Tutorial de Python

Versión 3.10

Guido van Rossum
y equipo de desarrollo de Python

febrero, 2022

Python Software Foundation
Email: docs@python.org

Índice general

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.

Este tutorial introduce al lector informalmente a los conceptos básicos y las funcionalidades del lenguaje de programación Python y a su sistema. Ayuda tener un interprete de Python accesible para una experiencia práctica, todos los ejemplos son auto-contenidos, permitiendo utilizar el tutorial sin conexión.

No busca 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 leer este material podrás leer y escribir módulos y programas en Python. ¡Estarás listo para aprender más acerca de las diversas bibliotecas y módulos!

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.

Puede escribir un script de shell de Unix o archivos por lotes de Windows para algunas de estas tareas, pero los scripts de shell son mejores para mover archivos y cambiar datos de texto, no son adecuados para juegos o aplicaciones GUI. Podría escribir un programa C / C ++ / Java, pero puede llevar mucho tiempo de desarrollo obtener incluso un programa de primer borrador. Python es más fácil de usar, está disponible en los sistemas operativos Windows, macOS y Unix,

y lo ayudará a hacer el trabajo más rápidamente.

Python es fácil de utilizar siendo un lenguaje de programación real ofreciendo mucha más estructura y soporte para programas grandes que la que ofrecen shell scripts o ficheros batch. 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 Invocando al intérprete

El intérprete de Python generalmente se instala como `/usr/local/bin/python3.10` en aquellas máquinas donde está disponible; poner `/usr/local/bin` en la ruta de búsqueda de su shell de Unix hace posible iniciarlo escribiendo el comando:

```
python3.10
```

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 experto Python local o administrador de sistemas. (Por ejemplo, `/usr/local/python` es una alternativa popular).

En máquinas con Windows en las que haya instalado Python desde Microsoft Store, el comando `python3.10` estará disponible. Si tiene el lanzador py.exe instalado, puede usar el comando `py`. Consulte setting-envvars para conocer otras formas de iniciar Python.

[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.

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 GNU Readline librería. Quizás la forma más rápida para comprobar si las características de edición se encuentran disponibles es escribir `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 no parece que ocurra 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* con comillas simples.

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

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` *comand* 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*:

```
$ python3.10
Python 3.10 (default, June 4 2019, 09:25:04)
[GCC 4.8.2] 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 *codificación* 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 línea *UNIX «shebang» line*. En ese caso, la declaración de codificación debe agregarse como la segunda línea del archivo. Por ejemplo:

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

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 un prompt secundario solo en una linea de ejemplo 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 espere el prompt primario, >>>. (No debería tardar mucho.)

3.1.1 Números

El intérprete puede utilizarse como una simple calculadora; puedes introducir una expresión y este escribirá los valores. La sintaxis es sencilla: los operadores `+`, `-`, `*` y `/`` funcionan como en la mayoría de los lenguajes (por ejemplo, Pascal o C); los paréntesis (``()) pueden ser usados para agrupar. Por ejemplo:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 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 punto flotante. Para hacer *floor division* y obtener un resultado entero (descartando cualquier resultado fraccionario) 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. Ningún resultado se mostrará 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:

[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`.

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

Además de números, Python puede manipular cadenas de texto, las cuales pueden ser expresadas de distintas formas. Pueden estar encerradas en comillas simples (`'...'`) o dobles (`"..."`) con el mismo resultado[2]. `\` puede ser usado para escapar comillas:

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> '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 interactivo, la salida de caracteres está encerrada en comillas y los caracteres especiales se escapan con barras invertidas. Aunque esto a veces se vea diferente de la entrada (las comillas que encierran pueden cambiar), las dos cadenas son equivalentes. La cadena se encierra

[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.

en comillas dobles si la cadena contiene una comilla simple y ninguna doble, de lo contrario es encerrada en comillas simples. La función `print()` produce una salida más legible, omitiendo las comillas que la encierran e imprimiendo caracteres especiales y escapados:

```
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
>>> print('"Isn\'t," they said.')
"Isn't," they said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a 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
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

