre del módulo como prefijo.

Por cierto, yo uso la palabra *atributo* para cualquier cosa después de un punto; por ejemplo, en la expresión `z.real`, `real` es un atributo del objeto `z`. Estrictamente hablando, las referencias a nombres en módulos son referencias a atributos: en la expresión `modulo.funcion`, `modulo` es un objeto módulo y `funcion` es un atributo de éste. En este caso hay una relación directa entre los atributos del módulo y los nombres globales definidos en el módulo: ¡están compartiendo el mismo espacio de nombres![1]

Los atributos pueden ser de solo lectura, o de escritura. En el segundo caso, es posible asignar valores a los atributos. Los atributos de módulo son modificables: puedes escribir `modname.the_answer = 42`. Los atributos modificables también se pueden eliminar con la declaración `del`. Por ejemplo, `del modname.the_answer` eliminará el atributo `the_answer` del objeto nombrado por `modname`.

Los espacios de nombres se crean en diferentes momentos y con diferentes tiempos de vida. El espacio de nombres que contiene los nombres incluidos se crea cuando se inicia el intérprete, y nunca se borra. El espacio de nombres global de un módulo se crea cuando se lee la definición de un módulo; normalmente, los espacios de nombres de módulos también duran hasta que el intérprete finaliza. Las instrucciones ejecutadas en el nivel de llamadas superior del intérprete, ya sea desde un script o interactivamente, se consideran parte del módulo llamado `__main__`, por lo tanto tienen su propio espacio de nombres global. (Los nombres incluidos en realidad también viven en un módulo; este se llama `builtins`.)

El espacio de nombres local a una función se crea cuando la función es llamada, y se elimina cuando la función retorna o lanza una excepción que no se maneje dentro de la función. (Podríamos decir que lo que pasa en realidad es que ese espacio de nombres se «olvida».) Por supuesto, las llamadas recursivas tienen cada una su propio espacio de nombres local.

Un *ámbito* es una región textual de un programa en Python donde un espacio de nombres es accesible directamente. «Accesible directamente» significa que una referencia sin calificar a un nombre intenta encontrar dicho nombre dentro del espacio de nombres.

Aunque los alcances se determinan de forma estática, se utilizan de forma dinámica. En cualquier momento durante la ejecución, hay 3 o 4 ámbitos anidados cuyos espacios de nombres son directamente accesibles:

- el alcance más interno, que es inspeccionado primero, contiene los nombres locales
- los alcances de cualquier función que encierra a otra, son inspeccionados a partir del alcance más cercano, contienen nombres no locales, pero también no globales
- el penúltimo alcance contiene nombres globales del módulo actual
- el alcance más externo (el último inspeccionado) es el espacio de nombres que contiene los nombres integrados

Si un nombre se declara global, entonces todas las referencias y asignaciones se realizan directamente en el ámbito penúltimo que contiene los nombres globales del módulo. Para volver a enlazar variables encontradas fuera del ámbito más interno, se puede utilizar la declaración `nonlocal`; si no se declara nonlocal, esas variables serán de sólo lectura

[1] Except for one thing. Module objects have a secret read-only attribute called `__dict__` which returns the dictionary used to implement the module's namespace; the name `__dict__` is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

(un intento de escribir en una variable de este tipo simplemente creará una *nueva* variable local en el ámbito más interno, dejando sin cambios la variable con el mismo nombre en el ámbito externo).

Habitualmente, el ámbito local referencia los nombres locales de la función actual. Fuera de una función, el ámbito local referencia al mismo espacio de nombres que el ámbito global: el espacio de nombres del módulo. Las definiciones de clases crean un espacio de nombres más en el ámbito local.

Es importante notar que los alcances se determinan textualmente: el ámbito global de una función definida en un módulo es el espacio de nombres de ese módulo, no importa desde dónde o con qué alias se llame a la función. Por otro lado, la búsqueda de nombres se hace dinámicamente, en tiempo de ejecución; sin embargo, la definición del lenguaje está evolucionando a hacer resolución de nombres estáticamente, en tiempo de «compilación», ¡así que no te confíes de la resolución de nombres dinámica! (De hecho, las variables locales ya se determinan estáticamente.)

Una peculiaridad especial de Python es que, si no hay una declaración `global` o `nonlocal` en efecto, las asignaciones a nombres siempre van al ámbito interno. Las asignaciones no copian datos, solamente asocian nombres a objetos. Lo mismo cuando se borra: la declaración `del x` quita la asociación de `x` del espacio de nombres referenciado por el ámbito local. De hecho, todas las operaciones que introducen nuevos nombres usan el ámbito local: en particular, las instrucciones `import` y las definiciones de funciones asocian el módulo o nombre de la función al espacio de nombres en el ámbito local.

La declaración `global` puede usarse para indicar que ciertas variables viven en el ámbito global y deberían reasignarse allí; la declaración `nonlocal` indica que ciertas variables viven en un ámbito encerrado y deberían reasignarse allí.

9.2.1 Ejemplo de ámbitos y espacios de nombre

Este es un ejemplo que muestra como hacer referencia a distintos ámbitos y espacios de nombres, y cómo las declaraciones `global` y `nonlocal` afectan la asignación de variables:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

El resultado del código ejemplo es:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Notá como la asignación *local* (que es el comportamiento normal) no cambió la vinculación de *spam* de *scope_test*. La asignación `nonlocal` cambió la vinculación de *spam* de *scope_test*, y la asignación `global` cambió la vinculación a

nivel de módulo.

También podés ver que no había vinculación para *spam* antes de la asignación `global`.

9.3 Un primer vistazo a las clases

Las clases introducen un poquito de sintaxis nueva, tres nuevos tipos de objetos y algo de semántica nueva.

9.3.1 Sintaxis de definición de clases

La forma más sencilla de definición de una clase se ve así:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Las definiciones de clases, al igual que las definiciones de funciones (instrucciones `def`) deben ejecutarse antes de que tengan efecto alguno. (Es concebible poner una definición de clase dentro de una rama de un `if`, o dentro de una función.)

En la práctica, las declaraciones dentro de una clase son definiciones de funciones, pero otras declaraciones son permitidas, y a veces resultan útiles; veremos esto más adelante. Las definiciones de funciones dentro de una clase normalmente tienen una lista de argumentos peculiar, dictada por las convenciones de invocación de métodos; a esto también lo veremos más adelante.

Cuando se ingresa una definición de clase, se crea un nuevo espacio de nombres, el cual se usa como ámbito local; por lo tanto, todas las asignaciones a variables locales van a este nuevo espacio de nombres. En particular, las definiciones de funciones asocian el nombre de las funciones nuevas allí.

Cuando una definición de clase se finaliza normalmente (al llegar al final) se crea un *objeto clase*. Básicamente, este objeto envuelve los contenidos del espacio de nombres creado por la definición de la clase; aprenderemos más acerca de los objetos clase en la sección siguiente. El ámbito local original (el que tenía efecto justo antes de que ingrese la definición de la clase) es restablecido, y el objeto clase se asocia allí al nombre que se le puso a la clase en el encabezado de su definición (`ClassName` en el ejemplo).

9.3.2 Objetos clase

Los objetos clase soportan dos tipos de operaciones: hacer referencia a atributos e instanciación.

Para *hacer referencia a atributos* se usa la sintaxis estándar de todas las referencias a atributos en Python: `objeto.nombre`. Los nombres de atributo válidos son todos los nombres que estaban en el espacio de nombres de la clase cuando ésta se creó. Por lo tanto, si la definición de la clase es así:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

La *instanciación* de clases usa la notación de funciones. Hacé de cuenta que el objeto de clase es una función sin parámetros que retorna una nueva instancia de la clase. Por ejemplo (para la clase de más arriba):

```
x = MyClass()
```

crea una nueva *instancia* de la clase y asigna este objeto a la variable local `x`.

La operación de instanciación («llamar» a un objeto clase) crea un objeto vacío. Muchas clases necesitan crear objetos con instancias en un estado inicial particular. Por lo tanto una clase puede definir un método especial llamado `__init__()`, de esta forma:

```
def __init__(self):
    self.data = []
```

Cuando una clase define un método `__init__()`, la instanciación de la clase automáticamente invoca a `__init__()` para la instancia recién creada. Entonces, en este ejemplo, una instancia nueva e inicializada se puede obtener haciendo:

```
x = MyClass()
```

Por supuesto, el método `__init__()` puede tener argumentos para mayor flexibilidad. En ese caso, los argumentos que se pasen al operador de instanciación de la clase van a parar al método `__init__()`. Por ejemplo,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Objetos instancia

Ahora, ¿Qué podemos hacer con los objetos instancia? La única operación que es entendida por los objetos instancia es la referencia de atributos. Hay dos tipos de nombres de atributos válidos, atributos de datos y métodos.

Data attributes correspond to «instance variables» in Smalltalk, and to «data members» in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value `16`, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

The other kind of instance attribute reference is a *method*. A method is a function that «belongs to» an object.

Los nombres válidos de métodos de un objeto instancia dependen de su clase. Por definición, todos los atributos de clase que son objetos funciones definen métodos correspondientes de sus instancias. Entonces, en nuestro ejemplo, `x.f` es una referencia a un método válido, dado que `MyClass.f` es una función, pero `x.i` no lo es, dado que `MyClass.i` no lo es. Pero `x.f` no es la misma cosa que `MyClass.f`; es un *objeto método*, no un objeto función.

9.3.4 Objetos método

Generalmente, un método es llamado luego de ser vinculado:

```
x.f()
```

If `x = MyClass()`, as above, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print(xf())
```

continuará imprimiendo `hello world` hasta el fin de los días.

¿Qué sucede exactamente cuando un método es llamado? Debés haber notado que `x.f()` fue llamado más arriba sin ningún argumento, a pesar de que la definición de función de `f()` especificaba un argumento. ¿Qué pasó con ese argumento? Seguramente Python lanza una excepción cuando una función que requiere un argumento es llamada sin ninguno, aún si el argumento no es utilizado…

De hecho, tal vez hayas adivinado la respuesta: lo que tienen de especial los métodos es que el objeto es pasado como el primer argumento de la función. En nuestro ejemplo, la llamada `x.f()` es exactamente equivalente a `MyClass.f(x)`. En general, llamar a un método con una lista de *n* argumentos es equivalente a llamar a la función correspondiente con una lista de argumentos que es creada insertando el objeto del método antes del primer argumento.

In general, methods work as follows. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a function object, references to both the instance object and the function object are packed into a method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

9.3.5 Variables de clase y de instancia

En general, las variables de instancia son para datos únicos de cada instancia y las variables de clase son para atributos y métodos compartidos por todas las instancias de la clase:

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

Como se vio en *Unas palabras sobre nombres y objetos*, los datos compartidos pueden tener efectos inesperados que involucren objetos *mutable* como ser listas y diccionarios. Por ejemplo, la lista *tricks* en el siguiente código no debería ser usada como variable de clase porque una sola lista sería compartida por todos las instancias de *Dog*:

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

El diseño correcto de esta clase sería usando una variable de instancia:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4 Algunas observaciones

Si el mismo nombre de atributo aparece tanto en la instancia como en la clase, la búsqueda del atributo prioriza la instancia:

```
>>> class Warehouse:
...     purpose = 'storage'
...     region = 'west'
...
>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

A los atributos de datos los pueden hacer referencia tanto los métodos como los usuarios («clientes») ordinarios de un objeto. En otras palabras, las clases no se usan para implementar tipos de datos abstractos puros. De hecho, en Python no hay nada que haga cumplir el ocultar datos; todo se basa en convención. (Por otro lado, la implementación de Python, escrita en C, puede ocultar por completo detalles de implementación y el control de acceso a un objeto si es necesario; esto se puede usar en extensiones a Python escritas en C.)

Los clientes deben usar los atributos de datos con cuidado; éstos pueden romper invariantes que mantienen los métodos si pisan los atributos de datos. Observá que los clientes pueden añadir sus propios atributos de datos a una instancia sin afectar la validez de sus métodos, siempre y cuando se eviten conflictos de nombres; de nuevo, una convención de nombres puede ahorrar un montón de dolores de cabeza.

No hay un atajo para hacer referencia a atributos de datos (¡u otros métodos!) desde dentro de un método. A mi

parecer, esto en realidad aumenta la legibilidad de los métodos: no existe posibilidad alguna de confundir variables locales con variables de instancia cuando repasamos un método.

A menudo, el primer argumento de un método se llama `self` (uno mismo). Esto no es nada más que una convención: el nombre `self` no significa nada en especial para Python. Observá que, sin embargo, si no seguís la convención tu código puede resultar menos legible a otros programadores de Python, y puede llegar a pasar que un programa *navegador de clases* pueda escribirse de una manera que dependa de dicha convención.

Cualquier objeto función que es un atributo de clase define un método para instancias de esa clase. No es necesario que el la definición de la función esté textualmente dentro de la definición de la clase: asignando un objeto función a una variable local en la clase también está bien. Por ejemplo:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Ahora `f`, `g` y `h` son todos atributos de la clase `C` que hacen referencia a objetos función, y consecuentemente son todos métodos de las instancias de `C` — `h` siendo exactamente equivalente a `g`. Fijate que esta práctica normalmente sólo sirve para confundir al que lea un programa.

