PYTHON

Nivel I

UNEWEB – Instituto de nuevas tecnologías.

Tabla de contenido

Introducción

Python es un lenguaje de programación poderoso y fácil de aprender. Cuenta con estructuras de datos eficientes y de alto nivel y un enfoque simple pero efectivo a la programación orientada a objetos. La elegante sintaxis de Python y su tipiado dinámico, junto con su naturaleza interpretada, hacen de éste un lenguaje ideal para scripting y desarrollo rápido de aplicaciones en diversas áreas y sobre la mayoría de las plataformas.

El intérprete de Python y la extensa biblioteca estándar están a libre disposición en forma binaria y de código fuente para las principales plataformas desde el sitio web de Python, https://www.python.org/, y puede distribuirse libremente. El mismo sitio contiene también distribuciones y enlaces de muchos módulos libres de Python de terceros, programas y herramientas, y documentación adicional.

El intérprete de Python puede extenderse fácilmente con nuevas funcionalidades y tipos de datos implementados en C o C++ (u otros lenguajes accesibles desde C). Python también puede usarse como un lenguaje de extensiones para aplicaciones personalizables.

Características

- Python es fácil de usar, pero es un lenguaje de programación de verdad, ofreciendo mucha más estructura y soporte para programas grandes de lo que pueden ofrecer los scripts de Unix o archivos por lotes. Por otro lado, Python ofrece mucho más chequeo de error que C, y siendo un *lenguaje de muy alto nivel*, tiene tipos de datos de alto nivel incorporados como arreglos de tamaño flexible y diccionarios. Debido a sus tipos de datos más generales, Python puede aplicarse a un dominio de problemas mayor que Awk o incluso Perl, y aún así muchas cosas siguen siendo al menos igual de fácil en Python que en esos lenguajes.

- Python te permite separar tu programa en módulos que pueden reusarse en otros programas en Python. Viene con una gran colección de módulos estándar que puedes usar como base de tus programas, o como ejemplos para empezar a aprender a programar en Python. Algunos de estos módulos proveen cosas como entrada/salida a

archivos, llamadas al sistema, sockets, e incluso interfaces a sistemas de interfaz gráfica de usuario 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 descartables, 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 por sangría 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 agregar una nueva función o módulo al intérprete, ya sea para realizar operaciones críticas a velocidad máxima, o para enlazar programas Python con bibliotecas que tal vez sólo estén disponibles en forma binaria (por ejemplo bibliotecas gráficas específicas de un fabricante). Una vez que estés realmente entusiasmado, es posible enlazar el intérprete Python en una aplicación hecha en C y usarlo como lenguaje de extensión o de comando para esa aplicación.

Usando el intérprete de Python

Invocando al intérprete

Por lo general, el intérprete de Python se instala en `/usr/local/bin/python3.6` en las máquinas dónde está disponible; poner `/usr/local/bin` en el camino de búsqueda de tu intérprete de comandos Unix hace posible iniciarlo ingresando la orden:

```
python3.6
```

...en la terminal. Ya que la elección del directorio dónde vivirá el intérprete es una opción del proceso de instalación, puede estar en otros lugares. (Por ejemplo, `/usr/local/python` es una alternativa popular).
En máquinas con Windows, la instalación de Python por lo general se encuentra en `C:\Python36`, aunque se puede cambiar durante la instalación. Para añadir este directorio al camino, podes ingresar la siguiente orden en el prompt de DOS:

```
set path=%path%;C:\python36
```

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) en el prompt primario. Si esto no funciona, se puede salir del intérprete ingresando: `quit()`.

Las características para editar líneas del intérprete incluyen edición interactiva, sustitución usando el historial y completado de código en sistemas que soportan readline. Tal vez la forma más rápida de detectar si las características de edición están presentes es ingresar `Control-P` en el primer prompt de Python que aparezca. Si se escucha un beep, las características están presentes. Si no pasa nada, o si aparece `^P`, estas características no están disponibles; solo se va a poder usar backspace para borrar los caracteres de la línea actual.

La forma de operar del intérprete es parecida a la línea de comandos de Unix: cuando se la llama con la entrada estándar conectada a una terminal, lee y ejecuta comandos en forma interactiva; cuando es llamada con un nombre de archivo como argumento o con un archivo como entrada estándar, lee y ejecuta un *script* del archivo.

Una segunda forma de iniciar el intérprete es `python -c comando [arg] ...`, que ejecuta las sentencias en *comando*, similar a la opción *-c* de la línea de comandos, debido a que las sentencias de Python suelen tener espacios en blanco u otros caracteres que son especiales en la línea de comandos, es normalmente recomendado citar el *comando* entre comillas dobles.

Algunos módulos de Python son también útiles como scripts. Pueden invocarse usando `python -m module [arg] ...`, que ejecuta el código de *module* como si se hubiese ingresado su nombre completo en la línea de comandos.

Cuando se usa un script, a veces es útil correr primero el script y luego entrar al modo interactivo. Esto se puede hacer pasándole la opción *-i* antes del nombre del script.

Pasaje de argumentos

Cuando son conocidos por el intérprete, el nombre del script y los argumentos adicionales son entonces convertidos a una lista de cadenas de texto asignada a la variable `argv` del módulo `sys`. Se puede acceder a esta lista haciendo `import sys`. El largo de esta lista es al menos uno; cuando ningún script o argumentos son pasados, `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 command*, `sys.argv[0]` vale `'-c'`. Cuando se usa *-m module*, `sys.argv[0]` toma el valor del nombre completo del módulo. Las opciones encontradas luego de *-c command* o *-m module* no son consumidas por el procesador de opciones de Python pero de todas formas almacenadas en `sys.argv` para ser manejadas por el comando o módulo.