Las cadenas de texto literales pueden contener múltiples líneas. Una forma es usar triples comillas: `"""..."""` o `'''...'''`. Los fin de línea son incluidos automáticamente, pero es posible prevenir esto agregando una `\` al final de la línea. Por ejemplo:

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

produce la siguiente salida (tened en cuenta que la línea inicial no está incluida):

```
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'
'unununium'
```

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 quizás sean números negativos, para empezar a contar desde la derecha:

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

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

Además de los índices, las *rebanadas* también están soportadas. Mientras que los índices se utilizar para obtener caracteres individuales, las *rebanadas* te permiten obtener partes de las cadenas de texto:

```
>>> 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'
```

Nota 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]
```

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:

```
>>> squares[:]
[1, 4, 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]
```

También puede agregar nuevos elementos al final de la lista, utilizando el *método* `append()` (vamos a ver más sobre los métodos luego):

```
>>> 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]
```

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

Por supuesto, podemos usar Python para tareas más complicadas que sumar dos y dos. Por ejemplo, podemos escribir una parte inicial de la serie de Fibonacci <https://en.wikipedia.org/wiki/Fibonacci_number>`_ así:

```
>>> # 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
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.

- La función `print()` escribe el valor de los argumentos que se le dan. Difiere de simplemente escribir la expresión que se quiere mostrar (como hicimos antes en los ejemplos de la calculadora) en la forma en que maneja múltiples argumentos, cantidades de punto flotante y cadenas. Las cadenas de texto son impresas sin comillas y un espacio en blanco se inserta entre los elementos, así puedes formatear cosas de una forma agradable:

```
>>> 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,
```

CAPÍTULO 4

Más herramientas para control de flujo

Además de la sentencia `while` que acabamos de introducir, Python soporta las sentencias de control de flujo que podemos encontrar en otros lenguajes, con algunos cambios.

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 keyword:*!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()`, mira *Técnicas de iteración*.

Algo extraño sucede si muestras un ` *range*`:

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

De muchas maneras el objeto retornado por `range()` se comporta 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 Las sentencias `break`, `continue`, y `else` en bucles

La sentencia `break`, como en C, termina el bucle `for` o `while` más anidado.

Las sentencias de bucle pueden tener una cláusula`!else` que es ejecutada cuando el bucle termina, después de agotar el iterable (con `for`) o cuando la condición se hace falsa (con `while`), pero no cuando el bucle se termina con la sentencia `break`. Se puede ver el ejemplo en el siguiente bucle, 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
```

(Sí, este es el código correcto. Fíjate bien: el `else` pertenece al ciclo `for`, no al `if`.)

Cuando se usa con un bucle, la cláusula `else` tiene más en común con el `else` de una sentencia `try` que con el de un `if`: en una sentencia `try` la cláusula `else` se ejecuta cuando no se genera ninguna excepción, y el `else` de un bucle se ejecuta cuando no hay ningún `break`. Para más sobre la declaración `try` y excepciones, mira *Gestionando excepciones*.

La declaración `continue`, también tomada de C, continua con la siguiente iteración del ciclo:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("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
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5 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!
...
```

4.6 La sentencia `match`

Una sentencia match recibe una expresión y compara su valor a patrones sucesivos que aparecen en uno o más bloques case. Esto es similar a grandes rasgos con una sentencia switch en C, Java o JavaScript (y muchos otros lenguajes), pero también es capaz de extraer componentes (elementos de una secuencia o atributos de un objeto) de un valor y ponerlos en variables.

La forma más simple compara un valor, el «sujeto», 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"
```

Observa el último bloque: el «nombre de variable» _ funciona como un *comodín* y nunca fracasa la coincidencia. Si ninguno de los casos `case` coincide, ninguna de las ramas es ejecutada.

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

```
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 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:
    x: int
    y: int