Los métodos pueden llamar a otros métodos de la instancia usando el argumento `self`:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Los métodos pueden hacer referencia a nombres globales de la misma manera que lo hacen las funciones comunes. El ámbito global asociado a un método es el módulo que contiene su definición. (Una clase nunca se usa como un ámbito global). Si bien es raro encontrar una buena razón para usar datos globales en un método, hay muchos usos legítimos del ámbito global: por lo menos, las funciones y módulos importados en el ámbito global pueden usarse por los métodos, al igual que las funciones y clases definidas en él. Habitualmente, la clase que contiene el método está definida en este ámbito global, y en la siguiente sección veremos algunas buenas razones por las que un método querría hacer referencia a su propia clase.

Todo valor es un objeto, y por lo tanto tiene una *clase* (también llamado su *tipo*). Ésta se almacena como `objeto.__class__`.

9.5 Herencia

Por supuesto, una característica del lenguaje no sería digna del nombre «clase» si no soportara herencia. La sintaxis para una definición de clase derivada se ve así:

```
class DerivedClassName(BaseClassName):
    <statement-1>
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    .
    .
    .
    <statement-N>
```

El nombre `BaseClassName` debe estar definido en un ámbito que contenga a la definición de la clase derivada. En el lugar del nombre de la clase base se permiten otras expresiones arbitrarias. Esto puede ser útil, por ejemplo, cuando la clase base está definida en otro módulo:

```
class DerivedClassName(modname.BaseClassName):
```

La ejecución de una definición de clase derivada procede de la misma forma que una clase base. Cuando el objeto clase se construye, se tiene en cuenta a la clase base. Esto se usa para resolver referencias a atributos: si un atributo solicitado no se encuentra en la clase, la búsqueda continúa por la clase base. Esta regla se aplica recursivamente si la clase base misma deriva de alguna otra clase.

No hay nada en especial en la instanciación de clases derivadas: `DerivedClassName()` crea una nueva instancia de la clase. Las referencias a métodos se resuelven de la siguiente manera: se busca el atributo de clase correspondiente, descendiendo por la cadena de clases base si es necesario, y la referencia al método es válida si se entrega un objeto función.

Las clases derivadas pueden redefinir métodos de su clase base. Como los métodos no tienen privilegios especiales cuando llaman a otros métodos del mismo objeto, un método de la clase base que llame a otro método definido en la misma clase base puede terminar llamando a un método de la clase derivada que lo haya redefinido. (Para los programadores de C++: en Python todos los métodos son en efecto `virtuales`.)

Un método redefinido en una clase derivada puede de hecho querer extender en vez de simplemente reemplazar al método de la clase base con el mismo nombre. Hay una manera simple de llamar al método de la clase base directamente: simplemente llamás a `BaseClassName.methodname(self, arguments)`. En ocasiones esto es útil para los clientes también. (Observá que esto sólo funciona si la clase base es accesible como `BaseClassName` en el ámbito global).

Python tiene dos funciones integradas que funcionan con herencia:

- Usar `isinstance()` para verificar el tipo de una instancia: `isinstance(obj, int)` será `True` sólo si `obj.__class__` es `int` o alguna clase derivada de `int`.
- Usar `issubclass()` para verificar la herencia de clases: `issubclass(bool, int)` es `True` ya que `bool` es una subclase de `int`. Sin embargo, `issubclass(float, int)` es `False` ya que `float` no es una subclase de `int`.

9.5.1 Herencia múltiple

Python también soporta una forma de herencia múltiple. Una definición de clase con múltiples clases base se ve así:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Para la mayoría de los propósitos, en los casos más simples, podés pensar en la búsqueda de los atributos heredados de una clase padre como una búsqueda en profundidad, de izquierda a derecha, sin repetir la misma clase cuando está dos veces en la jerarquía. Por lo tanto, si un atributo no se encuentra en `DerivedClassName`, se busca en `Base1`, luego (recursivamente) en las clases base de `Base1`, y solo si no se encuentra allí se lo busca en `Base2`, y así sucesivamente.

En realidad es un poco más complejo que eso; el orden de resolución de métodos cambia dinámicamente para soportar las llamadas cooperativas a `super()`. Este enfoque es conocido en otros lenguajes con herencia múltiple como

«llámese al siguiente método» y es más poderoso que la llamada al superior que se encuentra en lenguajes con sólo herencia simple.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see python_2.3_mro.

9.6 Variables privadas

Las variables «privadas» de instancia, que no pueden accederse excepto desde dentro de un objeto, no existen en Python. Sin embargo, hay una convención que se sigue en la mayoría del código Python: un nombre prefijado con un guión bajo (por ejemplo, `_spam`) debería tratarse como una parte no pública de la API (más allá de que sea una función, un método, o un dato). Debería considerarse un detalle de implementación y que está sujeto a cambios sin aviso.

Ya que hay un caso de uso válido para los identificadores privados de clase (a saber: colisión de nombres con nombres definidos en las subclases), hay un soporte limitado para este mecanismo. Cualquier identificador con la forma `__spam` (al menos dos guiones bajos al principio, como mucho un guión bajo al final) es textualmente reemplazado por `_nombredeclase__spam`, donde `nombredeclase` es el nombre de clase actual al que se le sacan guiones bajos del comienzo (si los tuviera). Se modifica el nombre del identificador sin importar su posición sintáctica, siempre y cuando ocurra dentro de la definición de una clase.

Ver también

The private name mangling specifications for details and special cases.

La modificación de nombres es útil para dejar que las subclases sobreescriban los métodos sin romper las llamadas a los métodos desde la misma clase. Por ejemplo:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

El ejemplo de arriba funcionaría incluso si `MappingSubclass` introdujera un identificador `__update` ya que se reemplaza con `_Mapping__update` en la clase `Mapping` y `_MappingSubclass__update` en la clase `MappingSubclass` respectivamente.

Hay que aclarar que las reglas de modificación de nombres están diseñadas principalmente para evitar accidentes; es

posible acceder o modificar una variable que es considerada como privada. Esto hasta puede resultar útil en circunstancias especiales, tales como en el depurador.

Notar que el código pasado a `exec` o `eval()` no considera que el nombre de clase de la clase que invoca sea la clase actual; esto es similar al efecto de la sentencia `global`, efecto que es de similar manera restringido a código que es compilado en conjunto. La misma restricción aplica a `getattr()`, `setattr()` y `delattr()`, así como cuando se referencia a `__dict__` directamente.

9.7 Detalles y Cuestiones Varias

A veces es útil tener un tipo de datos similar al «registro» de Pascal o la «estructura» de C, que sirva para juntar algunos pocos ítems con nombre. El enfoque idiomático es utilizar `dataclasses` con este propósito:

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int
```

```
>>> john = Employee('john', 'computer lab', 1000)
>>> john.dept
'computer lab'
>>> john.salary
1000
```

Un fragmento de código en Python que espera un tipo de dato abstracto en particular a menudo puede recibir una clase que emule los métodos de ese tipo de dato en su lugar. Por ejemplo, si tienes una función que formatea algunos datos de un objeto de archivo, puedes definir una clase con los métodos `read()` y `readline()` que obtienen los datos de un búfer de cadena en su lugar, y pasarla como argumento.

Instance method objects have attributes, too: `m.__self__` is the instance object with the method `m()`, and `m.__func__` is the function object corresponding to the method.

9.8 Iteradores

Es probable que hayas notado que la mayoría de los objetos contenedores pueden ser recorridos usando una sentencia `for`:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

Este estilo de acceso es limpio, conciso y conveniente. El uso de iteradores está impregnado y unifica a Python. En bambalinas, la sentencia `for` llama a `iter()` en el objeto contenedor. La función retorna un objeto iterador que define el método `__next__()` que accede elementos en el contenedor de a uno por vez. Cuando no hay más elementos, `__next__()` lanza una excepción `StopIteration` que le avisa al bucle del `for` que hay que terminar. Podés llamar al método `__next__()` usando la función integrada `next()`; este ejemplo muestra como funciona todo esto:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Habiendo visto la mecánica del protocolo de iteración, es fácil agregar comportamiento de iterador a tus clases. Definí un método `__iter__()` que retorne un objeto con un método `__next__()`. Si la clase define `__next__()`, entonces alcanza con que `__iter__()` retorne `self`:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.9 Generadores

Generators son una herramienta simple y poderosa para crear iteradores. Están escritas como funciones regulares pero usan la palabra clave `yield` siempre que quieran retornar datos. Cada vez que se llama a `next()`, el generador se reanuda donde lo dejó (recuerda todos los valores de datos y qué instrucción se ejecutó por última vez). Un ejemplo muestra que los generadores pueden ser trivialmente fáciles de crear:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Todo lo que puede ser hecho con generadores también puede ser hecho con iteradores basados en clases, como se describe en la sección anterior. Lo que hace que los generadores sean tan compactos es que los métodos `__iter__()` y `__next__()` son creados automáticamente.

Otra característica clave es que las variables locales y el estado de la ejecución son guardados automáticamente entre llamadas. Esto hace que la función sea más fácil de escribir y quede mucho más claro que hacerlo usando variables de instancia tales como `self.indice` y `self.datos`.

Además de la creación automática de métodos y el guardar el estado del programa, cuando los generadores terminan automáticamente lanzan `StopIteration`. Combinadas, estas características facilitan la creación de iteradores, y hacen que no sea más esfuerzo que escribir una función regular.

9.10 Expresiones generadoras

Algunos generadores simples pueden ser escritos de manera concisa como expresiones usando una sintaxis similar a las comprensiones de listas pero con paréntesis en lugar de corchetes. Estas expresiones están hechas para situaciones donde el generador es utilizado de inmediato por la función que lo encierra. Las expresiones generadoras son más compactas pero menos versátiles que las definiciones completas de generadores y tienden a ser más amigables con la memoria que sus comprensiones de listas equivalentes.

Ejemplos:

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> unique_words = set(word for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

Notas al pie

CAPÍTULO 10

Breve recorrido por la Biblioteca Estándar

10.1 Interfaz del sistema operativo

El módulo `os` proporciona docenas de funciones para interactuar con el sistema operativo:

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python314'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

Asegúrate de utilizar el estilo `import os` en lugar de `from os import *`. Esto evitará que `os.open()` oculte la función integrada `open()`, que funciona de manera muy diferente.

Las funciones integradas `dir()` y `help()` son útiles como ayudas interactivas para trabajar con módulos grandes como `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

Para las tareas diarias de administración de archivos y directorios, el módulo `shutil` proporciona una interfaz en un nivel superior que es más fácil de usar:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.2 Comodines de archivos

El módulo `glob` proporciona una función para hacer listas de archivos a partir de búsquedas con comodines en directorios:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Argumentos de Líneas de Comandos

Los programas frecuentemente necesitan procesar argumentos de linea de comandos. Estos argumentos se almacenan en el atributo *argv* del módulo `sys` como una lista. Por ejemplo, consideremos el siguiente archivo :file: “demo.py”:

```
# File demo.py
import sys
print(sys.argv)
```

Este es el resultado de ejecutar `python demo.py one two three` en la línea de comandos:

```
['demo.py', 'one', 'two', 'three']
```

El modulo `argparse` provee un mecanismo más sofisticado para procesar argumentos recibidos vía línea de comandos. El siguiente *script* extrae uno o más nombres de archivos y un número opcional de líneas para mostrar:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

Cuando se ejecuta en la línea de comandos con `python top.py --lines=5 alpha.txt beta.txt`, el *script* establece `args.lines` a `5` y `args.filenames` a `['alpha.txt', 'beta.txt']`.

10.4 Redirigir la salida de error y finalización del programa

El módulo `sys` también tiene sus atributos para *stdin*, *stdout*, y *stderr*. Este último es útil para emitir mensajes de alerta y error para que se vean incluso cuando se haya redirigido *stdout*:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

La forma más directa de terminar un programa es usar `sys.exit()`.

10.5 Coincidencia de patrones de cadena

El módulo `re` provee herramientas de expresiones regulares para un procesamiento avanzado de cadenas. Para manipulación y coincidencias complejas, las expresiones regulares ofrecen soluciones concisas y optimizadas:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Cuando sólo se necesitan funciones sencillas, se prefieren los métodos de cadenas porque son más fáciles de leer y depurar:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Matemáticas

The `math` module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

El módulo `random` provee herramientas para realizar selecciones aleatorias:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

El módulo `statistics` calcula propiedades de estadística básica (la media, mediana, varianza, etc) de datos numéricos:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

El proyecto SciPy <https://scipy.org> tiene muchos otros módulos para cálculos numéricos.

10.7 Acceso a Internet

Hay varios módulos para acceder a Internet y procesar sus protocolos. Dos de los más simples son `urllib.request` para traer data de URLs y `smtplib` para enviar correos:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()             # Convert bytes to a str
...         if line.startswith('datetime'):
```

(continúe en la próxima página)