Modo interactivo

Se dice que estamos usando el intérprete en modo interactivo, cuando los comandos son leídos desde una terminal. En este modo espera el siguiente comando con el *prompt primario*: usualmente tres signos mayor-que (`>>>`); para las líneas de continuación espera con el *prompt secundario*, por defecto tres puntos (`...`). Antes de mostrar el prompt primario, el intérprete muestra un mensaje de bienvenida reportando su número de versión y una nota de copyright:

```
$ python3.6
Python 3.6 (default, Sep 16 2015, 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 queremos ingresar un constructor multilínea. Como en el ejemplo la sentencia **if**:

```
>>> el_mundo_es_plano = True
>>> if el_mundo_es_plano:
...     print("¡Ten cuidado de no caerte!")
...
¡Ten cuidado de no caerte!
```

El intérprete y su entorno

Codificación del código fuente

Por default, los archivos fuente de Python son tratados como codificados en UTF-8. En esa codificación, los caracteres de la mayoría de los lenguajes del mundo pueden ser usados simultáneamente en literales, identificadores y comentarios a pesar que la biblioteca estándar usa solamente caracteres ASCII para los identificadores. Una convención que debería seguir cualquier código que sea portable. Para mostrar estos caracteres correctamente, tu editor debe reconocer que el archivo está en UTF-8 y usar una tipografía que soporte todos los caracteres del archivo.

También es posible especificar una codificación distinta para los archivos fuente. Para hacer esto, se debe poner una o más líneas de comentarios especiales luego de la línea del `#!` para definir la codificación del archivo fuente:

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

Con esa declaración, todo en el archivo fuente será tratado utilizando la codificación *encoding* en lugar de UTF-8.

Por ejemplo, si tu editor no soporta la codificación UTF-8 e insiste en usar alguna otra, digamos Windows-1252, se puede escribir:

```
# -*- coding: cp-1252 -*-
```

y usar todos los caracteres del conjunto de Windows-1252 en los archivos fuente. El comentario especial de la codificación debe estar en la *primera o segunda* línea del archivo.

Una introducción informal a Python

Usar Python como una calculadora

Vamos a probar algunos comandos simples en Python. Iniciar primero un intérprete y esperar a que el prompt primario, `>>>`. cargue. (No debería demorar tanto).

Números

El intérprete actúa como una simple calculadora; se puede ingresar 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  # la división siempre retorna un número de punto flotante
1.6
```

Los números enteros (por ejemplo `2`, `4`, `20`) son de tipo **`int`**, aquellos con una parte fraccional (por ejemplo `5.0`, `1.6`) son de tipo **`float`**. Vamos a ver más sobre tipos de números luego en este tutorial.

La división (`/`) siempre retorna un punto flotante. Para hacer *floor division* y obtener un resultado entero (descartando cualquier resultado fraccional) se puede usar el operador `//`; para calcular el resto se usa `%`:

```
>>> 17 / 3  # la división clásica retorna un punto flotante
5.666666666666667
>>>
>>> 17 // 3  # la división entera descarta la parte fraccional
5
>>> 17 % 3  # el operado % retorna el resto de la división
```

```
2
>>> 5 * 3 + 2  # resultado * divisor + resto
17
```

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

```
>>> 5 ** 2  # 5 al cuadrado
25
>>> 2 ** 7  # 2 a la potencia de 7
128
```

El signo igual (`=`) es usado para asignar un valor a una variable. Luego, ningún resultado es mostrado antes del próximo prompt:

```
>>> ancho = 20
>>> largo = 5 * 9
>>> ancho * largo
900
```

Si una variable no está “definida” (con un valor asignado), intentar usarla producirá un error:

```
>>> n  # tratamos de acceder a una variable no definida
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

En el modo interactivo, la última expresión impresa es asignada a la variable `_`. Esto significa que cuando estés usando Python como una calculadora de escritorio, es más fácil seguir calculando, por ejemplo:

```
>>> impuesto = 12.5 / 100
>>> precio = 100.50
```

```
>>> precio * impuesto
12.5625
>>> precio + _
113.0625
>>> round(_, 2)
113.06
```

Esta variable debería 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 soporta otros tipos de números, como ser **`Decimal`** y **`Fraction`**. Python también tiene soporte integrado para *números complejos*, y usa el sufijo `j` o `J` para indicar la parte imaginaria (por ejemplo `3+5j`).

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. `\` puede ser usado para escapar comillas:

```
>>> 'huevos y pan'  # comillas simples
'huevos y pan'
>>> 'doesn\'t'  # usa \' para escapar comillas simples...
"doesn't"
>>> "doesn't"  # ...o de lo contrario usa comillas doblas
"doesn't"
>>> '"Si," le dijo.'
'"Si," le dijo.'
>>> "\"Si,\" le dijo."
'"Si," le dijo.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