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 cuales variables tomarían que 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, podríamos aplicar match así:

```
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)]:
        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 el mismo significado y en realidad coinciden con cualquier secuencia. 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]` and `(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 del 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 en el sujeto son ignoradas. Puede usarse un desempaquetado como `**rest`. (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 los singletons `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 = 0
    GREEN = 1
    BLUE = 2

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.

4.7 Definiendo funciones

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

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to 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, y hacerlo un buen hábito.

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 (argumentos) reales para una llamada de función se introducen en la tabla de símbolos local de la función llamada cuando se llama; por lo tanto, los argumentos se pasan usando *llamada por valor* (donde el *valor* es siempre un objeto *referencia*, 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 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. Si realmente se quiere, se puede verlo usando la función `print()`

[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).

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

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`.
- La sentencia `result.append(a)` llama a un método del objeto lista `result`. Un método es una función que "pertenece" a un objeto y se nombra `obj.methodname`, dónde `obj` es algún objeto (puede ser una expresión), y `methodname` es el nombre del método que está definido por el tipo del objeto. Distintos tipos definen distintos métodos. Métodos de diferentes tipos pueden tener el mismo nombre sin causar ambigüedad. (Es posible definir tipos de objetos propios, y métodos, usando clases, mira *Clases*). El método `append()` mostrado en el ejemplo está definido para objetos lista; añade un nuevo elemento al final de la lista. En este ejemplo es equivalente a `result = result + [a]`, pero más eficiente.

4.8 Más sobre definición de funciones

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

4.8.1 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:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok 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
```

4.8.2 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
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 `**name` 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 `*name` (descrito en la siguiente sección) que recibe una *tuple* conteniendo los argumentos posicionales además de la lista de parámetros formales. (`*name` debe ocurrir antes de `**name`). 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.

4.8.3 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 por como 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

En 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'
```

La tercer función `kwd_only_args` solo permite parámetros con palabras clave, indicado por un `*` en la definición de la función:

```
>>> 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.

4.8.4 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 (mira *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 de cantidad variables son 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 nombrado”, 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'
```

4.8.5 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 !
```

4.8.6 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
```

El ejemplo anterior muestra el uso de una expresión lambda para retornar una función. Otro uso es para pasar pequeñas funciones como argumentos

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

4.8.7 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.
```

4.8.8 Anotación de funciones

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

Las *anotaciones* se almacenan en el atributo `__annotations__` de la función como un diccionario y no tienen efecto en ninguna otra parte de la función. Las anotaciones de los parámetros se definen luego de dos puntos después del nombre del parámetro, seguido de una expresión que evalúa al valor de la anotación. Las anotaciones de retorno son definidas por el literal ->, seguidas de una expresión, entre la lista de parámetros y los dos puntos que marcan el final de la declaración `def`. El siguiente ejemplo tiene un argumento posicional, uno nombrado, y el valor de retorno anotado:

```
>>> 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'
```

4.9 Intermezzo: Estilo de codificación

Ahora que estás a punto de escribir piezas de Python más largas y complejas, es un buen momento para hablar sobre *estilo de codificación*. 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 codificación 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 codificación 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 (mirar *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.

CAPÍTULO 5

Estructuras de datos

Este capítulo describe algunas cosas que ya has aprendido en más detalle 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*`)`
Agrega un ítem al final de la lista. Equivale a `a[len(a):] = [x]`.

`list.`**`extend`**`(`*iterable*`)`
Extiende la lista agregándole todos los ítems del iterable. Equivale a `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*]`)`
Quita el ítem en la posición dada de la lista y lo retorna. Si no se especifica un índice, `a.pop()` quita y retorna el último elemento de la lista. (Los corchetes que encierran a *i* en la firma del método denotan que el parámetro es opcional, no que deberías escribir corchetes en esa posición. Verás esta notación con frecuencia en la Referencia de la Biblioteca de Python.)

`list.`**`clear`**`()`
Elimina todos los elementos de la lista. Equivalente a `del a[:]`.

`list.`**`index`**`(`*x*[, *start*[, *end*]]`)`
Retorna el índice basado en cero del primer elemento cuyo valor sea igual a *x*. Lanza una excepción `ValueError` si no existe tal elemento.

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 con respecto al argumento *start*.

`list.`**`count`**`(`*x*`)`
Retorna el número de veces que *x* aparece en la lista.

`list.`**`sort`**`(*, *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`**`()`
Retorna una copia superficial de la lista. Equivalente a `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 a 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 impreso un valor de retorno – retornan el valor por defecto `None`.[1] Esto es un principio de diseño para todas las estructuras de datos mutables en Python.