(proviene de la página anterior)

```
...             print(line.rstrip())          # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Nota que el segundo ejemplo necesita un servidor de correo corriendo en la máquina local)

10.8 Fechas y tiempos

El módulo `datetime` ofrece clases para gestionar fechas y tiempos tanto de manera simple como compleja. Aunque soporta aritmética sobre fechas y tiempos, la aplicación se centra en la extracción eficiente de elementos para el formateo y la manipulación de los datos de salida. El módulo también admite objetos que tienen en cuenta la zona horaria.

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Compresión de datos

Los módulos admiten directamente los formatos más comunes de archivo y compresión de datos, entre ellos : `zlib`, `gzip`, `bz2`, `lzma`, `zipfile` y `tarfile`.

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10 Medición de rendimiento

Algunos usuarios de Python desarrollan un profundo interés en conocer el rendimiento relativo de las diferentes soluciones al mismo problema. Python provee una herramienta de medición que responde inmediatamente a esas preguntas.

Por ejemplo, puede ser tentador usar la característica de empaquetado y desempaquetado de las tuplas en lugar de la solución tradicional para intercambiar argumentos. El módulo `timeit` muestra rápidamente una ligera ventaja de rendimiento:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

En contraste con el fino nivel de medición del módulo `timeit`, los módulos `profile` y `pstats` proveen herramientas para identificar secciones críticas de tiempo en bloques de código más grandes.

10.11 Control de calidad

Una forma para desarrollar software de alta calidad consiste en escribir pruebas para cada función mientras se desarrolla y correr esas pruebas frecuentemente durante el proceso de desarrollo.

El módulo `doctest` provee una herramienta para revisar un módulo y validar las pruebas integradas en las cadenas de documentación (o *docstring*) del programa. La construcción de las pruebas es tan sencillo como cortar y pegar una ejecución típica junto con sus resultados en los docstrings. Esto mejora la documentación al proveer al usuario un ejemplo y permite que el módulo *doctest* se asegure que el código permanece fiel a la documentación:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()   # automatically validate the embedded tests
```

El módulo `unittest` no es tan sencillo como el módulo `doctest`, pero permite mantener un conjunto más completo de pruebas en un archivo independiente:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main()  # Calling from the command line invokes all tests
```

10.12 Pilas incluidas

Python tiene una filosofía de «pilas incluidas». Esto se ve mejor en las capacidades robustas y sofisticadas de sus paquetes más grandes. Por ejemplo:

- Los módulos `xmlrpc.client` y `xmlrpc.server` convierten la implementación de llamadas a procedimientos remotos en una tarea casi trivial. A pesar de los nombres de los módulos, no se necesita ningún conocimiento o manejo directo de XML.
- El paquete `email` es una biblioteca para administrar mensajes de correo electrónico, incluidos MIME y otros documentos de mensajes basados en **RFC 2822**. A diferencia de `smtplib` y `poplib` que realmente envían y reciben mensajes, el paquete de correo electrónico tiene un conjunto de herramientas completo para crear o decodificar estructuras de mensajes complejas (incluidos los archivos adjuntos) y para implementar protocolos de codificación y encabezado de Internet.
- El paquete `json` proporciona un sólido soporte para analizar este popular formato de intercambio de datos. El módulo `csv` permite la lectura y escritura directa de archivos en formato de valor separado por comas, comúnmente compatible con bases de datos y hojas de cálculo. El procesamiento XML es compatible con los paquetes `xml.etree.ElementTree`, `xml.dom` y `xml.sax`. Juntos, estos módulos y paquetes simplifican en gran medida el intercambio de datos entre aplicaciones de Python y otras herramientas.
- El módulo `sqlite3` es un wrapper de la biblioteca de bases de datos SQLite, proporciona una base de datos constante que se puede actualizar y a la que se puede acceder utilizando una sintaxis SQL ligeramente no estándar.
- La internacionalización es compatible con una serie de módulos, incluyendo `gettext`, `locale`, y el paquete `codecs`.

CAPÍTULO 11

Pequeño paseo por la Biblioteca Estándar— Parte II

Este segundo paseo cubre módulos más avanzados que facilitan necesidades de programación complejas. Estos módulos raramente se usan en scripts cortos.

11.1 Formato de salida

El módulo `reprlib` provee una versión de `repr()` ajustada para mostrar contenedores grandes o profundamente anidados, en forma abreviada:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

El módulo `pprint` ofrece un control más sofisticado de la forma en que se imprimen tanto los objetos predefinidos como los objetos definidos por el usuario, de manera que sean legibles por el intérprete. Cuando el resultado ocupa más de una línea, el generador de «impresiones lindas» agrega saltos de línea y sangrías para mostrar la estructura de los datos más claramente:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

El módulo `textwrap` formatea párrafos de texto para que quepan dentro de cierto ancho de pantalla:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

El módulo `locale` accede a una base de datos de formatos específicos a una cultura. El atributo *grouping* de la función *format* permite una forma directa de formatear números con separadores de grupo:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 Plantillas

El módulo `string` incluye una clase versátil `Template` (plantilla) con una sintaxis simplificada apta para ser editada por usuarios finales. Esto permite que los usuarios personalicen sus aplicaciones sin necesidad de modificar la aplicación en sí.

El formato usa marcadores cuyos nombres se forman con `$` seguido de identificadores Python válidos (caracteres alfanuméricos y guión de subrayado). Si se los encierra entre llaves, pueden seguir más caracteres alfanuméricos sin necesidad de dejar espacios en blanco. `$$` genera un `$`:

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

El método `substitute()` lanza `KeyError` cuando no se suministra ningún valor para un marcador mediante un diccionario o argumento por nombre. Para algunas aplicaciones los datos suministrados por el usuario puede ser incompletos, y el método `safe_substitute()` puede ser más apropiado: deja los marcadores inalterados cuando falten datos:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Las subclases de *Template* pueden especificar un delimitador propio. Por ejemplo, una utilidad de renombrado por lotes para una galería de fotos puede escoger usar signos de porcentaje para los marcadores tales como la fecha actual, el número de secuencia de la imagen, o el formato de archivo:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
```

(continúe en la próxima página)

(proviene de la página anterior)

```
...     delimiter = '%'
...
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Las plantillas también pueden ser usadas para separar la lógica del programa de los detalles de múltiples formatos de salida. Esto permite sustituir plantillas específicas para archivos XML, reportes en texto plano, y reportes web en HTML.

11.3 Trabajar con registros estructurados conteniendo datos binarios

El módulo `struct` provee las funciones `pack()` y `unpack()` para trabajar con formatos de registros binarios de longitud variable. El siguiente ejemplo muestra cómo recorrer la información de encabezado en un archivo ZIP sin usar el módulo `zipfile`. Los códigos `"H"` e `"I"` representan números sin signo de dos y cuatro bytes respectivamente. El `"<"` indica que son de tamaño estándar y los bytes tienen ordenamiento *little-endian*:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):                      # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size     # skip to the next header
```

11.4 Multi-hilos

La técnica de multi-hilos (o *multi-threading*) permite desacoplar tareas que no tienen dependencia secuencial. Los hilos se pueden usar para mejorar el grado de reacción de las aplicaciones que aceptan entradas del usuario mientras otras tareas se ejecutan en segundo plano. Un caso de uso relacionado es ejecutar E/S en paralelo con cálculos en otro hilo.

El código siguiente muestra cómo el módulo de alto nivel `threading` puede ejecutar tareas en segundo plano mien-

tras el programa principal continúa su ejecución:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')
```

El desafío principal de las aplicaciones multi-hilo es la coordinación entre los hilos que comparten datos u otros recursos. A ese fin, el módulo *threading* provee una serie de primitivas de sincronización que incluyen bloqueos, eventos, variables de condición, y semáforos.

Aún cuando esas herramientas son poderosas, pequeños errores de diseño pueden resultar en problemas difíciles de reproducir. La forma preferida de coordinar tareas es concentrar todos los accesos a un recurso en un único hilo y después usar el módulo `queue` para alimentar dicho hilo con pedidos desde otros hilos. Las aplicaciones que usan objetos `Queue` para comunicación y coordinación entre hilos son más fáciles de diseñar, más legibles, y más confiables.

11.5 Registrando

El módulo `logging` ofrece un sistema de registros (*logs*) completo y flexible. En su forma más simple, los mensajes de registro se envían a un archivo o a `sys.stderr`:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

Ésta es la salida obtenida:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

De forma predeterminada, los mensajes de depuración e informativos se suprimen, y la salida se envía al error estándar. Otras opciones de salida incluyen mensajes de enrutamiento a través de correo electrónico, datagramas, sockets, o un servidor HTTP. Nuevos filtros pueden seleccionar diferentes rutas basadas en la prioridad del mensaje: `DEBUG`, `INFO`, `WARNING`, `ERROR`, y `CRITICAL` (Depuración, Informativo, Atención, Error y Crítico respectivamente)

El sistema de registro puede configurarse directamente desde Python o puede cargarse la configuración desde un archivo modificable por el usuario para personalizar el registro sin alterar la aplicación.

11.6 Referencias débiles

Python realiza administración de memoria automática (cuenta de referencias para la mayoría de los objetos, y *garbage collection* para eliminar ciclos). La memoria se libera poco después de que la última referencia a la misma haya sido eliminada.

Este enfoque funciona bien para la mayoría de las aplicaciones pero de vez en cuando existe la necesidad de controlar objetos sólo mientras estén siendo utilizados por otra cosa. Desafortunadamente, el sólo hecho de controlarlos crea una referencia que los convierte en permanentes. El módulo `weakref` provee herramientas para controlar objetos sin crear una referencia. Cuando el objeto no se necesita mas, es removido automáticamente de una tabla de referencias débiles y se dispara una *callback* para objetos *weakref*. Comúnmente las aplicaciones incluyen cacheo de objetos que son costosos de crear:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right away
0
>>> d['primary']                # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # entry was automatically removed
  File "C:/python314/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Herramientas para trabajar con listas

Muchas necesidades de estructuras de datos pueden ser satisfechas con el tipo integrado lista. Sin embargo, a veces se hacen necesarias implementaciones alternativas con rendimientos distintos.

The `array` module provides an `array` object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode `"H"`) rather than the usual 16 bytes per entry for regular lists of Python int objects:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

The `collections` module provides a `deque` object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

Además de las implementaciones alternativas de listas, la biblioteca ofrece otras herramientas como el módulo `bisect` con funciones para manipular listas ordenadas:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

El módulo `heapq` provee funciones para implementar pilas (*heaps*) basados en listas comunes. El menor valor ingresado se mantiene en la posición cero. Esto es útil para aplicaciones que acceden a menudo al elemento más chico pero no quieren hacer un orden completo de la lista:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rearrange the list into heap order
>>> heappush(data, -5)                 # add a new entry
>>> [heappop(data) for i in range(3)]  # fetch the three smallest entries
[-5, 0, 1]
```

11.8 Decimal Floating-Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating-point arithmetic. Compared to the built-in `float` implementation of binary floating point, the class is especially helpful for

- aplicaciones financieras y otros usos que requieren representación decimal exacta,
- control sobre la precisión,
- control sobre el redondeo para cumplir requisitos legales,
- seguimiento de dígitos decimales significativos, o
- aplicaciones donde el usuario espera que los resultados coincidan con cálculos hecho a mano.

Por ejemplo, calcular un impuesto del 5% de una tarifa telefónica de 70 centavos da resultados distintos con punto flotante decimal y punto flotante binario. La diferencia se vuelve significativa si los resultados se redondean al centavo más próximo:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

El resultado con `Decimal` conserva un cero al final, calculando automáticamente cuatro cifras significativas a partir de los multiplicandos con dos cifras significativas. Decimal reproduce la matemática como se la hace a mano, y

evita problemas que pueden surgir cuando el punto flotante binario no puede representar exactamente cantidades decimales.