En el intérprete interactivo, la salida de cadenas está encerrada en comillas y los caracteres especiales son escapados con barras invertidas. Aunque esto a veces luzca diferente de la

entrada (las comillas que encierran pueden cambiar), las dos cadenas son equivalentes. La cadena se encierra 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," she said.'
'"Isn\'t," she said.'
>>> print('"Isn\'t," she said.')
"Isn't," she said.
>>> s = 'Primera línea.\nSegunda línea.'  # \n significa nueva línea
>>> s  # sin print(), \n es incluído en la salida
'Primera línea.\nSegunda línea.'
>>> print(s)  # con print(), \n produce una nueva línea
Primera línea.
Segunda línea.
```

Si no se quiere que los caracteres antepuestos por `\` sean interpretados como caracteres especiales, se puede usar *cadenas crudas* agregando una `r` antes de la primera comilla:

```
>>> print('C:\algun\nombre')  # aquí \n significa nueva línea!
C:\algun
ombre
>>> print(r'C:\algun\nombre')  # nota la r antes de la comilla
C:\algun\nombre
```

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

```
print("""\
Uso: algo [OPTIONS]
     -h                        Muestra el mensaje de uso
     -H nombrehost             Nombre del host al cual conectarse
```

```
""")
```

produce la siguiente salida: (nota que la línea inicial no está incluida)

```
Uso: algo [OPTIONS]
     -h                        Muestra el mensaje de uso
     -H nombrehost             Nombre del host al cual conectarse
```

Las cadenas de texto pueden ser concatenadas (pegadas juntas) con el operador `+` y repetidas con `*`:

```
>>> # 3 veces 'un', seguido de 'ium'
>>> 3 * 'un' + 'ium'
'ununununium'
```

Dos o más *cadenas literales* (aquellas encerradas entre comillas) una al lado de la otra son automáticamente concatenadas:

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

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

```
>>> prefix = 'Py'
>>> prefix 'thon'  # no se puede concatenar una variable y una cadena literal
  ...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  ...
SyntaxError: invalid syntax
```

Si se desea concatenar variables o una variable con un literal, usar `+`:

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

Esta característica es particularmente útil cuando se desea separar cadenadas largas:

```
>>> texto = ('Pon muchas cadenas dentro de paréntesis '
...          'para que ellas sean unidas juntas.')
>>> texto
'Pon muchas cadenas dentro de paréntesis para que ellas sean unidas juntas.'
```

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

```
>>> palabra = 'Python'
>>> palabra[0]  # caracter en la posición 0
'P'
>>> palabra[5]  # caracter en la posición 5
'n'
```

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

```
>>> palabra[-1]  # último caracter
'n'
>>> palabra[-2]  # ante último caracter
'o'
>>> palabra[-6]
'P'
```

Nota que -0 es lo mismo que 0, los índices negativos comienzan desde -1.
Además de los índices, las *rebanadas* también están soportadas. Mientras que los índices son usados para obtener caracteres individuales, las *rebanadas* te permiten obtener sub-cadenas:

```
>>> palabra[0:2]  # caracteres desde la posición 0 (incluida) hasta la 2 (excluida)
'Py'
>>> palabra[2:5]  # caracteres desde la posición 2 (incluida) hasta la 5 (excluida)
'tho'
```

Nota como el primero es siempre incluido, y que el último es siempre excluido. Esto asegura que `s[:i] + s[i:]` siempre sea igual a `s`:

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

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.

```
>>> palabra[:2]  # caracteres desde el principio hasta la posición 2 (excluida)
'Py'
>>> palabra[4:]  # caracteres desde la posición 4 (incluida) hasta el final
'on'
>>> palabra[-2:] # caracteres desde la ante-última (incluida) hasta el final
'on'
```

Una forma de recordar cómo funcionan las rebanadas es pensar en los índices como puntos *entre* caracteres, con el punto a la izquierda del primer carácter numerado en 0. Luego, el punto a la derecha del último carácter de una cadena de *n* caracteres tiene índice *n*, por ejemplo:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
```

```
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

La primer fila de números da la posición de los índices 0...6 en la cadena; la segunda fila da los correspondientes índices negativos. La rebanada de *i* a *j* consiste en todos los caracteres entre los puntos etiquetados *i* y *j*, respectivamente.

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

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

```
>>> palabra[42]  # la palabra solo tiene 6 caracteres
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Sin embargo, índices fuera de rango en rebanadas son manejados satisfactoriamente:

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

Las cadenas de Python no pueden ser modificadas – son *inmutables*. Por eso, asignar a una posición indexada de la cadena resulta en un error:

```
>>> palabra[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> palabra[2:] = 'py'
  ...
```

```
TypeError: 'str' object does not support item assignment
```

Si necesitas una cadena diferente, deberías crear una nueva:

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

La función incorporada **`len()`** devuelve la longitud de una cadena de texto:

```
>>> s = 'supercalifrastilisticoespialidoso'
>>> len(s)
33
```

Listas

Python tiene varios tipos de datos *compuestos*, usados 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:

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

Como las cadenas de caracteres (y todos los otros tipos *sequence* integrados), las listas pueden ser indexadas y rebanadas:

```
>>> cuadrados[0]  # índices retornan un ítem
1
>>> cuadrados[-1]
25
```

```
>>> cuadrados[-3:]  # rebanadas retornan una nueva lista
[9, 16, 25]
```

Todas las operaciones de rebanado devuelven una nueva lista conteniendo los elementos pedidos. Esto significa que la siguiente rebanada devuelve una copia superficial de la lista:

```
>>> cuadrados[:]
[1, 4, 9, 16, 25]
```

Las listas también soportan operaciones como concatenación:

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

A diferencia de las cadenas de texto, que son *inmutables*, las listas son un tipo *mutable*, es posible cambiar un su contenido:

```
>>> cubos = [1, 8, 27, 65, 125]  # hay algo mal aquí
>>> 4 ** 3  # el cubo de 4 es 64, no 65!
64
>>> cubos[3] = 64  # reemplazar el valor incorrecto
>>> cubos
[1, 8, 27, 64, 125]
```

También se puede agregar nuevos ítems al final de la lista, usando el *método* **append():**

```
>>> cubos.append(216)  # agregar el cubo de 6
>>> cubos.append(7 ** 3)  # y el cubo de 7
>>> cubos
[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:

```
>>> letras = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letras
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # reemplazar algunos valores
>>> letras[2:5] = ['C', 'D', 'E']
>>> letras
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # ahora borrarlas
>>> letras[2:5] = []
>>> letras
['a', 'b', 'f', 'g']
>>> # borrar la lista reemplazando todos los elementos por una lista vacía
>>> letras[:] = []
>>> letras
[]
```

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

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

Primeros pasos hacia la programación

Por supuesto, podemos usar Python para tareas más complicadas que sumar dos y dos. Por ejemplo, podemos escribir una sub-secuencia inicial de la serie de *Fibonacci* así:

```
>>> # Series de Fibonacci:
... # la suma de dos elementos define el siguiente
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Este ejemplo introduce varias características nuevas.

- La primera línea contiene una *asignación múltiple*: las variables `a` y `b` toman en forma simultánea los nuevos valores 0 y 1. En la última línea esto se vuelve a usar, demostrando que las expresiones a la derecha son evaluadas antes de que suceda cualquier asignación. Las expresiones a la derecha son evaluadas de izquierda a derecha.
- El bucle **while** se ejecuta mientras la condición (aquí: `b < 10`) sea verdadera. En Python, como en C, cualquier entero distinto de 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 longitud distinta de cero es verdadero, las secuencias vacías son falsas. La prueba usada en el ejemplo es una comparación simple. Los operadores estándar de comparación se escriben igual que en C: `<` (menor

qué), `>` (mayor qué),`==` (igual a), `<=` (menor o igual qué), `>=` (mayor o igual qué) y `!=` (distinto a).

- El *cuerpo* del bucle está *sangrado*: la sangría es la forma que usa Python para agrupar declaraciones. En el intérprete interactivo debes teclear un tab o espacio(s) para cada línea sangrada. En la práctica vas a preparar entradas más complicadas para Python con un editor de texto; todos los editores de texto decentes tienen la facilidad de agregar la sangría automáticamente. Al ingresar una declaración compuesta en forma interactiva, debes 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 el o los argumentos que se le pasan. 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 en punto flotante, y cadenas. Las cadenas de texto son impresas sin comillas, y un espacio en blanco es insertado entre los elementos, así podrás formatear cosas de una forma agradable:

```
>>> i = 256*256
>>> print('El valor de i es', i)
El valor de i es 65536
```

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

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

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.

La sentencia if

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

```
>>> x = int(input("Ingresa un entero, por favor: "))
Ingresa un entero, por favor: 42
>>> if x < 0:
...      x = 0
...      print('Negativo cambiado a cero')
... elif x == 0:
...      print('Cero')
... elif x == 1:
...      print('Simple')
... else:
...      print('Más')
...
'Mas'
```

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.

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:

```
>>> # Midiendo cadenas de texto
... palabras = ['gato', 'ventana', 'defenestrado']
>>> for p in palabras:
...     print(p, len(p))
...
gato 4
ventana 7
defenestrado 12
```

Si necesitas modificar la secuencia sobre la que estás iterando mientras estás adentro del ciclo (por ejemplo para borrar algunos ítems), se recomienda que hagas primero una copia. Iterar sobre una secuencia no hace implícitamente una copia. La notación de rebanada es especialmente conveniente para esto:

```
>>> for p in palabras[:]:  # hace una copia por rebanada de toda la lista
...     if len(p) > 6:
...         palabras.insert(0, p)
...
>>> palabras
['defenestrado', 'ventana', 'gato', 'ventana', 'defenestrado']
```

Con `for w in words:`, el ejemplo intentaría crear una lista infinita, insertando `defenestrado` una y otra vez.

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'):

```
range(5, 10)
   5 through 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

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

```
>>> a = ['Mary', 'tenia', 'un', 'corderito']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 tenia
2 un
3 corderito
```

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

Algo extraño sucede si mostrás un `range`:

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

De muchas maneras el objeto devuelto por **`range()`** se comporta como si fuera una lista, pero no lo es. Es un objeto que devuelve 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 lo 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 un *iterador* en ese sentido. La función **`list()`** es otra; crea listas a partir de iterables:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Más tarde veremos más funciones que devuelven iterables y que toman iterables como entrada.

Las sentencias break, continue, y else en lazos

La sentencia **break**, como en C, termina el lazo **for** o **while** más anidado.

Las sentencias de lazo pueden tener una cláusula `else` que es ejecutada cuando el lazo termina, luego de agotar la lista (con **for**) o cuando la condición se hace falsa (con **while**), pero no cuando el lazo es terminado con la sentencia **break**. Se ejemplifica en el siguiente lazo, que busca números primos:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'es igual a', x, '*', n/x)
...             break
...     else:
...         # sigue el bucle sin encontrar un factor
```

```
...         print(n, 'es un numero primo')
...
2 es un numero primo
3 es un numero primo
4 es igual a 2 * 2
5 es un numero primo
6 es igual a 2 * 3
7 es un numero primo
8 es igual a 2 * 4
9 es igual a 3 * 3
```

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

Cuando se usa con un ciclo, el `else` tiene más en común con el `else` de una declaración **try** que con el de un **if**: el `else` de un **try** se ejecuta cuando no se genera ninguna excepción, y el `else` de un ciclo se ejecuta cuando no hay ningún `break`.