Otra cosa que puedes observar es que no todos los datos se pueden ordenar o comparar. Por ejemplo, `[None, 'hello', 10]` no se puede ordenar ya que los enteros no se pueden comparar con strings y *None* no se puede comparar con los otros tipos. También hay algunos tipos que no tienen una relación de orden definida. Por ejemplo, `3+4j < 5+7j` no es una comparación válida.

5.1.1 Usando 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:

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

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

5.1.2 Usando 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 por 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]
```

Nota 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)]
```

Notá 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, in <module>
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # 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 en 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 sección). 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 indizado y las operaciones de seccionado. 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 consiste de 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. Notá 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*, está también soportada la comprensión de conjuntos:

```
>>> 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 agrega, de inicio, pares clave:valor 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 no existente.

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 en orden ordenado.

```
>>> 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 (no ocurre) en una secuencia. Los operadores `is` y `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'
```

Nota que en Python, a diferencia de C, asignaciones dentro de expresiones deben realizarse explícitamente con walrus operator `:=`. 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 los strings 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 objetas 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 generará una excepción `TypeError`.

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 en el espacio de nombres actual; sólo añade el nombre del módulo `fibo`. 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, el cual es usado como espacio de nombres global para todas las funciones definidas en el módulo. Por lo tanto, el autor de un módulo puede usar variables globales en el módulo sin preocuparse acerca de conflictos con una variable global del usuario. Por otro lado, si sabes lo que estás haciendo puedes acceder a las variables globales de un módulo con la misma notación usada para referirte a sus funciones, `nombremodulo.nombreitem`.

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 son ubicados en el espacio de nombres global del módulo que hace la importación.

Hay una variante de la declaración `import` que importa nos 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.

Nota 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.

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

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

```
>>> 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 Ejecutando 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 propósitos de prueba (ejecutar el módulo como un script ejecuta el juego 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. 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 que incluye un directorio `site-packages`, manejado por el módulo `site`).

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 el cual 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']
```

Note 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 cada uno, el uso de nombres de módulo con puntos salva a los autores de paquetes con múltiples módulos, como NumPy o Pillow de preocupaciones por los nombres de los módulos de cada uno.

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
              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.

Los archivos `__init__.py` son obligatorios para que Python trate los directorios que contienen los archivos como paquetes. Esto evita que los directorios con un nombre común, como `string`, oculten involuntariamente módulos válidos que se producen luego en el camino de búsqueda del módulo. En el caso mas simple, `__init__.py` puede ser solo un archivo vacío, pero también puede ejecutar código de inicialización para el paquete o el conjunto de variables `__all__`, descriptas luego.

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)
```

Note 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 Importando * 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`.

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 `import` 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 `import` 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 que los imports relativos se basan en el nombre del módulo actual. Ya que el nombre del módulo principal es siempre `"__main__"`, los módulos pensados para usarse como módulo principal de una aplicación Python siempre deberían usar `import` absolutos.

6.4.3 Paquetes en múltiples directorios

Los paquetes soportan un atributo especial más, `__path__`. Este se inicializa a una lista que contiene el nombre del directorio donde está el archivo `__init__.py` del paquete, antes de que el código en ese archivo se ejecute. Esta variable puede modificarse, afectando búsquedas futuras de módulos y subpaquetes contenidos en el paquete.

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

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 *formatted string literals*, 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'
```

- El método `str.format()` requiere más esfuerzo manual. Se seguirá usando { y } para marcar dónde se sustituirá una variable y puede proporcionar directivas de formato detalladas, pero también se debe proporcionar la información de lo que se va a formatear.

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

- 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'))"
```

El módulo `string` contiene una clase `Template` que ofrece otra forma de sustituir valores en cadenas, utilizando marcadores de posición como *$x*` y reemplazarlos con valores desde un diccionario, pero esto ofrece mucho menos control en el formato.

7.1.1 Formatear cadenas literales

Formatted string literals (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'.
```

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 dict 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 la tabla como argumentos nombrados 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
```

Esto es particularmente útil en combinación con la función integrada `vars()`, que retorna un diccionario conteniendo todas las variables locales.