La representación exacta permite a la clase `Decimal` hacer cálculos de modulo y pruebas de igualdad que son inadecuadas para punto flotante binario:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
```

El módulo `decimal` provee aritmética con tanta precisión como haga falta:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

CAPÍTULO 12

Entornos virtuales y paquetes

12.1 Introducción

Las aplicaciones en Python usualmente hacen uso de paquetes y módulos que no forman parte de la librería estándar. Las aplicaciones a veces necesitan una versión específica de una librería, debido a que dicha aplicación requiere que un bug particular haya sido solucionado o bien la aplicación ha sido escrita usando una versión obsoleta de la interfaz de la librería.

Esto significa que tal vez no sea posible para una instalación de Python cumplir los requerimientos de todas las aplicaciones. Si la aplicación A necesita la versión 1.0 de un módulo particular y la aplicación B necesita la versión 2.0, entonces los requerimientos entran en conflicto e instalar la versión 1.0 o 2.0 dejará una de las aplicaciones sin funcionar.

La solución a este problema es crear un *entorno virtual*, un directorio que contiene una instalación de Python de una versión en particular, además de unos cuantos paquetes adicionales.

Diferentes aplicaciones pueden entonces usar entornos virtuales diferentes. Para resolver el ejemplo de requerimientos en conflicto citado anteriormente, la aplicación A puede tener su propio entorno virtual con la versión 1.0 instalada mientras que la aplicación B tiene otro entorno virtual con la versión 2.0. Si la aplicación B requiere que actualizar la librería a la versión 3.0, ésto no afectará el entorno virtual de la aplicación A.

12.2 Creando entornos virtuales

The module used to create and manage virtual environments is called `venv`. `venv` will install the Python version from which the command was run (as reported by the `--version` option). For instance, executing the command with `python3.12` will install version 3.12.

Para crear un entorno virtual, decide en que carpeta quieres crearlo y ejecuta el módulo `venv` como script con la ruta a la carpeta:

```
python -m venv tutorial-env
```

Esto creará el directorio `tutorial-env` si no existe, y también creará directorios dentro de él que contienen una copia del intérprete de Python y varios archivos de soporte.

Una ruta común para el directorio de un entorno virtual es `.venv`. Ese nombre mantiene el directorio típicamente escondido en la consola y fuera de vista mientras le da un nombre que explica cuál es el motivo de su existencia. También permite que no haya conflicto con los ficheros de definición de variables de entorno `.env` que algunas herramientas soportan.

Una vez creado el entorno virtual, podrás activarlo.

En Windows, ejecuta:

```
tutorial-env\Scripts\activate
```

En Unix o MacOS, ejecuta:

```
source tutorial-env/bin/activate
```

(Este script está escrito para la consola bash. Si usas las consolas **csh** or **fish**, hay scripts alternativos `activate.csh` y `activate.fish` que deberá usar en su lugar.)

Activar el entorno virtual cambiará el prompt de tu consola para mostrar que entorno virtual está usando, y modificará el entorno para que al ejecutar `python` sea con esa versión e instalación en particular. Por ejemplo:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May  6 2016, 10:59:36)
  ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
'~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

Para desactivar el entorno virtual, digita:

```
deactivate
```

en el terminal.

12.3 Manejando paquetes con pip

Puede instalar, actualizar y eliminar paquetes usando un programa llamado **pip**. De forma predeterminada, `pip` instalará paquetes desde el Indice de Paquetes de Python. Puede navegar por el índice de paquetes de Python yendo a él en su navegador web.

`pip` tiene varios subcomandos: «install», «uninstall», «freeze», etc. (Consulte la guía installing-index para obtener la documentación completa de `pip`).

Se puede instalar la última versión de un paquete especificando el nombre del paquete:

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

También se puede instalar una versión específica de un paquete ingresando el nombre del paquete seguido de == y el número de versión:

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

Si se ejecuta de nuevo el comando, `pip` detectará que la versión ya está instalada y no hará nada. Se puede ingresar un número de versión diferente para instalarlo, o se puede ejecutar `pip install --upgrade` para actualizar el paquete a la última versión:

```
(tutorial-env) $ python -m pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

`pip -m pip uninstall` seguido de uno o varios nombres de paquetes eliminará los paquetes del entorno virtual.

`python -m pip show` mostrará información de un paquete en particular:

```
(tutorial-env) $ python -m pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

`python -m pip list` mostrará todos los paquetes instalados en el entorno virtual:

```
(tutorial-env) $ python -m pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`python -m pip freeze` retorna una lista de paquetes instalados, pero el formato de salida es el requerido por `python -m pip install`. Una convención común es poner esta lista en un archivo `requirements.txt`:

```
(tutorial-env) $ python -m pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

El archivo `requirements.txt` puede ser agregado al controlador de versiones y distribuido como parte de la aplicación. Los usuarios pueden entonces instalar todos los paquetes necesarios con `install -r`:

```
(tutorial-env) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
  ...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
  ...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
  ...
Installing collected packages: novas, numpy, requests
```

(continúe en la próxima página)

(proviene de la página anterior)

```
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` tiene muchas más opciones. Consulte la guía installing-index para obtener documentación completa de `pip`. Cuando haya escrito un paquete y desee que esté disponible en el índice de paquetes de Python, consulte la Guía de usuario de empaquetado de Python.

CAPÍTULO 13

¿Y ahora qué?

Leer este tutorial probablemente reforzó tu interés por usar Python, deberías estar ansioso por aplicar Python a la resolución de tus problemas reales. ¿A dónde deberías ir para aprender más?

Este tutorial forma parte del conjunto de documentación de Python. Algunos otros documentos que encontrarás en este conjunto son:

- library-index:

  Deberías navegar a través de este manual, que da una completa (aunque breve) referencia sobre los tipos, funciones y módulos en la librería estándar. La distribución estándar de Python incluye *mucho* más código adicional. Hay módulos para leer buzones Unix, recuperar documentos vía HTTP, generar números aleatorios, analizar opciones de línea de comandos, escribir programas CGI, comprimir datos y muchas más tareas. Echar una ojeada a la Librería de Referencia te dará una idea de lo que está disponible.

- installing-index explica como instalar módulos adicionales escritos por otros usuarios de Python.

- reference-index: Una explicación detallada de la sintaxis y la semántica de Python. Es una lectura pesada, pero es muy útil como guía complete del lenguaje.

Más recursos sobre Python:

- https://www.python.org: El mayor sitio web de Python. Contiene código, documentación, y enlaces a páginas web relacionadas con Python.

- https://docs.python.org: Acceso rápido a la documentación de Python.

- https://pypi.org: El Python Package Index, apodado previamente la Tienda de Queso[1], es un índice de módulos de Python creados por usuarios que están disponibles para su descarga. Cuando empiezas a distribuir código, lo puedes registrar allí para que otros lo encuentren.

- https://code.activestate.com/recipes/langs/python/: El Python Cookbook es una gran colección de ejemplos de código, módulos grandes y scripts útiles. Las contribuciones más notables también están recogidas en un libro titulado Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.)

- http://www.pyvideo.org recoge enlaces a vídeos relacionados con Python provenientes de conferencias y reuniones de grupos de usuarios.

- https://scipy.org: El proyecto de Python científico incluye módulos para el cálculo rápido de operaciones y manipulaciones sobre arrays además de muchos paquetes para cosas como Álgebra Lineal, Transformadas de

[1] La tienda de queso, *Cheese Shop*, es un chiste de *Monty Python*: un cliente entra a una tienda de queso pero para cualquier queso que pide, el vendedor le dice que no lo tienen.

Fourier, solucionadores de sistemas no-lineales, distribuciones de números aleatorios, análisis estadísticos y otras herramientas.

Para preguntas relacionadas con Python y reportes de problemas puedes escribir al grupo de noticias *comp.lang.python*, o enviarlas a la lista de correo que hay en python-list@python.org. El grupo de noticias y la lista de correo están conectados, por lo que los mensajes enviados a uno serán retransmitidos al otro. Hay alrededor de cientos de mensajes diarios (con picos de hasta varios cientos), haciendo (y respondiendo) preguntas, sugiriendo nuevas características, y anunciando nuevos módulos. Los archivos de la lista de correos están disponibles en https://mail.python.org/pipermail/.

Antes de escribir, asegúrate de haber revisado la lista de Frequently Asked Questions (también llamado el *FAQ*). Muchas veces responde las preguntas que se hacen una y otra vez, y quizás contenga la solución a tu problema.

Notas al pie

CAPÍTULO 14

Edición de entrada interactiva y sustitución de historial

Algunas versiones del intérprete de Python permiten editar la línea de entrada actual, y sustituir en base al historial, de forma similar a las capacidades del intérprete de comandos *Korn* y el GNU bash. Esto se implementa con la biblioteca GNU Readline, que soporta varios estilos de edición. Esta biblioteca tiene su propia documentación la cuál no vamos a duplicar aquí.

14.1 Autocompletado con tab e historial de edición

El autocompletado de variables y nombres de módulos es automatically enabled al iniciar el intérprete, por lo tanto la tecla `Tab` invoca la función de autocompletado; ésta mira en los nombres de sentencia, las variables locales y los nombres de módulos disponibles. Para expresiones con puntos como `string.a`, va a evaluar la expresión hasta el `'.'` final y entonces sugerir autocompletado para los atributos del objeto resultante. Nota que esto quizás ejecute código de aplicaciones definidas si un objeto con un método `__getattr__()` es parte de la expresión. La configuración por omisión también guarda tu historial en un archivo llamado `.python_history` en tu directorio de usuario. El historial estará disponible durante la próxima sesión interactiva del intérprete.

14.2 Alternativas al intérprete interactivo

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an `INDENT` token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

Un intérprete interactivo mejorado alternativo que está dando vueltas desde hace rato es IPython, que ofrece completado por tab, exploración de objetos, y administración avanzada del historial. También puede ser configurado en profundidad, e integrarse en otras aplicaciones. Otro entorno interactivo mejorado similar es bpython.

CAPÍTULO 15

Floating-Point Arithmetic: Issues and Limitations

Los números de punto flotante se representan en el hardware de computadoras como fracciones en base 2 (binarias). Por ejemplo, la fracción **decimal** `0.625` tiene un valor de 6/10 + 2/100 + 5/1000, y de la misma manera, la fracción binaria `0.101` tiene un valor de 1/2 + 0/4 + 1/8. Estas dos fracciones tienen valores idénticos; la única diferencia real radica en que la primera se escribe en notación fraccional en base 10, y la segunda en base 2.

Desafortunadamente, la mayoría de las fracciones decimales no pueden representarse exactamente como fracciones binarias. Como consecuencia, en general los números de punto flotante decimal que ingresás en la computadora son sólo aproximados por los números de punto flotante binario que realmente se guardan en la máquina.

El problema es más fácil de entender primero en base 10. Considerá la fracción 1/3. Podés aproximarla como una fracción de base 10

```
0.3
```

…o, mejor,

```
0.33
```

…o, mejor,

```
0.333
```

…y así. No importa cuantos dígitos desees escribir, el resultado nunca será exactamente 1/3, pero será una aproximación cada vez mejor de 1/3.

De la misma manera, no importa cuantos dígitos en base 2 quieras usar, el valor decimal 0.1 no puede representarse exactamente como una fracción en base 2. En base 2, 1/10 es la siguiente fracción que se repite infinitamente:

```
0.0001100110011001100110011001100110011001100110011...
```

Frená en cualquier número finito de bits, y tendrás una aproximación. En la mayoría de las máquinas hoy en día, los float se aproximan usando una fracción binaria con el numerador usando los primeros 53 bits con el bit más significativos y el denominador como una potencia de dos. En el caso de 1/10, la fracción binaria es `3602879701896397 / 2 ** 55` que está cerca pero no es exactamente el valor verdadero de 1/10.

La mayoría de los usuarios no son conscientes de esta aproximación por la forma en que se muestran los valores. Python solamente muestra una aproximación decimal al valor verdadero decimal de la aproximación binaria almacenada por la máquina. En la mayoría de las máquinas, si Python fuera a imprimir el verdadero valor decimal de la aproximación binaria almacenada para 0.1, debería mostrar:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Esos son más dígitos que lo que la mayoría de la gente encuentra útil, por lo que Python mantiene manejable la cantidad de dígitos al mostrar un valor redondeado en su lugar:

```
>>> 1 / 10
0.1
```

Sólo recordá que, a pesar de que el valor mostrado resulta ser exactamente 1/10, el valor almacenado realmente es la fracción binaria más cercana posible.

Interesantemente, hay varios números decimales que comparten la misma fracción binaria más aproximada. Por ejemplo, los números `0.1`, `0.10000000000000001` y `0.1000000000000000055511151231257827021181583404541015625` son todos aproximados por `3602879701896397 / 2 ** 55`. Ya que todos estos valores decimales comparten la misma aproximación, se podría mostrar cualquiera de ellos para preservar el invariante `eval(repr(x)) == x`.

Históricamente, el prompt de Python y la función integrada `repr()` eligieron el valor con los 17 dígitos, `0.10000000000000001`. Desde Python 3.1, en la mayoría de los sistemas Python ahora es capaz de elegir la forma más corta de ellos y mostrar `0.1`.