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("Encontré un número par", num)
...         continue
...     print("Encontré un número", num)
Encontré un número par 2
Encontré un número 3
Encontré un número par 4
Encontré un número 5
Encontré un número par 6
Encontré un número 7
Encontré un número par 8
Encontré un número 9
```

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  # Espera ocupada hasta una interrupción de teclado (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   # Acordate de implementar esto!
...
```

Definiendo funciones

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

```
>>> def fib(n):  # escribe la serie de Fibonacci hasta n
...     """Escribe la serie de Fibonacci hasta n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
```

```
...
>>> # Ahora llamamos a la funcion que acabamos de definir:
... 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 primer 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*.

Hay herramientas que usan las docstrings para producir automáticamente documentación en línea o imprimible, o para permitirle al usuario que navegue el código en forma interactiva; es una buena práctica incluir docstrings en el código que uno escribe, por lo que se debe hacer un hábito de esto.

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, no se les puede asignar directamente un valor a las variables globales dentro de una función (a menos se las nombre en la sentencia **global**), aunque si pueden ser referenciadas.

Los parámetros reales (argumentos) de una función se introducen en la tabla de símbolos local de la función llamada cuando esta es ejecutada; así, los argumentos son pasados *por valor* (dónde el *valor* es siempre una *referencia* a un objeto, no el valor del objeto). [1] Cuando una función llama a otra función, una nueva tabla de símbolos local es creada para esa llamada.

La definición de una función introduce el nombre de la función en la tabla de símbolos actual. El valor del nombre de la función tiene un tipo que es reconocido por el intérprete como una función definida por el usuario. Este valor puede ser asignado a otro nombre que luego puede ser usado como una función. Esto sirve como un mecanismo general para renombrar:

```
>>> 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 devuelve un valor. De hecho, técnicamente hablando, los procedimientos 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()`**:

```
>>> 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): # devuelve la serie de Fibonacci hasta n
...     """Devuelve una lista conteniendo la serie de Fibonacci hasta n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # ver abajo
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # llamarla
>>> f100                # escribir el resultado
[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** devuelve 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. 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.

Más sobre definición de funciones

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

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 pedir_confirmacion(prompt, reintentos=4, recordatorio='Por favor, intente nu
evamente!'):
    while True:
        ok = input(prompt)
        if ok in ('s', 'S', 'si', 'Si', 'SI'):
            return True
        if ok in ('n', 'no', 'No', 'NO'):
            return False
        reintentos = reintentos - 1
        if reintentos < 0:
            raise ValueError('respuesta de usuario inválida')
        print(recordatorio)
```

Esta función puede ser llamada de distintas maneras:

- Pasando sólo el argumento obligatorio: `pedir_confirmacion('¿Realmente quieres salir?').`
- pasando uno de los argumentos opcionales: `pedir_confirmacion('¿Sobreescribir archivo?', 2).`
- o pasando todos los argumentos: `pedir_confirmacion('¿Sobreescribir archivo?', 2, "Vamos, solo si o 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
```

Palabras claves como argumentos

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

```
def loro(tension, estado='muerto', accion='explotar', tipo='Azul Nordico'):
    print("-- Este loro no va a", accion, end=' ')
    print("si le aplicás", tension, "voltios.")
    print("-- Gran plumaje tiene el", tipo)
    print("-- Está", estado, "!")
```

...acepta un argumento obligatorio (`tension`) y tres argumentos opcionales (`estado`, `accion`, y `tipo`). Esta función puede llamarse de cualquiera de las siguientes maneras:

```
loro(1000)                                  # 1 argumento posicional
loro(tension=1000)                          # 1 argumento nombrado
loro(tension=1000000, accion='VOOOOOM')     # 2 argumentos nombrados
loro(accion='VOOOOOM', tension=1000000)     # 2 argumentos nombrados
```

```
loro('un millón', 'despojado de vida', 'saltar')    # 3 args posicionales
loro('mil', estado='viendo crecer las flores desde abajo')  # uno y uno
```

...pero estas otras llamadas serían todas inválidas:

```
loro()                      # falta argumento obligatorio
loro(tension=5.0, 'muerto') # argumento posicional luego de uno nombrado
loro(110, tension=220)      # valor duplicado para el mismo argumento
loro(actor='Juan Garau')    # nombre del argumento desconocido
```

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 `loro`), y el orden de los mismos no es importante. Esto también se aplica a los argumentos obligatorios (por ejemplo, `loro(tension=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 funcion(a):
...     pass
...
>>> funcion(0, a=0)
Traceback (most recent call last):
...
TypeError: funcion() got multiple values for keyword argument 'a'
```

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

```
def ventadequeso(tipo, *argumentos, **palabrasclaves):
```

```
    print("-- ¿Tiene", tipo, "?")
    print("-- Lo siento, nos quedamos sin", tipo)
    for arg in argumentos:
        print(arg)
    print("-" * 40)
    claves = sorted(palabrasclaves.keys())
    for c in claves:
        print(c, ":", palabrasclaves[c])
```

Puede ser llamada así:

```
ventadequeso("Limburger", "Es muy liquido, sr.",
             "Realmente es muy muy liquido, sr.",
             cliente="Juan Garau",
             vendedor="Miguel Paez",
             puesto="Venta de Queso Merideño")
```

...y por supuesto imprimirá:

```
.. code-block:: none
```

Se debe notar que la lista de nombres de argumentos nombrados se crea al ordenar el resultado del método `keys()` del diccionario antes de imprimir su contenido; si esto no se hace, el orden en que los argumentos son impresos no está definido.