Como ejemplo, las siguientes lineas producen un conjunto de columnas alineadas ordenadamente 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()`, mirá en 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
```

(Resaltar 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 dejar sin usar un valor. (Si realmente deseas truncado 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

El operador % (módulo) también se puede utilizar para formatear cadenas de caracteres. Dados los `'cadena de caracteres' % valores`, las instancias de `%` en `cadena de caracteres` se reemplazan con cero o más elementos de `valores`. Esta operación se conoce comúnmente como interpolación de cadenas. Por ejemplo:

```
>>> 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: `open(nombre_de_archivo, modo)`.

```
>>> f = open('workfile', 'w')
```

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*, significa que lees y escribes caracteres desde y hacia el fichero, el cual se codifica con una codificación específica. Si no se especifica la codificación el valor por defecto depende de la plataforma (ver `open()`). `'b'` agregado al modo abre el fichero en *modo binario*: y los datos se leerán y escribirán en forma de objetos de bytes. Este modo debería usarse en todos los ficheros que no contienen texto.

Cuando se lee en modo texto, por defecto se convierten los fines 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 finales 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`. Asegurate de usar modo binario cuando leas y escribas tales archivos.

Es una buena práctica usar la declaración `with` cuando manejamos objetos archivo. Tiene la ventaja 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') as f:
...     read_data = f.read()

>>> # 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: Al llamar a `f.write()` sin usar la palabra clave :keyword::*!with* o llamar a `f.close()` **podría** dar como resultado los argumentos de `f.write()` no se escribe completamente en el disco, incluso si el programa se cierra 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 de (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, como máximo *size* caracteres (en modo texto) o *size* bytes (en modo binario) son leídos y retornados. 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, podés 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='')
...
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 grabar 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 grabar tipos de datos complicados, Python te permite usar formato intercambiable de datos popular 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 *serializing*. Reconstruir los datos desde la representación de cadena de caracteres es llamado *deserializing*. 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 usando 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 *archivo de texto* que fue abierto para lectura:

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

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 idiomas. 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
```

El intérprete reproduce la línea responsable del error y muestra una pequeña “flecha” que apunta al primer lugar donde se detectó el error. El error ha sido provocado (o al menos detectado) en

el elemento que *precede* a la flecha: en el ejemplo, el error se detecta en la función `print()`, ya que faltan dos puntos (`':'`) antes del mismo. Se muestran el nombre del archivo y el número de línea para que sepas dónde mirar en caso de que la entrada venga de un programa.

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>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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.
- Si ocurre una excepción que no coincide con la indicada en la *cláusula except* se pasa a los `try` más externos; si no se encuentra un gestor, se genera una *unhandled exception* (excepción no gestionada) y la ejecución se interrumpe con un mensaje como el que se muestra arriba.

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
```

Una clase en una cláusula `except` es compatible con una excepción si es de la misma clase o de una clase derivada de la misma (pero no de la otra manera — una *cláusula except* listando una clase derivada no es compatible con una clase base). Por ejemplo, el siguiente código imprimirá B, C y D, en ese orden:

```
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.

Todas las excepciones heredan de `BaseException`, por lo que se puede utilizar como comodín. ¡Use esto con extrema precaución, ya que es fácil enmascarar un error de programación real de esta manera! También se puede usar para imprimir un mensaje de error y luego volver a generar la excepción (permitiendo que la función que llama también maneje la excepción):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except BaseException as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

Alternativamente, la última cláusula except puede omitir el(los) nombre(s) de excepción, sin embargo, el valor de la excepción debe recuperarse de `sys.exc_info()[1]`.

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`.

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 vinculada a una instancia de excepción con los argumentos almacenados en `instance.args`. Por conveniencia, la instancia de excepción define `__str__()` para que los argumentos se puedan imprimir directamente sin tener que hacer referencia a `.args`. También se puede crear una instancia de una excepción antes de generarla y agregarle los atributos que desee.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     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
```

Si una excepción tiene argumentos, estos se imprimen como en la parte final (el "detalle") del mensaje para las excepciones no gestionadas ("*Unhandled exception*").