Note that this is in the very nature of binary floating point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

Para una salida más elegante, quizás quieras usar el formateo de cadenas de texto para generar un número limitado de dígitos significativos:

```
>>> format(math.pi, '.12g')  # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f')   # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

Es importante darse cuenta que esto es, realmente, una ilusión: estás simplemente redondeando al *mostrar* el valor verdadero de la máquina.

Una ilusión puede generar otra. Por ejemplo, ya que 0.1 no es exactamente 1/10, sumar tres veces 0.1 podría también no generar exactamente 0.3:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

También, ya que 0.1 no puede acercarse más al valor exacto de 1/10 y 0.3 no puede acercarse más al valor exacto de 3/10, redondear primero con la función `round()` no puede ayudar:

```
>>> round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1)
False
```

Aunque los números no pueden acercarse más a sus valores exactos previstos, la función `math.isclose()` puede ser útil para comparar valores inexactos:

```
>>> math.isclose(0.1 + 0.1 + 0.1, 0.3)
True
```

Alternatively, the `round()` function can be used to compare rough approximations:

```
>>> round(math.pi, ndigits=2) == round(22 / 7, ndigits=2)
True
```

Binary floating-point arithmetic holds many surprises like this. The problem with «0.1» is explained in precise detail below, in the «Representation Error» section. See Examples of Floating Point Problems for a pleasant summary of how binary floating point works and the kinds of problems commonly encountered in practice. Also see The Perils of Floating Point for a more complete account of other common surprises.

As that says near the end, «there are no easy answers.» Still, don't be unduly wary of floating point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2**53 per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic and that every float operation can suffer a new rounding error.

A pesar de que existen casos patológicos, para la mayoría de usos casuales de la aritmética de punto flotante al final verás el resultado que esperás si simplemente redondeás lo que mostrás de tus resultados finales al número de dígitos decimales que esperás. `str()` es normalmente suficiente, y para un control más fino mirá los parámetros del método de formateo `str.format()` en string-formatting.

Para los casos de uso que necesitan una representación decimal exacta, probá el módulo `decimal`, que implementa aritmética decimal útil para aplicaciones de contabilidad y de alta precisión.

El módulo `fractions` soporta otra forma de aritmética exacta, ya que implementa aritmética basada en números racionales (por lo que números como 1/3 pueden ser representados exactamente).

Si eres un usuario intensivo de operaciones de punto flotante, deberías echar un vistazo al paquete NumPy y a muchos otros paquetes para operaciones matemáticas y estadísticas proporcionados por el proyecto SciPy. Ver <https://scipy.org>.

Python provee herramientas que pueden ayudar en esas raras ocasiones cuando realmente *querés* saber el valor exacto de un punto flotante. El método `float.as_integer_ratio()` expresa el valor del punto flotante como una fracción:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Ya que la fracción es exacta, se puede usar para recrear sin pérdidas el valor original:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

El método `float.hex()` expresa un punto flotante en hexadecimal (base 16), nuevamente retornando el valor exacto almacenado por tu computadora:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

Esta representación hexadecimal precisa se puede usar para reconstruir el valor exacto del punto flotante:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Ya que la representación es exacta, es útil para portar valores a través de diferentes versiones de Python de manera confiable (independencia de plataformas) e intercambiar datos con otros lenguajes que soportan el mismo formato (como Java y C99).

Otra herramienta útil es la función `sum()` que ayuda a mitigar la pérdida de precisión durante la suma. Utiliza precisión extendida para pasos de redondeo intermedios a medida que se agregan valores a un total en ejecución. Esto puede marcar la diferencia en la precisión general para que los errores no se acumulen hasta el punto en que afecten el total final:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
>>> sum([0.1] * 10) == 1.0
True
```

The `math.fsum()` goes further and tracks all of the «lost digits» as values are added onto a running total so that the result has only a single rounding. This is slower than `sum()` but will be more accurate in uncommon cases where large magnitude inputs mostly cancel each other out leaving a final sum near zero:

```
>>> arr = [-0.10430216751806065, -266310978.67179024, 143401161448607.16,
...         -143401161400469.7, 266262841.31058735, -0.003244936839808227]
>>> float(sum(map(Fraction, arr)))   # Exact summation with single rounding
8.042173697819788e-13
>>> math.fsum(arr)                   # Single rounding
8.042173697819788e-13
>>> sum(arr)                         # Multiple roundings in extended precision
8.042178034628478e-13
>>> total = 0.0
>>> for x in arr:
...     total += x                   # Multiple roundings in standard precision
...
>>> total                            # Straight addition has no correct digits!
-0.0051575902860057365
```

15.1 Error de Representación

Esta sección explica el ejemplo «0.1» en detalle, y muestra como en la mayoría de los casos vos mismo podés realizar un análisis exacto como este. Se asume un conocimiento básico de la representación de punto flotante binario.

Error de representación se refiere al hecho de que algunas (la mayoría) de las fracciones decimales no pueden representarse exactamente como fracciones binarias (en base 2). Esta es la razón principal de por qué Python (o Perl, C, C++, Java, Fortran, y tantos otros) frecuentemente no mostrarán el número decimal exacto que esperás.

¿Por qué sucede esto? 1/10 no es exactamente representable como una fracción binaria. Desde al menos el año 2000, casi todas las máquinas utilizan la aritmética de punto flotante binaria IEEE 754, y casi todas las plataformas asignan los números de punto flotante de Python a valores binarios de 64 bits de precisión «doble» de IEEE 754. Los valores binarios de IEEE 754 de 64 bits contienen 53 bits de precisión, por lo que en la entrada, la computadora se esfuerza por convertir 0.1 en la fracción más cercana de la forma *J*/2***N* donde *J* es un número entero que contiene exactamente 53 bits. Reescribiendo

```
1 / 10 ~= J / (2**N)
```

…como

```
J ~= 2**N / 10
```

…y recordando que *J* tiene exactamente 53 bits (es `>= 2**52` pero `< 2**53`), el mejor valor para *N* es 56:

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

O sea, 56 es el único valor para *N* que deja *J* con exactamente 53 bits. El mejor valor posible para *J* es entonces el cociente redondeado:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Ya que el resto es más que la mitad de 10, la mejor aproximación se obtiene redondeándolo:

```
>>> q+1
7205759403792794
```

Por lo tanto la mejor aproximación a 1/10 en doble precisión IEEE 754 es:

```
7205759403792794 / 2 ** 56
```

El dividir tanto el numerador como el denominador reduce la fracción a:

```
3602879701896397 / 2 ** 55
```

Notá que como lo redondeamos, esto es un poquito más grande que 1/10; si no lo hubiéramos redondeado, el cociente hubiese sido un poquito menor que 1/10. ¡Pero no hay caso en que sea *exactamente* 1/10!

Entonces la computadora nunca «ve» 1/10: lo que ve es la fracción exacta de arriba, la mejor aproximación al flotante doble IEEE 754 que puede obtener:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

Si multiplicamos esa fracción por 10**55, podemos ver el valor hasta los 55 dígitos decimales:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

lo que significa que el valor exacto almacenado en la computadora es igual al valor decimal 0.1000000000000000055511151231257827021181583404541015625. En lugar de mostrar el valor decimal completo, muchos lenguajes (incluyendo versiones anteriores de Python), redondean el resultado a 17 dígitos significativos:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

Los módulos `fractions` y `decimal` hacen fácil estos cálculos:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```

CAPÍTULO 16

Apéndice

16.1 Modo interactivo

There are two variants of the interactive *REPL*. The classic basic interpreter is supported on all platforms with minimal line control capabilities.

On Windows, or Unix-like systems with `curses` support, a new interactive shell is used by default since Python 3.13. This one supports color, multiline editing, history browsing, and paste mode. To disable color, see using-on-controlling-color for details. Function keys provide some additional functionality. `F1` enters the interactive help browser `pydoc`. `F2` allows for browsing command-line history with neither output nor the `>>>` and `...` prompts. `F3` enters «paste mode», which makes pasting larger blocks of code easier. Press `F3` to return to the regular prompt.

When using the new interactive shell, exit the shell by typing `exit` or `quit`. Adding call parentheses after those commands is not required.

If the new interactive shell is not desired, it can be disabled via the `PYTHON_BASIC_REPL` environment variable.

16.1.1 Manejo de errores

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an `except` clause in a `try` statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit status; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from executed commands is written to standard output.

Al ingresar el carácter de interrupción (por lo general `Control-C` o `Supr`) en el prompt primario o secundario, se cancela la entrada y retorna al prompt primario.[1] Tipear una interrupción mientras un comando se están ejecutando lanza la excepción `KeyboardInterrupt`, que puede ser manejada con una sentencia `try`.

16.1.2 Programas ejecutables de Python

En los sistemas Unix y tipo BSD, los programas Python pueden convertirse directamente en ejecutables, como programas del intérprete de comandos, poniendo la linea:

```
#!/usr/bin/env python3
```

[1] Un problema con el paquete GNU Readline puede prevenir esto.

…al principio del script y dándole al archivo permisos de ejecución (asumiendo que el intérprete están en la variable de entorno `PATH` del usuario). `#!` deben ser los primeros dos caracteres del archivo. En algunas plataformas, la primera línea debe terminar al estilo Unix (`'\n'`), no como en Windows (`'\r\n'`). Notá que el carácter numeral `'#'` se usa en Python para comenzar un comentario.

Se le puede dar permisos de ejecución al script usando el comando **chmod**.

```
$ chmod +x myscript.py
```

En sistemas Windows, no existe la noción de «modo ejecutable». El instalador de Python asocia automáticamente la extensión `.py` con `python.exe` para que al hacerle doble clic a un archivo Python se corra el script. La extensión también puede ser `.pyw`, en este caso se omite la ventana con la consola que normalmente aparece.

16.1.3 El archivo de inicio interactivo

Cuando usas Python en forma interactiva, suele ser útil que algunos comandos estándar se ejecuten cada vez que el intérprete se inicia. Puedes hacer esto configurando la variable de entorno `PYTHONSTARTUP` con el nombre de un archivo que contenga tus comandos de inicio. Esto es similar al archivo `.profile` en los intérpretes de comandos de Unix.

Este archivo es solo leído en las sesiones interactivas del intérprete, no cuando Python lee comandos de un script ni cuando `/dev/tty` se explicita como una fuente de comandos (que de otro modo se comporta como una sesión interactiva). Se ejecuta en el mismo espacio de nombres en el que los comandos interactivos se ejecutan, entonces los objetos que define o importa pueden ser usados sin cualificaciones en la sesión interactiva. En este archivo también puedes cambiar los prompts `sys.ps1` y `sys.ps2`.

Si quieres leer un archivo de inicio adicional desde el directorio actual, puedes programarlo en el archivo de inicio global usando algo como `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())`. Si quieres usar el archivo de inicio en un script, tienes que hacer lo siguiente de forma explícita en el script:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

16.1.4 Los módulos de customización

Python provee dos formas para customizarlo: sitecustomize y usercustomize. Para ver cómo funciona, necesitas primero encontrar dónde está tu directorio para tu usuario de paquetes del sistema. Inicia Python y ejecuta el siguiente código:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.x/site-packages'
```

Ahora puedes crear un archivo llamado `usercustomize.py` en ese directorio y poner lo que quieras en él. Eso afectará cada ejecución de Python, a menos que se inicie con la opción `-s` para deshabilitar esta importación automática.

sitecustomize funciona de la misma manera, pero normalmente lo crea el administrador de la computadora en el directorio global de paquetes del sistema, y se importa antes que usercustomize. Para más detalles, mira la documentación del módulo `site`.

Notas al pie

APÉNDICE A

Glosario

>>>
: The default Python prompt of the *interactive* shell. Often seen for code examples which can be executed interactively in the interpreter.

...
: Puede referirse a:

  - The default Python prompt of the *interactive* shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.
  - The three dots form of the Ellipsis object.

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`.

annotate function
: A function that can be called to retrieve the *annotations* of an object. This function is accessible as the `__annotate__` attribute of functions, classes, and modules. Annotate functions are a subset of *evaluate functions*.

anotación
: Una etiqueta asociada a una variable, atributo de clase, parámetro de función o valor de retorno, usado por convención como un *type hint*.

  Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions can be retrieved by calling `annotationlib.get_annotations()` on modules, classes, and functions, respectively.

  See *variable annotation*, *function annotation*, **PEP 484**, **PEP 526**, and **PEP 649**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

argumento
: Un valor pasado a una *function* (o *method*) cuando se llama a la función. Hay dos clases de argumentos:

- *argumento nombrado*: es un argumento precedido por un identificador (por ejemplo, `nombre=`) en una llamada a una función o pasado como valor en un diccionario precedido por `**`. Por ejemplo `3` y `5` son argumentos nombrados en las llamadas a `complex()`:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *argumento posicional* son aquellos que no son nombrados. Los argumentos posicionales deben aparecer al principio de una lista de argumentos o ser pasados como elementos de un *iterable* precedido por `*`. Por ejemplo, `3` y `5` son argumentos posicionales en las siguientes llamadas:

```
complex(3, 5)
complex(*(3, 5))
```

Los argumentos son asignados a las variables locales en el cuerpo de la función. Vea en la sección calls las reglas que rigen estas asignaciones. Sintácticamente, cualquier expresión puede ser usada para representar un argumento; el valor evaluado es asignado a la variable local.

Vea también el *parameter* en el glosario, la pregunta frecuente la diferencia entre argumentos y parámetros, y **PEP 362**.

administrador asincrónico de contexto
Un objeto que controla el entorno visible en un sentencia `async with` al definir los métodos `__aenter__()` y `__aexit__()`. Introducido por **PEP 492**.

generador asincrónico
Una función que retorna un *asynchronous generator iterator*. Es similar a una función corrutina definida con `async def` excepto que contiene expresiones `yield` para producir series de variables usadas en un ciclo `async for`.

Usualmente se refiere a una función generadora asincrónica, pero puede referirse a un *iterador generador asincrónico* en ciertos contextos. En aquellos casos en los que el significado no está claro, usar los términos completos evita la ambigüedad.

Una función generadora asincrónica puede contener expresiones `await` así como sentencias `async for`, y `async with`.

iterador generador asincrónico
An object created by an *asynchronous generator* function.

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`.

Each `yield` temporarily suspends processing, remembering the execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** and **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**.

attached thread state
A *thread state* that is active for the current OS thread.

When a *thread state* is attached, the OS thread has access to the full Python C API and can safely invoke the bytecode interpreter.

Unless a function explicitly notes otherwise, attempting to call the C API without an attached thread state will result in a fatal error or undefined behavior. A thread state can be attached and detached explicitly by the

user through the C API, or implicitly by the runtime, including during blocking C calls and by the bytecode interpreter in between calls.

On most builds of Python, having an attached thread state implies that the caller holds the *GIL* for the current interpreter, so only one OS thread can have an attached thread state at a given moment. In *free-threaded* builds of Python, threads can concurrently hold an attached thread state, allowing for true parallelism of the bytecode interpreter.

atributo
Un valor asociado a un objeto al que se suele hacer referencia por su nombre utilizando expresiones punteadas. Por ejemplo, si un objeto *o* tiene un atributo *a* se referenciaría como *o.a*.

Es posible dar a un objeto un atributo cuyo nombre no sea un identificador definido por identifiers, por ejemplo usando `setattr()`, si el objeto lo permite. Dicho atributo no será accesible utilizando una expresión con puntos, y en su lugar deberá ser recuperado con `getattr()`.

a la espera
Un objeto que puede utilizarse en una expresión `await`. Puede ser una *corutina* o un objeto con un método `__await__()`. Véase también **PEP 492**.

BDFL
Sigla de *Benevolent Dictator For Life*, benevolente dictador vitalicio, es decir Guido van Rossum, el creador de Python.

archivo binario
A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

Vea también *text file* para un objeto archivo capaz de leer y escribir objetos `str`.

referencia prestada
En la API C de Python, una referencia prestada es una referencia a un objeto, donde el código usando el objeto no posee la referencia. Se convierte en un puntero colgante si se destruye el objeto. Por ejemplo, una recolección de basura puede eliminar el último *strong reference* del objeto y así destruirlo.

Se recomienda llamar a `Py_INCREF()` en la *referencia prestada* para convertirla en una *referencia fuerte* in situ, excepto cuando el objeto no se puede destruir antes del último uso de la referencia prestada. La función `Py_NewRef()` se puede utilizar para crear una nueva *referencia fuerte*.

objetos tipo binarios
Un objeto que soporta bufferobjects y puede exportar un búfer C-*contiguous*. Esto incluye todas los objetos `bytes`, `bytearray`, y `array.array`, así como muchos objetos comunes `memoryview`. Los objetos tipo binarios pueden ser usados para varias operaciones que usan datos binarios; éstas incluyen compresión, salvar a archivos binarios, y enviarlos a través de un socket.

Algunas operaciones necesitan que los datos binarios sean mutables. La documentación frecuentemente se refiere a éstos como «objetos tipo binario de lectura y escritura». Ejemplos de objetos de búfer mutables incluyen a `bytearray` y `memoryview` de la `bytearray`. Otras operaciones que requieren datos binarios almacenados en objetos inmutables («objetos tipo binario de sólo lectura»); ejemplos de éstos incluyen `bytes` y `memoryview` del objeto `bytes`.

bytecode
El código fuente Python es compilado en *bytecode*, la representación interna de un programa python en el intérprete CPython. El *bytecode* también es guardado en caché en los archivos *.pyc* de tal forma que ejecutar el mismo archivo es más fácil la segunda vez (la recompilación desde el código fuente a *bytecode* puede ser evitada). Este «lenguaje intermedio» deberá corren en una *virtual machine* que ejecute el código de máquina correspondiente a cada *bytecode*. Note que los *bytecodes* no tienen como requisito trabajar en las diversas máquina virtuales de Python, ni de ser estable entre versiones Python.

Una lista de las instrucciones en *bytecode* está disponible en la documentación de el módulo dis.

callable
Un callable es un objeto que puede ser llamado, posiblemente con un conjunto de argumentos (véase *argument*), con la siguiente sintaxis:

```
callable(argument1, argument2, argumentN)
```

Una *function*, y por extensión un *method*, es un callable. Una instancia de una clase que implementa el método `__call__()` también es un callable.

retrollamada
Una función de subrutina que se pasa como un argumento para ejecutarse en algún momento en el futuro.

clase
Una plantilla para crear objetos definidos por el usuario. Las definiciones de clase normalmente contienen definiciones de métodos que operan una instancia de la clase.

variable de clase
Una variable definida en una clase y prevista para ser modificada sólo a nivel de clase (es decir, no en una instancia de la clase).

closure variable
A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

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.

context
This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
- A `contextvars.Context` object. Also see *current context*.

context management protocol
The `__enter__()` and `__exit__()` methods called by the `with` statement. See **PEP 343**.

administrador de contextos
An object which implements the *context management protocol* and controls the environment seen in a `with` statement. See **PEP 343**.

variable de contexto
A variable whose value depends on which context is the *current context*. Values are accessed via

`contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

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*.

current context
The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

cyclic isolate
A subgroup of one or more objects that reference each other in a reference cycle, but are not referenced by objects outside the group. The goal of the *cyclic garbage collector* is to identify these groups and break the reference cycles so that the memory can be reclaimed.

decorador
Una función que retorna otra función, usualmente aplicada como una función de transformación empleando la sintaxis `@envoltorio`. Ejemplos comunes de decoradores son `classmethod()` y `staticmethod()`.

La sintaxis del decorador es meramente azúcar sintáctico, las definiciones de las siguientes dos funciones son semánticamente equivalentes:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

El mismo concepto existe para clases, pero son menos usadas. Vea la documentación de function definitions y class definitions para mayor detalle sobre decoradores.

descriptor
Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

Para obtener más información sobre los métodos de los descriptores, consulte descriptors o Guía práctica de uso de los descriptores.

diccionario
An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

comprensión de diccionarios
Una forma compacta de procesar todos o parte de los elementos en un iterable y retornar un diccionario con los resultados. `results = {n: n ** 2 for n in range(10)}` genera un diccionario que contiene la clave `n` asignada al valor `n ** 2`. Ver comprehensions.

vista de diccionario
Los objetos retornados por los métodos `dict.keys()`, `dict.values()`, y `dict.items()` son llamados vistas de diccionarios. Proveen una vista dinámica de las entradas de un diccionario, lo que significa que cuando el diccionario cambia, la vista refleja éstos cambios. Para forzar a la vista de diccionario a convertirse en una lista completa, use `list(dictview)`. Vea dict-views.

docstring
A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

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*.

dunder
An informal short-hand for «double underscore», used when talking about a *special method*. For example, `__init__` is often pronounced «dunder init».

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.

evaluate function
A function that can be called to evaluate a lazily evaluated attribute of an object, such as the value of type aliases created with the `type` statement.

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
f-strings
String literals prefixed with `f` or `F` are commonly called «f-strings» which is short for formatted string literals. See also **PEP 498**.

objeto archivo
An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource.

Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

Existen tres categorías de objetos archivo: crudos *raw* *archivos binarios*, con búfer *archivos binarios* y *archivos de texto*. Sus interfaces son definidas en el módulo `io`. La forma canónica de crear objetos archivo es usando la función `open()`.

objetos tipo archivo
Un sinónimo de *file object*.

codificación del sistema de archivos y manejador de errores
Controlador de errores y codificación utilizado por Python para decodificar bytes del sistema operativo y codificar Unicode en el sistema operativo.

La codificación del sistema de archivos debe garantizar la decodificación exitosa de todos los bytes por debajo de 128. Si la codificación del sistema de archivos no proporciona esta garantía, las funciones de API pueden lanzar `UnicodeError`.

Las funciones `sys.getfilesystemencoding()` y `sys.getfilesystemencodeerrors()` se pueden utilizar para obtener la codificación del sistema de archivos y el controlador de errores.

La *codificación del sistema de archivos y el manejador de errores* se configuran al inicio de Python mediante la función `PyConfig_Read()`: consulte los miembros `filesystem_encoding` y `filesystem_errors` de `PyConfig`.

Vea también *locale encoding*.

buscador
Un objeto que trata de encontrar el *loader* para el módulo que está siendo importado.

There are two types of finder: *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See finders-and-loaders and `importlib` for much more detail.

división entera a la baja
Una división matemática que se redondea hacia el entero menor más cercano. El operador de la división entera a la baja 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**.

free threading
A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

free variable
Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

función
Una serie de sentencias que retornan un valor al que las llama. También se le puede pasar cero o más *argumentos* los cuales pueden ser usados en la ejecución de la misma. Vea también *parameter*, *method*, y la sección function.

anotación de función
Una *annotation* del parámetro de una función o un valor de retorno.

Las anotaciones de funciones son usadas frecuentemente para *indicadores de tipo*, por ejemplo, se espera que una función tome dos argumentos de clase `int` y también se espera que retorne dos valores `int`:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

La sintaxis de las anotaciones de funciones son explicadas en la sección function.

Consulte *variable annotation* y **PEP 484**, que describen esta funcionalidad. Consulte también annotations-howto para conocer las mejores prácticas sobre cómo trabajar con anotaciones.

__future__
Un future statement, `from __future__ import <feature>`, indica al compilador que compile el módulo actual utilizando una sintaxis o semántica que se convertirá en estándar en una versión futura de Python. El módulo `__future__` documenta los posibles valores de *feature*. Al importar este módulo y evaluar sus variables, puede ver cuándo se agregó por primera vez una nueva característica al lenguaje y cuándo se convertirá (o se convirtió) en la predeterminada:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

recolección de basura
El proceso de liberar la memoria de lo que ya no está en uso. Python realiza recolección de basura (*garbage collection*) llevando la cuenta de las referencias, y el recogedor de basura cíclico es capaz de detectar y romper las referencias cíclicas. El recogedor de basura puede ser controlado mediante el módulo `gc` .

generador
Una función que retorna un *generator iterator*. Luce como una función normal excepto que contiene la expresión `yield` para producir series de valores utilizables en un bucle *for* o que pueden ser obtenidas una por una con la función `next()`.

Usualmente se refiere a una función generadora, pero puede referirse a un *iterador generador* en ciertos contextos. En aquellos casos en los que el significado no está claro, usar los términos completos evita la ambigüedad.

iterador generador
Un objeto creado por una función *generator*.

Each `yield` temporarily suspends processing, remembering the execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

expresión generadora
An *expression* that returns an *iterator*. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

función genérica
Una función compuesta de muchas funciones que implementan la misma operación para diferentes tipos. Qué implementación deberá ser usada durante la llamada a la misma es determinado por el algoritmo de despacho.

Vea también la entrada de glosario *single dispatch*, el decorador `functools.singledispatch()`, y **PEP 443**.

tipos genéricos
Un *type* que se puede parametrizar; normalmente un container class como `list` o `dict`. Usado para *type hints* y *annotations*.

Para más detalles, véase generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, y el módulo `typing`.

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.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

In prior versions of Python's C API, a function might declare that it requires the GIL to be held in order to use it. This refers to having an *attached thread state*.

hash-based pyc
Un archivo cache de *bytecode* que usa el *hash* en vez de usar el tiempo de la última modificación del archivo fuente correspondiente para determinar su validez. Vea pyc-invalidation.

hashable
An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

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
Un Entorno Integrado de Desarrollo y Aprendizaje para Python. idle es un editor básico y un entorno de intérprete que se incluye con la distribución estándar de Python.

immortal
Immortal objects are a CPython implementation detail introduced in **PEP 683**.

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

Immortal objects can be identified via `sys._is_immortal()`, or via `PyUnstable_IsImmortal()` in the C API.

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 has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`). For more on interactive mode, see *Modo interactivo*.

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.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, …). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterador
An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

Puede encontrar más información en typeiter.

Detalles de implementación de CPython: CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

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])`. Además, `operator.attrgetter()`, `operator.itemgetter()` y `operator.methodcaller()` son tres constructores de funciones clave. Consulte Sorting HOW TO para ver ejemplos de cómo crear y utilizar 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.

lexical analyzer
Formal name for the *tokenizer*; see *token*.

lista
A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is *O*(1).

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
An object that loads a module. It must define the `exec_module()` and `create_module()` methods to implement the `Loader` interface. A loader is typically returned by a *finder*. See also:

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

codificación de la configuración regional
En Unix, es la codificación de la configuración regional LC_CTYPE. Se puede configurar con `locale.setlocale(locale.LC_CTYPE, new_locale)`.

En Windows, es la página de códigos ANSI (por ejemplo, `"cp1252"`).

En Android y VxWorks, Python utiliza `"utf-8"` como codificación regional.

`locale.getencoding()` can be used to get the locale encoding.

Vea también *filesystem encoding and error handler*.

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 `collections.abc.Mapping` o `collections.abc.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
Method Resolution Order is the order in which base classes are searched for a member during lookup. See python_2.3_mro for details of the algorithm used by the Python interpreter since the 2.3 release.

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`.

See also module-specs.

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
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

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
A *package* which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a *regular package* because they have no `__init__.py` file.

Namespace packages allow several individually installable packages to have a common parent package. Otherwise, it is recommended to use a *regular package*.

For more information, see **PEP 420** and reference-namespace-package.

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
Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

objeto
Cualquier dato con estado (atributo o valor) y comportamiento definido (métodos). También es la más básica clase base para cualquier *new-style class*.

optimized scope
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

paquete
Un *module* Python que puede contener submódulos o recursivamente, subpaquetes. Técnicamente, un paquete es un módulo Python con un atributo `__path__`.

Vea también *regular package* y *namespace package*.

parámetro
Una entidad nombrada en una definición de una *function* (o método) que especifica un *argument* (o en algunos casos, varios argumentos) que la función puede aceptar. Existen cinco tipos de argumentos:

- *posicional o nombrado*: especifica un argumento que puede ser pasado tanto como *posicional* o como *nombrado*. Este es el tipo por defecto de parámetro, como *foo* y *bar* en el siguiente ejemplo:

```
def func(foo, bar=None): ...
```

- *sólo posicional*: especifica un argumento que puede ser pasado sólo por posición. Los parámetros sólo posicionales pueden ser definidos incluyendo un carácter `/` en la lista de parámetros de la función después de ellos, como *posonly1* y *posonly2* en el ejemplo que sigue:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *sólo nombrado*: especifica un argumento que sólo puede ser pasado por nombre. Los parámetros sólo por nombre pueden ser definidos incluyendo un parámetro posicional de una sola variable o un simple `*`` antes de ellos en la lista de parámetros en la definición de la función, como *kw_only1* y *kw_only2* en el ejemplo siguiente:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *variable posicional*: especifica una secuencia arbitraria de argumentos posicionales que pueden ser brindados (además de cualquier argumento posicional aceptado por otros parámetros). Este parámetro puede ser definido anteponiendo al nombre del parámetro `*`, como a *args* en el siguiente ejemplo:

```
def func(*args, **kwargs): ...
```

- *variable nombrado*: especifica que arbitrariamente muchos argumentos nombrados pueden ser brindados (además de cualquier argumento nombrado ya aceptado por cualquier otro parámetro). Este parámetro puede ser definido anteponiendo al nombre del parámetro con `**`, como *kwargs* en el ejemplo precedente.

Los parámetros puede especificar tanto argumentos opcionales como requeridos, así como valores por defecto para algunos argumentos opcionales.

Vea también el glosario de *argument*, la pregunta respondida en la diferencia entre argumentos y parámetros, la clase `inspect.Parameter`, la sección function , y **PEP 362**.

entrada de ruta
Una ubicación única en el *import path* que el *path based finder* consulta para encontrar los módulos a importar.

buscador de entradas de ruta
Un *finder* retornado por un invocable en `sys.path_hooks` (esto es, un *path entry hook*) que sabe cómo localizar módulos dada una *path entry*.

Vea en `importlib.abc.PathEntryFinder` los métodos que los buscadores de entradas de ruta implementan.

gancho a entrada de ruta
A callable on the `sys.path_hooks` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

buscador basado en ruta
Uno de los *meta buscadores de ruta* por defecto que busca un *import path* para los módulos.

objeto tipo ruta
Un objeto que representa una ruta del sistema de archivos. Un objeto tipo ruta puede ser tanto una `str` como un `bytes` representando una ruta, o un objeto que implementa el protocolo `os.PathLike`. Un objeto que soporta el protocolo `os.PathLike` puede ser convertido a ruta del sistema de archivo de clase `str` o `bytes` usando la función `os.fspath()`; `os.fsdecode()` `os.fsencode()` pueden emplearse para garantizar que retorne respectivamente `str` o `bytes`. Introducido por **PEP 519**.

PEP
Propuesta de mejora de Python, del inglés *Python Enhancement Proposal*. Un PEP es un documento de diseño que brinda información a la comunidad Python, o describe una nueva capacidad para Python, sus procesos o entorno. Los PEPs deberían dar una especificación técnica concisa y una fundamentación para las capacidades propuestas.

Los PEPs tienen como propósito ser los mecanismos primarios para proponer nuevas y mayores capacidad, para recoger la opinión de la comunidad sobre un tema, y para documentar las decisiones de diseño que se han hecho en Python. El autor del PEP es el responsable de lograr consenso con la comunidad y documentar las opiniones disidentes.

Vea **PEP 1**.

porción
Un conjunto de archivos en un único directorio (posiblemente guardo en un archivo comprimido *zip*) que contribuye a un espacio de nombres de paquete, como está definido en **PEP 420**.

argumento posicional
Vea *argument*.

API provisional
Una API provisoria es aquella que deliberadamente fue excluida de las garantías de compatibilidad hacia atrás de la biblioteca estándar. Aunque no se esperan cambios fundamentales en dichas interfaces, como están marcadas como provisionales, los cambios incompatibles hacia atrás (incluso remover la misma interfaz) podrían

ocurrir si los desarrolladores principales lo estiman. Estos cambios no se hacen gratuitamente – solo ocurrirán si fallas fundamentales y serias son descubiertas que no fueron vistas antes de la inclusión de la API.

Incluso para APIs provisorias, los cambios incompatibles hacia atrás son vistos como una «solución de último recurso» - se intentará todo para encontrar una solución compatible hacia atrás para los problemas identificados.

Este proceso permite que la biblioteca estándar continúe evolucionando con el tiempo, sin bloquearse por errores de diseño problemáticos por períodos extensos de tiempo. Vea **PEP 411** para más detalles.

paquete provisorio
Vea *provisional API*.

Python 3000
Apodo para la fecha de lanzamiento de Python 3.x (acuñada en un tiempo cuando llegar a la versión 3 era algo distante en el futuro.) También se lo abrevió como *Py3k*.

Pythónico
Una idea o pieza de código que sigue ajustadamente la convenciones idiomáticas comunes del lenguaje Python, en vez de implementar código usando conceptos comunes a otros lenguajes. Por ejemplo, una convención común en Python es hacer bucles sobre todos los elementos de un iterable con la sentencia `for`. Muchos otros lenguajes no tienen este tipo de construcción, así que los que no están familiarizados con Python podrían usar contadores numéricos:

```
for i in range(len(food)):
    print(food[i])
```

En contraste, un método Pythónico más limpio:

```
for piece in food:
    print(piece)
```

nombre calificado
Un nombre con puntos mostrando la ruta desde el alcance global del módulo a la clase, función o método definido en dicho módulo, como se define en **PEP 3155**. Para las funciones o clases de más alto nivel, el nombre calificado es el igual al nombre del objeto:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Cuando es usado para referirse a los módulos, *nombre completamente calificado* significa la ruta con puntos completo al módulo, incluyendo cualquier paquete padre, por ejemplo, `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

contador de referencias
The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are *immortal* and have reference counts that are never modified, and therefore the objects are never deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. Programmers can call the `sys.getrefcount()` function to return the reference count for a particular object.

In *CPython*, reference counts are not considered to be stable or well-defined values; the number of references to an object, and how that number is affected by Python code, may be different between versions.

paquete regular
Un *package* tradicional, como aquellos con un directorio conteniendo el archivo `__init__.py`.

Vea también *namespace package*.

REPL
An acronym for the «read–eval–print loop», another name for the *interactive* interpreter shell.

__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
An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *hashable* keys rather than integers.

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()`. For more documentation on sequence methods generally, see Common Sequence Operations.

comprensión de conjuntos
Una forma compacta de procesar todos o parte de los elementos en un iterable y retornar un conjunto con los resultados. `results = {c for c in 'abracadabra' if c not in 'abc'}` genera el conjunto de cadenas `{'r', 'd'}`. Ver comprehensions.

despacho único
Una forma de despacho de una *generic function* donde la implementación es elegida a partir del tipo de un sólo argumento.

rebanada
Un objeto que contiene una porción de una *sequence*. Una rebanada es creada usando la notación de suscripto, `[]` con dos puntos entre los números cuando se ponen varios, como en `nombre_variable[1:3:5]`. La notación con corchete (suscrito) usa internamente objetos `slice`.

soft deprecated
A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See PEP 387: Soft Deprecation.

método especial
Un método que es llamado implícitamente por Python cuando ejecuta ciertas operaciones en un tipo, como la adición. Estos métodos tienen nombres que comienzan y terminan con doble barra baja. Los métodos especiales están documentados en specialnames.

standard library
The collection of *packages*, *modules* and *extension modules* distributed as a part of the official Python interpreter package. The exact membership of the collection may vary based on platform, available system libraries, or other criteria. Documentation can be found at library-index.

See also `sys.stdlib_module_names` for a list of all possible standard library module 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`.

static type checker
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

stdlib
An abbreviation of *standard library*.

referencia fuerte
En la API de C de Python, una referencia fuerte es una referencia a un objeto que es propiedad del código que mantiene la referencia. La referencia fuerte se toma llamando a `Py_INCREF()` cuando se crea la referencia y se libera con `Py_DECREF()` cuando se elimina la referencia.

La función `Py_NewRef()` se puede utilizar para crear una referencia fuerte a un objeto. Por lo general, se debe llamar a la función `Py_DECREF()` en la referencia fuerte antes de salir del alcance de la referencia fuerte, para evitar filtrar una referencia.

Consulte también *borrowed reference*.

t-string
t-strings
String literals prefixed with `t` or `T` are commonly called «t-strings» which is short for template string literals.

codificación de texto
Una cadena de caracteres en Python es una secuencia de puntos de código Unicode (en el rango `U+0000–U+10FFFF`). Para almacenar o transferir una cadena de caracteres, es necesario serializarla como una secuencia de bytes.

La serialización de una cadena de caracteres en una secuencia de bytes se conoce como «codificación», y la recreación de la cadena de caracteres a partir de la secuencia de bytes se conoce como «decodificación».

Existe una gran variedad de serializaciones de texto codecs, que se denominan colectivamente «codificaciones de texto».

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*.

thread state
The information used by the *CPython* runtime to run in an OS thread. For example, this includes the current exception, if any, and the state of the bytecode interpreter.

Each thread state is bound to a single OS thread, but threads may have many thread states available. At most, one of them may be *attached* at once.

An *attached thread state* is required to call most of Python's C API, unless a function explicitly documents otherwise. The bytecode interpreter only runs under an attached thread state.

Each thread state belongs to a single interpreter, but each interpreter may have many thread states, including multiple for the same OS thread. Thread states from multiple interpreters may be bound to the same thread, but only one can be *attached* in that thread at any given moment.

See Thread State and the Global Interpreter Lock for more information.

token
A small unit of source code, generated by the lexical analyzer (also called the *tokenizer*). Names, numbers, strings, operators, newlines and similar are represented by tokens.

The `tokenize` module exposes Python's lexical analyzer. The `token` module contains information on the various types of tokens.

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
The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

alias de tipos
Un sinónimo para un tipo, creado al asignar un tipo a un identificador.

Los alias de tipos son útiles para simplificar los *indicadores de tipo*. Por ejemplo:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

podría ser más legible así:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

Vea `typing` y **PEP 484**, que describen esta funcionalidad.

indicador de tipo
Una *annotation* que especifica el tipo esperado para una variable, un atributo de clase, un parámetro para una función o un valor de retorno.

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

Los indicadores de tipo de las variables globales, atributos de clase, y funciones, no de variables locales, pueden ser accedidos usando `typing.get_type_hints()`.

Vea `typing` y **PEP 484**, que describen esta funcionalidad.

saltos de líneas universales
Una manera de interpretar flujos de texto en la cual son reconocidos como finales de línea todas siguientes formas: la convención de Unix para fin de línea `'\n'`, la convención de Windows `'\r\n'`, y la vieja convención de Macintosh `'\r'`. Vea **PEP 278** y **PEP 3116**, además de `bytes.splitlines()` para usos adicionales.

anotación de variable
Una *annotation* de una variable o un atributo de clase.

Cuando se anota una variable o un atributo de clase, la asignación es opcional:

```
class C:
    field: 'annotation'
```

Las anotaciones de variables son frecuentemente usadas para *type hints*: por ejemplo, se espera que esta variable tenga valores de clase `int`:

```
count: int = 0
```

La sintaxis de la anotación de variables está explicada en la sección annassign.

Consulte *function annotation*, **PEP 484** y **PEP 526**, que describen esta funcionalidad. Consulte también annotations-howto para conocer las mejores prácticas sobre cómo trabajar con anotaciones.

entorno virtual
Un entorno cooperativamente aislado de ejecución que permite a los usuarios de Python y a las aplicaciones instalar y actualizar paquetes de distribución de Python sin interferir con el comportamiento de otras aplicaciones de Python en el mismo sistema.

Vea también `venv`.

máquina virtual
: Una computadora definida enteramente por software. La máquina virtual de Python ejecuta el *bytecode* generado por el compilador de *bytecode*.

walrus operator
: A light-hearted way to refer to the assignment expression operator `:=` because it looks a bit like a walrus if you turn your head.

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

About this documentation

Python's documentation is generated from reStructuredText sources using Sphinx, a documentation generator originally created for Python and now maintained as an independent project.

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., the creator of the original Python documentation toolset and author of much of the content;
- el proyecto Docutils para creación de reStructuredText y la suite Docutils;
- Fredrik Lundh por su proyecto Referencia Alternativa de Python del que Sphinx obtuvo muchas buenas ideas.

B.1 Contributors to the Python documentation

Muchas personas han contribuido para el lenguaje de Python, la librería estándar de Python, y la documentación de Python. Revisa Misc/ACKS la distribución de Python para una lista parcial de contribuidores.

Es solamente con la aportación y contribuciones de la comunidad de Python que Python tiene tan fantástica documentación – Muchas gracias!

APÉNDICE C

Historia y Licencia

C.1 Historia del software

Python 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, which became Zope Corporation. 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 was 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.

Lanzamiento	Derivado de	Año	Dueño/a	GPL-compatible? (1)
0.9.0 hasta 1.2	n/a	1991-1995	CWI	sí
1.3 hasta 1.5.2	1.2	1995-1999	CNRI	sí
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	yes (2)
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	sí
2.1.1	2.1+2.0.1	2001	PSF	sí
2.1.2	2.1.1	2002	PSF	sí
2.1.3	2.1.2	2002	PSF	sí
2.2 y superior	2.1.1	2001-ahora	PSF	sí

Nota

(1) 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.

(2) According to Richard Stallman, 1.6.1 is not GPL-compatible, because its license has a choice of law clause. According to CNRI, however, Stallman's lawyer has told CNRI's lawyer that 1.6.1 is «not incompatible» with the GPL.

Gracias a los muchos voluntarios externos que han trabajado bajo la dirección de Guido para hacer posibles estos lanzamientos.

C.2 Términos y condiciones para acceder o usar Python

Python software and documentation are licensed under the Python Software Foundation License Version 2.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Version 2 and the *Zero-Clause BSD license*.

Parte del software incorporado en Python está bajo diferentes licencias. Las licencias se enumeran con el código correspondiente a esa licencia. Consulte *Licencias y reconocimientos para software incorporado* para obtener una lista incompleta de estas licencias.

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") 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 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001 Python Software Foundation; All Rights
   Reserved" are retained in Python 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 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.

4. PSF is making Python 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 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, 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
```

(continúe en la próxima página)

(proviene de la página anterior)

```
   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, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 ACUERDO DE LICENCIA DE BEOPEN.COM PARA PYTHON 2.0

ACUERDO DE LICENCIA DE CÓDIGO ABIERTO DE BEOPEN PYTHON VERSIÓN 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 ACUERDO DE LICENCIA CNRI PARA PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   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.
```

(continúe en la próxima página)

(proviene de la página anterior)

```
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 ACUERDO DE LICENCIA CWI PARA PYTHON 0.9.0 HASTA 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 Licencias y reconocimientos para software incorporado

Esta sección es una lista incompleta, pero creciente, de licencias y reconocimientos para software de terceros incorporado en la distribución de Python.

C.3.1 Mersenne Twister

La extensión C `_random` subyacente al módulo `random` incluye código basado en una descarga de http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. Los siguientes son los comentarios textuales del código original:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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

El módulo `socket` usa las funciones, `getaddrinfo()`, y `getnameinfo()`, que están codificadas en archivos fuente separados del Proyecto WIDE, 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.
```

(continúe en la próxima página)

(proviene de la página anterior)

```
THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Servicios de socket asincrónicos

Los módulos `test.support.asynchat` y `test.support.asyncore` contienen el siguiente aviso:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Gestión de cookies

El módulo `http.cookies` contiene el siguiente aviso:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
```

(continúe en la próxima página)

(proviene de la página anterior)

```
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Seguimiento de ejecución

El módulo `trace` contiene el siguiente aviso:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 funciones UUencode y UUdecode

The `uu` codec 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
```

(continúe en la próxima página)

(proviene de la página anterior)

```
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 Llamadas a procedimientos remotos XML

El módulo `xmlrpc.client` contiene el siguiente aviso:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

El módulo `test.test_epoll` contiene el siguiente aviso:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:
```

(continúe en la próxima página)

(proviene de la página anterior)

```
The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Seleccionar kqueue

El módulo `select` contiene el siguiente aviso para la interfaz kqueue:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

El archivo `Python/pyhash.c` contiene la implementación de Marek Majkowski del algoritmo SipHash24 de Dan Bernstein. Contiene la siguiente nota:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
```

(continúe en la próxima página)

(proviene de la página anterior)

```
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod y dtoa

El archivo `Python/dtoa.c`, que proporciona las funciones de C dtoa y strtod para la conversión de dobles C hacia y desde cadenas de caracteres, se deriva del archivo del mismo nombre por David M. Gay, actualmente disponible en https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. El archivo original, recuperado el 16 de marzo de 2009, contiene el siguiente aviso de licencia y derechos de autor:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.
```

(continúe en la próxima página)

(proviene de la página anterior)

```
   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
```

(continúe en la próxima página)

(proviene de la página anterior)

```
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
```

(continúe en la próxima página)

(proviene de la página anterior)

```
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

La extensión `pyexpat` se construye usando una copia incluida de las fuentes de expatriados a menos que la construcción esté configurada `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

La extensión C `_ctypes` subyacente al módulo `ctypes` se construye usando una copia incluida de las fuentes de libffi a menos que la construcción esté configurada `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

La extensión `zlib` se crea utilizando una copia incluida de las fuentes de zlib si la versión de zlib encontrada en el sistema es demasiado antigua para ser utilizada para la compilación:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

La implementación de la tabla hash utilizada por `tracemalloc` se basa en el proyecto cfuhash:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    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,
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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

La extension C `_decimal` subyacente al módulo `decimal` se construye usando una copia incluida de la biblioteca libmpdec a menos que la construcción esté configurada `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 Conjunto de pruebas W3C C14N

El conjunto de pruebas C14N 2.0 en el paquete `test` (`Lib/test/xmltestdata/c14n-20/`) se recuperó del sitio web de W3C en https://www.w3.org/TR/xml-c14n2-testcases/ y se distribuye bajo la licencia BSD de 3 cláusulas:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* 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
```

(continúe en la próxima página)

(proviene de la página anterior)

```
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT License:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

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.3.20 asyncio

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

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.
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's «Global Unbounded Sequences» safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

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 unmodified, 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 "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 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.3.22 Zstandard bindings

Zstandard bindings in `Modules/_zstd` and `Lib/compression/zstd` are based on code from the pyzstd library, copyright Ma Lin and contributors. The pyzstd code is distributed under the 3-Clause BSD License:

```
Copyright (c) 2020-present, Ma Lin and contributors.
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 copyright holder nor the names of its
   contributors may be used to endorse or promote products derived from
```

(continúe en la próxima página)

(proviene de la página anterior)

```
   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 HOLDER 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

Derechos de autor

Python y esta documentación es:

Copyright © 2001 Python Software Foundation. All rights reserved.

Derechos de autor © 2000 BeOpen.com. Todos los derechos reservados.

Derechos de autor © 1995-2000 Corporation for National Research Initiatives. Todos los derechos reservados.

Derechos de autor © 1991-1995 Stichting Mathematisch Centrum. Todos los derechos reservados.

Consulte *Historia y Licencia* para obtener información completa sobre licencias y permisos.

Índice

No alfabético

. . ., **125**
    ellipsis literal, 22, 125
(*hash*)
    comentario, 9
* (*asterisco*)
    in function calls, 32
**
    in function calls, 33
: (*dos puntos*)
    function annotations, 34
->
    function annotations, 34
>>>, **125**
__all__, 56
__future__, **132**
__slots__, **140**

A

a la espera, **127**
administrador asincrónico de contexto, **126**
administrador de contextos, **128**
alcances anidados, **137**
alias de tipos, **142**
alteración
    nombre, 88
annotate function, **125**
annotations
    function, 34
anotación, **125**
anotación de función, **131**
anotación de variable, **142**
apagado del intérprete, **134**
API provisional, **138**
archivo binario, **127**
archivo de texto, **141**
argumento, **125**
argumento nombrado, **135**
argumento posicional, **138**
atributo, **127**
attached thread state, **126**
ayuda
    función incorporada, 93

B

BDFL, **127**
bloqueo global del intérprete, **132**
builtins
    module, 54
buscador, **131**
buscador basado en ruta, **138**
buscador de entradas de ruta, **138**
bytecode, **127**

C

cadena con triple comilla, **141**
callable, **127**
cargador, **135**
C-contiguous, 129
clase, **128**
clase base abstracta, **125**
clase de nuevo estilo, **137**
closure variable, **128**
codificación de la configuración regional, **135**
codificación de texto, **141**
codificación del sistema de archivos y manejador de errores, **131**
coding
    style, 34
comprensión de conjuntos, **140**
comprensión de diccionarios, **130**
comprensión de listas, **135**
contador de referencias, **139**
context, **128**
context management protocol, **128**
contiguo, **129**
corrutina, **129**
CPython, **129**
current context, **129**
cyclic isolate, **129**

D

decorador, **129**
descriptor, **129**
despacho único, **140**
diccionario, **130**

división entera a la baja, **131**
docstring, **130**
docstrings, 26, 33
documentation strings, 26, 33
dunder, **130**

E

EAFP, **130**
entorno virtual, **142**
entrada de ruta, **138**
espacio de nombres, **136**
especificador de módulo, **136**
evaluate function, **130**
expresión, **130**
expresión generadora, **132**

F

f-string, **130**
f-strings, **130**
file
    object, 63
for
    statement, 19
formatted string literal, 60
Fortran contiguous, 129
free threading, **131**
free variable, **131**
fstring, 60
f-string, 60
función, **131**
función clave, **134**
función corrutina, **129**
función genérica, **132**
función incorporada
    ayuda, 93
    open, 63
function
    annotations, 34

G

gancho a entrada de ruta, **138**
generador, **132**
generador asincrónico, **126**
GIL, **132**

H

hash-based pyc, **133**
hashable, **133**

I

IDLE, **133**
immortal, **133**
importador, **133**
importar, **133**
indicador de tipo, **142**
inmutable, **133**
interactivo, **133**
interpolated string literal, 60
interpretado, **134**
iterable, **134**
iterable asincrónico, **126**
iterador, **134**
iterador asincrónico, **126**
iterador generador, **132**
iterador generador asincrónico, **126**

J

json
    módulo, 65

L

lambda, **135**
LBYL, **135**
lexical analyzer, **135**
lista, **135**

M

magic
    método, 135
mapeado, **135**
máquina virtual, **143**
meta buscadores de ruta, **135**
metaclase, **135**
método, **136**
    magic, 135
    objeto, 83
    special, 140
método especial, **140**
método mágico, **135**
module
    builtins, 54
    search path, 51
    sys, 52
módulo, **136**
    json, 65
módulo de extensión, **130**
MRO, **136**
mutable, **136**

N

nombre
    alteración, 88
nombre calificado, **139**
número complejo, **128**

O

object
    file, 63
objeto, **137**
    método, 83
objeto archivo, **130**
objeto tipo ruta, **138**
objetos tipo archivo, **131**
objetos tipo binarios, **127**
open
    función incorporada, 63

optimized scope, **137**
orden de resolución de métodos, **136**

P

paquete, **137**
paquete de espacios de nombres, **137**
paquete provisorio, **139**
paquete regular, **140**
parámetro, **137**
path
    module search, 51
PATH, 51, 122
PEP, **138**
porción, **138**
Python 3000, **139**
Python Enhancement Proposals
    PEP 1, 138
    PEP 8, 34
    PEP 238, 131
    PEP 278, 142
    PEP 302, 135
    PEP 343, 128
    PEP 362, 126, 138
    PEP 411, 139
    PEP 420, 137, 138
    PEP 443, 132
    PEP 483, 132
    PEP 484, 34, 125, 132, 142
    PEP 492, 126, 127, 129
    PEP 498, 130
    PEP 519, 138
    PEP 525, 126
    PEP 526, 125, 142
    PEP 585, 132
    PEP 636, 25
    PEP 649, 125
    PEP 683, 133
    PEP 703, 131, 133
    PEP 765, 73
    PEP 3107, 34
    PEP 3116, 142
    PEP 3147, 52
    PEP 3155, 139
PYTHON_BASIC_REPL, 121
PYTHON_GIL, 133
Pythónico, **139**
PYTHONPATH, 51, 53
PYTHONSTARTUP, 122

R

rebanada, **140**
recolección de basura, **132**
referencia fuerte, **141**
referencia prestada, **127**
REPL, **140**
retrollamada, **128**
RFC
    RFC 2822, 98
ruta de importación, **133**

S

saltos de líneas universales, **142**
search
    path, module, 51
secuencia, **140**
sentencia, **140**
sitecustomize, 122
soft deprecated, **140**
special
    método, 140
standard library, **140**
statement
    for, 19
static type checker, **141**
stdlib, **141**
string
    formatted literal, 60
    interpolated literal, 60
strings, documentation, 26, 33
style
    coding, 34
sys
    module, 52

T

t-string, **141**
t-strings, **141**
thread state, **141**
tipado de pato, **130**
tipo, **142**
tipos genéricos, **132**
token, **141**
tupla nombrada, **136**

U

usercustomize, 122

V

variable de clase, **128**
variable de contexto, **128**
variables de entorno
    PATH, 51, 122
    PYTHON_BASIC_REPL, 121
    PYTHON_GIL, 133
    PYTHONPATH, 51, 53
    PYTHONSTARTUP, 122
vista de diccionario, **130**

W

walrus operator, **143**

Z

Zen de Python, **143**