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. Antes del número variable de argumentos, cero o más argumentos normales pueden estar presentes:

```
def muchos_items(archivo, separador, *args):
    archivo.write(separador.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 nombrados y no posicionales:

```
>>> def concatenar(*args, sep="/"):
...     return sep.join(args)
...
>>> concatenar("tierra", "marte", "venus")
'tierra/marte/venus'
>>> concatenar("tierra", "marte", "venus", sep=".")
'tierra.marte.venus'
```

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 argumentos *inicio* y *fin*. Si no están disponibles en forma separada, se puede escribir la llamada a la función con el operador para desempaquetar argumentos de una lista o una tupla `*`::

```
>>> list(range(3, 6))   # llamada normal con argumentos separados
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))  # llamada con argumentos desempaquetados de la lista
[3, 4, 5]
```

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

```
>>> def loro(tension, estado='rostizado', accion='explotar'):
...     print("-- Este loro no va a", accion, end=' ')
...     print("si le aplicás", tension, "voltios.", end=' ')
```

```
...     print("Está", estado, "!")
...
>>> d = {"tension": "cinco mil", "estado": "demacrado",
...      "accion": "VOLAR"}
>>> loro(**d)
-- Este loro no va a VOLAR si le aplicás cinco mil voltios. Está demacrado !
```

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

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 primer línea que no está en blanco *siguiente* a la primera línea de la cadena determina la cantidad de sangría para toda la cadena de documentación. (No podemos usar la primera 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 debería 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 tabs (a 8 espacios, normalmente).

Este es un ejemplo de un docstring multi-línea:

```
>>> def mi_funcion():
...     """No hace mas que documentar la funcion.
...
...     No, de verdad. No hace nada.
...     """
...     pass
...
>>> print(mi_funcion.__doc__)
No hace mas que documentar la funcion.

No, de verdad. No hace nada.
```

Anotación de funciones

Las anotaciones de funciones son información completamente opcional sobre los tipos usadas en funciones definidas por el usuario.

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(jamon: str, huevos: str = 'huevos') -> str:
...     print("Anotaciones:", f.__annotations__)
...     print("Argumentos:", jamon, huevos)
...     return jamon + ' y ' + huevos
...
>>> f('carne')
Anotaciones: {'jamon': <class 'str'>, 'huevos': <class 'str'>, 'return': <class
'str'>}
Argumentos: carne huevos
'carne y huevos'
>>>
```

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 tabs. Cuatro espacios son un buen compromiso entre una sangría pequeña (permite mayor nivel de sangrado)y una sangría grande (más fácil de leer). Los tabs 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 usá `self` como el nombre para el primer argumento en los 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.

Estructuras de datos

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(L)` Extiende la lista agregándole todos los ítems de la lista dada. Equivale a `a[len(a):] = L`.

`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*. Es un error si no existe tal ítem.

`list.pop([i])` Quita el ítem en la posición dada de la lista, y lo devuelve. Si no se especifica un índice, `a.pop()` quita y devuelve el último ítem 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()` Quita todos los elementos de la lista. Equivalente a `del a[:]`.

`list.index(x[, start[, end]])` Devuelve un índice basado en cero en la lista del primer ítem cuyo valor sea *x*. Levanta una excepción **ValueError** si no existe tal ítem. Los argumentos opcionales *start* y *end* son interpetados como la notación de rebanadas y se usan para limitar la búsqueda a una subsecuencia particular de *x*. El index retornado se calcula de manera relativa al inicio de la secuencia completa en lugar de con respecto al argumento *start*.

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

`list.sort(key=None, reverse=False)` Ordena los ítems de la lista in situ (los argumentos pueden ser usados para personalizar el orden de la lista, ve **sorted()** para su explicación).

`list.reverse()` Invierte los elementos de la lista in situ.

`list.copy()` Devuelve una copia superficial de la lista. Equivalente a `a[:]`. Un ejemplo que usa la mayoría de los métodos de lista:

```
>>> frutas = ['naranja', 'manzana', 'pera', 'banana', 'kiwi', 'manzana', 'banana
']
>>> frutas.count('manzana')
2
>>> frutas.count('mandarina')
0
>>> frutas.index('banana')
3
>>> frutas.index('banana', 4)  # Find next banana starting a position 4
6
>>> frutas.reverse()
```

```
>>> frutas
['banana', 'manzana', 'kiwi', 'banana', 'pera', 'manzana', 'naranja']
>>> frutas.append('uva')
>>> frutas
    ['banana', 'manzana', 'kiwi', 'banana', 'pera', 'manzana', 'naranja', 'uva']
>>> frutas.sort()
>>> frutas
['manzana', 'manzana', 'banana', 'banana', 'uva', 'kiwi', 'naranja', 'pera']
>>> frutas.pop()
'pera'
```