Los gestores de excepciones no solo gestionan las excepciones si ocurren inmediatamente en la *cláusula try*, sino también si 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>
NameError: HiThere
```

El único argumento de `raise` indica la excepción a generarse. Tiene que ser o una instancia de excepción, o una clase de excepción (una clase que hereda de `Exception`). Si se pasa una clase de excepción, la misma será instanciada 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>
NameError: HiThere
```

8.5 Encadenamiento de excepciones

La declaración `raise` permite una palabra clave opcional `from` que habilita el encadenamiento de excepciones. Por ejemplo:

```
# 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>
  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>
RuntimeError: Failed to open database
```

El encadenamiento de excepciones ocurre automáticamente cuando se lanza una excepción dentro de una sección `except` o `finally`. Esto se puede desactivar usando 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>
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. Al crear un módulo que puede lanzar varios errores distintos, una práctica común es crear una clase base para excepciones definidas en ese módulo y extenderla para crear clases excepciones específicas para distintas condiciones de error:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

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. Se puede encontrar más información sobre clases en el capítulo *Clases*.

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!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

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.
- Si la cláusula `finally` ejecuta una declaración `break`, `continue` o `return`, las excepciones no se vuelven a lanzar.
- 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.
- Si una cláusula `finally` incluye una sentencia `return`, el valor retornado será el de la cláusula `finally`, no la del de la sentencia `return` de la cláusula `try`.

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)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
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.

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 sólo lectura, o de escritura. En el último caso es posible la asignación a atributos. Los atributos de módulo pueden escribirse: `modulo.la_respuesta = 42`. Los atributos de escritura se pueden borrar también con la declaración `del`. Por ejemplo, `del modulo.la_respuesta` va a eliminar el atributo `la_respuesta` del objeto con nombre `modulo`.

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.

[1] Excepto por una cosa. Los objetos módulo tienen un atributo de sólo lectura secreto llamado `__dict__` que retorna el diccionario usado para implementar el espacio de nombres del módulo; el nombre `__dict__` es un atributo pero no un nombre global. Obviamente, usar esto viola la abstracción de la implementación del espacio de nombres, y debería ser restringido a cosas como depuradores post-mortem.

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 las funciones que encierran a la función actual, que 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 como global, entonces todas las referencias y asignaciones al mismo van directo al ámbito intermedio que contiene los nombres globales del módulo. Para reasignar nombres encontrados afuera del ámbito más interno, se puede usar la declaración `nonlocal`; si no se declara nonlocal, esas variables serán de sólo lectura (un intento de escribir a esas variables simplemente crea una *nueva* variable local en el ámbito interno, dejando intacta la variable externa del mismo nombre).

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 *algo* de *prueba_ambitos*. La asignación `nonlocal` cambió la vinculación de *algo* de *prueba_ambitos*, y la asignación `global` cambió la vinculación a nivel de módulo.

También podés ver que no había vinculación para *algo* 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 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 (`Clase` 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'
```

…entonces `MiClase.i` y `MiClase.f` son referencias de atributos válidas, que retornan un entero y un objeto función respectivamente. Los atributos de clase también pueden ser asignados, o sea que podés cambiar el valor de `MiClase.i` mediante asignación. `__doc__` también es un atributo válido, que retorna la documentación asociada a la clase: `"Simple clase de ejemplo"`.

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.

Los *atributos de datos* se corresponden con las «variables de instancia» en Smalltalk, y con las «variables miembro» en C++. Los atributos de datos no necesitan ser declarados; tal como las variables locales son creados la primera vez que se les asigna algo. Por ejemplo, si `x` es la instancia de `MiClase` creada más arriba, el siguiente pedazo de código va a imprimir el valor `16`, sin dejar ningún rastro:

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

El otro tipo de atributo de instancia es el *método*. Un método es una función que «pertenece a» un objeto. En Python, el término método no está limitado a instancias de clase: otros tipos de objetos pueden tener métodos también. Por ejemplo, los objetos lista tienen métodos llamados append, insert, remove, sort, y así sucesivamente. Pero, en la siguiente explicación, usaremos el término método para referirnos exclusivamente a métodos de objetos instancia de clase, a menos que se especifique explícitamente lo contrario.

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 `MiClase.f` es una función, pero `x.i` no lo es, dado que `MiClase.i` no lo es. Pero `x.f` no es la misma cosa que `MiClase.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()
```

En el ejemplo `MiClase`, esto retorna la cadena `'hola mundo'`. Pero no es necesario llamar al método justo en ese momento: `x.f` es un objeto método, y puede ser guardado y llamado más tarde. Por ejemplo:

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

…continuará imprimiendo `hola mundo` 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 levanta 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 `MiClase.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.