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

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 ítem a la cima de la pila, use **append()**. Para retirar un ítem de la cima de la pila, use **pop()** sin un índice explícito. Por ejemplo:

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

```
[3, 4]
```

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, usa **collections.deque** el cual fue diseñado para agregar y sacar de ambas puntas de forma rápida. Por ejemplo:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # llega Terry
>>> queue.append("Graham")          # llega Graham
>>> queue.popleft()                 # el primero en llegar ahora se va
'Eric'
>>> queue.popleft()                 # el segundo en llegar ahora se va
'John'
>>> queue                           # el resto de la cola en órden de llegada
['Michael', 'Terry', 'Graham']
```

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 una subsecuencia de esos elementos para satisfacer una condición determinada.

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

```
>>> cuadrados = []
>>> for x in range(10):
```

```
...     cuadrados.append(x**2)
...
>>> cuadrados
[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:

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

o, un equivalente:

```
cuadrados = [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)]
```

Nota 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]
>>> # crear una nueva lista con los valores duplicados
>>> [x * 2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filtrar la lista para excluir números negativos
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # aplica una función a todos los elementos
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # llama un método a cada elemento
>>> frutafresca = ['  banana', '  mora de Logan ', 'maracuya  ']
>>> [arma.strip() for arma in frutafresca]
['banana', 'mora de Logan', 'maracuya']
>>> # crea una lista de tuplas de dos como (número, cuadrado)
>>> [(x, x ** 2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # la tupla debe estar entre paréntesis, sino es un error
>>> [x, x ** 2 for x in range(6)]
Traceback (most recent call last):
...
    [x, x ** 2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # aplanar una lista usando comprensión de listas con dos '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']
```

Listas por comprensión anidadas

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

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

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

La siguiente comprensión de lista transpondrá las filas y columnas:

```
>>> [[fila[i] for fila in matriz] 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:

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

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

```
>>> transpuesta = []
>>> for i in range(4):
...     # las siguientes 3 lineas hacen la comprensión de listas anidada
...     fila_transpuesta = []
...     for fila in matriz:
...         fila_transpuesta.append(fila[i])
...     transpuesta.append(fila_transpuesta)
...
>>> transpuesta
[[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(*matriz))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

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 devuelve 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.

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*. 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, 'hola!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hola!')
>>> # Las tuplas pueden anidarse:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hola!'), (1, 2, 3, 4, 5))
>>> # Las tuplas son inmutables:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # pero pueden contener objetos mutables:
... 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 *inmutables* 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 *mutables*, 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:

```
>>> vacia = ()
>>> singleton = 'hola',    # <-- notar la coma al final
>>> len(vacia)
0
>>> len(singleton)
1
>>> singleton
('hola',)
```

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.

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. Nota 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:

```
>>> canasta = {'manzana', 'naranja', 'manzana', 'pera', 'naranja', 'banana'}
>>> print fruta                      # muestra que se removieron los duplicados
{'pera', 'manzana', 'banana', 'naranja'}
>>> 'naranja' in canasta             # verificación de pertenencia rápida
True
>>> 'yerba' in canasta
False

>>> # veamos las operaciones para las letras únicas de dos palabras
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                # letras únicas en a
{a', 'r', 'b', 'c', 'd'}
>>> a - b                            # letras en a pero no en b
{'r', 'b', 'd'}
>>> a | b                            # letras en a o en b
{'a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'}
>>> a & b                            # letras en a y en b
```

```
{'a', 'c'}
>>> a ^ b                           # letras en a o b pero no en ambos
{'b', 'd', 'm', 'l', 'r', 'z'}
```

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'}
```

Diccionarios

Otro tipo de dato útil incluido en Python es el *diccionario*. 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 puedes usar listas como claves, ya que las listas pueden modificarse usando asignación por índice, asignación por sección, o métodos como **append()** y **extend()**.

Lo mejor es pensar en un diccionario como un conjunto no ordenado de pares *clave: valor*, con el requerimiento de que las claves sean únicas (dentro de un diccionario en particular). Un par de llaves crean un diccionario vacío: `{}`. Colocar una lista de pares clave:valor separados por comas entre las llaves añade pares clave:valor iniciales al diccionario; esta también es la forma en que los diccionarios se presentan 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 usas 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.

Hacer `list(d.keys())` en un diccionario devuelve una lista de todas las claves usadas en el diccionario, en un orden arbitrario (si las quieres ordenadas, usa en cambio `sorted(d.keys())`. Para controlar si una clave está en el diccionario, usa el **in**.

Un pequeño ejemplo de uso de un diccionario:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'irv': 4127, 'guido': 4127}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['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, 'jack': 4098, 'guido': 4127}
```

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, 'jack': 4098, 'guido': 4127}
```

Técnicas de iteración

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

```
>>> caballeros = {'gallahad': 'el puro', 'robin': 'el valiente'}
>>> for k, v in caballeros.items():
...     print(k, v)
...
gallahad el puro
robin el valiente
```

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(['ta', 'te', 'ti']):
...     print(i, v)
...
0 ta
1 te
2 ti
```

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

```
>>> preguntas = ['nombre', 'objetivo', 'color favorito']
>>> respuestas = ['lancelot', 'el santo grial', 'azul']
>>> for p, r in zip(preguntas, respuestas):
...     print('Cual es tu {0}?  {1}.'.format(p, r))
...
```

```
Cual es tu nombre?  lancelot.
Cual es tu objetivo?  el santo grial.
Cual es tu color favorito?  azul.
```

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 devuelve una nueva lista ordenada dejando a la original intacta.