Si todavía no entiendes como funcionan los métodos, una mirada a su implementación quizás pueda aclarar dudas. Cuando un atributo sin datos de una instancia es referenciado, la clase de la instancia es accedida. Si el nombre indica un atributo de clase válido que sea un objeto función, se crea un objeto método empaquetando (apunta a) la instancia y al objeto función, juntados en un objeto abstracto: este es el objeto método. Cuando el objeto método es llamado con una lista de argumentos, se crea una nueva lista de argumentos a partir del objeto instancia y la lista de argumentos. Finalmente el objeto función es llamado con esta nueva lista de argumentos.

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 *trucos* en el siguiente código no debería ser usada como variable de clase porque una sola lista sería compartida por todos las instancias de *Perro*:

```
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)

>>> 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>
    .
    .
    .
    <statement-N>
```

El nombre `ClaseBase` 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: `ClaseDerivada()` 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 `ClaseBase.metodo(self, argumentos)`. En ocasiones esto es útil para los clientes también. (Observá que esto sólo funciona si la clase base es accesible como `ClaseBase` 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 clases padres como primero 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 `ClaseDerivada`, se busca en `Base1`, luego (recursivamente) en las clases base de `Base1`, y sólo 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.

El ordenamiento dinámico es necesario porque todos los casos de herencia múltiple exhiben una o más relaciones en diamante (cuando se puede llegar al menos a una de las clases base por distintos caminos desde la clase de más abajo). Por ejemplo, todas las clases heredan de `object`, por lo tanto cualquier caso de herencia múltiple provee más de un camino para llegar a `object`. Para que las clases base no sean accedidas más de una vez, el algoritmo dinámico hace lineal el orden de búsqueda de manera que se preserve el orden de izquierda a derecha especificado en cada clase, que se llame a cada clase base sólo una vez, y que sea monótona (lo cual significa que una clase puede tener clases derivadas sin afectar el orden de precedencia de sus clases bases). En conjunto, estas propiedades hacen posible diseñar clases confiables y extensibles con herencia múltiple. Para más detalles mirá https://www.python.org/download/releases/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.

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 Cambalache

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. Una definición de clase vacía funcionará perfecto:

```
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Algún código Python que espera un tipo abstracto de datos en particular puede frecuentemente recibir en cambio una clase que emula los métodos de aquel tipo de datos. Por ejemplo, si tenés una función que formatea algunos datos a partir de un objeto archivo, podés definir una clase con métodos `read()` y `readline()` que obtengan los datos de alguna cadena en memoria intermedia, y pasarlo como argumento.

Los objetos método de instancia tienen atributos también: `m.__self__` es el objeto instancia con el método `m()`, y `m.__func__` es el objeto función correspondiente al método.

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__()` levanta 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
<iterator object at 0x00A1DB50>
>>> 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 levantan `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']
```

CAPÍTULO 10

Pequeño paseo por la Biblioteca Estándar

10.1 Interfaz al sistema operativo

El módulo `os` provee docenas de funciones para interactuar con el sistema operativo:

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python310'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

Asegúrate de usar el estilo `import os` en lugar de `from os import *`. Esto evitará que `os.open()` oculte a la función integrada `open()`, que trabaja bastante 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 tareas diarias de administración de archivos y directorios, el módulo `shutil` provee una interfaz de más alto nivel 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` provee 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 linea de órdenes

Los programas frecuentemente necesitan procesar argumentos de linea de órdenes. Estos argumentos se almacenan en el atributo *argv* del módulo `sys` como una lista. Por ejemplo, la siguiente salida resulta de ejecutar `python demo.py uno dos tres` en la línea de órdenes:

```
>>> import sys
>>> print(sys.argv)
['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 por línea de comandos haciendo `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 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 en patrones de cadenas

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']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Cuando se necesita algo más sencillo solamente, se prefieren los métodos de las cadenas porque son más fáciles de leer y depurar:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Matemática

El módulo `math` permite el acceso a las funciones de la biblioteca C subyacente para la matemática de punto flotante:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

El módulo `random` provee herramientas para realizar selecciones al azar:

```
>>> 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
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 mandar correos:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
...     for line in response:
...         line = line.decode('utf-8')  # Decoding the binary data to text.
...         if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...             print(line)

<BR>Nov. 25, 09:43:32 PM EST
```

```
>>> 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()
```

(Notá 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, el foco de la implementación es en la extracción eficiente de partes para gestionarlas o formatear la salida. El módulo también soporta objetos que son conscientes de 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 formatos para archivar y comprimir datos se soportan directamente con los módulos: `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 saber el rendimiento relativo de las diferentes soluciones al mismo problema. Python provee una herramienta de medición que responde esas preguntas inmediatamente.

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 modesta 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 es escribir pruebas para cada función mientras se la 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` necesita más esfuerzo que el módulo `doctest`, pero permite que se mantenga en un archivo separado un conjunto más comprensivo de pruebas:

```
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 Las 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 módulo `xmlrpc.client` y `xmlrpc.server` convierten una implementación de la llamada a un procedimiento en una tarea casi trivial. A pesar de los nombres de los nombres, no se necesita ningún conocimiento o manejo de archivos 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` admite 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* para la biblioteca de bases de datos SQLite, proporcionando una base de datos persistente que se puede actualizar y acceder mediante 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."""
...
>>> 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("%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):
...     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 mientras 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:/python310/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.

El módulo `array` provee un objeto `array()` (vector) que es como una lista que almacena sólo datos homogéneos y de una manera más compacta. Los ejemplos a continuación muestran un vector de números guardados como dos números binarios sin signo de dos bytes (código de tipo `"H"`) en lugar de los 16 bytes por elemento habituales en listas de objetos int de Python:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

El módulo `collections` provee un objeto `deque()` que es como una lista más rápida para agregar y quitar elementos por el lado izquierdo pero con búsquedas más lentas por el medio. Estos objetos son adecuados para implementar colas y árboles de búsqueda a lo ancho:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> 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 Aritmética de punto flotante decimal

El módulo `decimal` provee un tipo de dato `Decimal` para soportar aritmética de punto flotante decimal. Comparado con `float`, la implementación de punto flotante binario incluida, la clase es muy útil especialmente para

- 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
>>> sum([0.1]*10) == 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

El script usado para crear y manejar entornos virtuales es `pyvenv`. `pyvenv` normalmente instalará la versión mas reciente de Python que tengas disponible; el script también es instalado con un número de versión, con lo que si tienes múltiples versiones de Python en tu sistema puedes seleccionar una versión de Python específica ejecutando `python3` o la versión que desees.

Para crear un entorno virtual, decide en que carpeta quieres crearlo y ejecuta el módulo `venv` como script con la ruta a la carpeta:

```
python3 -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 colisione 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.bat
```

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']
>>>
```

12.3 Manejando paquetes con pip

Puede instalar, actualizar y eliminar paquetes usando un programa llamado **`pip`**. De forma predeterminada, `pip` instalará paquetes del índice de paquetes de Python, <https://pypi.org>. 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 re-ejecuta 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 uninstall` seguido de uno o varios nombres de paquetes eliminará los paquetes del entorno virtual.

`pip show` mostrará información de un paquete en particular:

```
(tutorial-env) $ 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:
```

`pip list` mostrará todos los paquetes instalados en el entorno virtual:

```
(tutorial-env) $ pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`pip freeze` retorna una lista de paquetes instalados similar, pero el formato de salida es el requerido por `pip install`. Una convención común es poner esta lista en un archivo `requirements.txt`:

```
(tutorial-env) $ 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` entonces puede ser agregado a nuestro control 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
  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 distributing-index.

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 pero 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, obtener 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 principal sitio web de Python. Contiene código, documentación y referencias a páginas relacionadas con Python en la web. Este sitio web se refleja en varios lugares del mundo, como Europa, Japón y Australia; un espejo puede ser más rápido que el sitio principal, dependiendo de su ubicación geográfica.
- 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 de 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 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/.

[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.

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.