```
>>> canasta = ['manzana', 'naranja', 'manzana', 'pera', 'naranja', 'banana']
>>> for f in sorted(set(canasta)):
...     print(f)
...
banana
manzana
naranja
pera
```

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

```
>>> datos = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> datos_filtrados = []
>>> for valor in datos:
...     if not math.isnan(valor):
...         datos_filtrados.append(valor)
```

```
...
>>> datos_filtrados
[56.2, 51.7, 55.3, 52.5, 47.8]
```

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 está (o no está) en una secuencia. Los operadores `is` e `is not` comparan si dos objetos son realmente el mismo objeto; esto es significativo sólo para objetos mutables como las listas. Todos los operadores de comparación tienen la misma prioridad, la cual 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 `(Aand (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 devuelto 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,

```
>>> cadena1, cadena2, cadena3 = '', 'Trondheim', 'Paso Hammer'
>>> non_nulo = cadena1 or cadena2 or cadena3
>>> non_nulo
'Trondheim'
```

Nota que en Python, a diferencia de C, la asignación no puede ocurrir dentro de expresiones. Los programadores de C pueden renegar por esto, pero es algo que evita un tipo de problema común encontrado en programas en C: escribir `=` en una expresión cuando lo que se quiere escribir es `==`.

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 una subsecuencia inicial de la otra, la secuencia más corta es la menor. El orden lexicográfico para cadenas de caracteres utiliza el orden de códigos Unicode para caracteres individuales. Algunos ejemplos de comparaciones 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)
```

Observa 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**.

Módulos

Si salís del intérprete de Python y entras de nuevo, las definiciones que hiciste (funciones y variables) se pierden. Por lo tanto, si quieres escribir un programa más o menos largo, es mejor que uses un editor de texto para preparar la entrada para el intérprete y ejecutarlo con ese archivo como entrada. Esto es conocido como crear un *guión*, o *script*. Si tu programa se vuelve más largo, quizás quieras separarlo en distintos archivos para un mantenimiento más fácil. Quizás también quieras usar una función útil que escribiste desde distintos programas sin copiar su definición a cada programa.

Para soportar esto, Python tiene una manera de poner definiciones en un archivo y usarlos en un script o en una instancia interactiva del intérprete. Tal archivo es llamado *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 archivo conteniendo definiciones y declaraciones de Python. El nombre del archivo es el nombre del módulo con el sufijo `.py` agregado. Dentro de un módulo, el nombre del mismo (como una cadena) está disponible en el valor de la variable global `__name__`. Por ejemplo, usa tu editor de textos favorito para crear un archivo llamado `fibo.py`en el directorio actual, con el siguiente contenido:

```
# módulo de números Fibonacci

def fib(n):    # escribe la serie Fibonacci hasta n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
     print()

def fib2(n):   # devuelve la serie Fibonacci hasta n
```

```
    resultado = []
    a, b = 0, 1
    while b < n:
        resultado.append(b)
        a, b = b, a+b
    return resultado
```

Ahora entra al intérprete de Python e importa este módulo con la siguiente orden:

```
>>> import fibo
```

Esto no mete los nombres de las funciones definidas en `fibo` directamente en el espacio de nombres actual; sólo mete ahí el nombre del módulo, `fibo`. Usando el nombre del módulo puedes acceder a las funciones:

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

Si se piensas usar la función frecuentemente, se puede asignar a un nombre local:

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

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 solamente la *primera* vez que el módulo se encuentra en una sentencia import. (Son también ejecutados si el archivo es ejecutado como un script).

Cada módulo tiene su propio espacio de nombres, el que es usado como espacio de nombres global por 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 tocar 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 el ubicar todas las declaraciones **import** al principio del módulo (o script, para el caso). Los nombres de los módulos importados se ubican en el espacio de nombres global del módulo que hace la importación.

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

```
>>> from fibo import fib, fib2
>>> fib(500)
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 (entonces, en el ejemplo, `fibo` no se define).

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

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

Esto importa todos los nombres excepto aquellos que comienzan con un subrayado (`_`). La mayoría de las veces los programadores de Python no usan esto ya que introduce un conjunto de nombres en el intérprete, posiblemente escondiendo cosas que ya estaban definidas.

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

Ejecutando módulos como scripts

Cuando ejecutas un módulo de Python con

```
python fibo.py <argumentos>
```

...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
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).

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 conteniendo el script (o el directorio actual cuando no se especifica un archivo).

- **PYTHONPATH** (una lista de nombres de directorios, con la misma sintaxis que la variable de entorno **PATH**.
- el directorio default de la instalación.

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.

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 version 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 evr 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 librería 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.
Algunos consejos para expertos:

- Podés usar la *-O* o *-OO* en el comando de Python para reducir el tamaño de los módulos compilados. La `-O` quita assert statements, la `--O` quita ambos, assert statements y cadenas de caracteres __doc__. Debido a que algunos programas se basan en que estos estén disponibles, deberías usar esta opción únicamente si sabés lo que estás haciendo. Los módulos "optimizados" tienen una etiqueta `opt-` y son normalmente más pequeños. Releases futuras quizás cambien los efectos de la optimización.

- Un programa no corre más rápido cuando se lee de un archivo *.pyc* que cuando se lee del *.py*; lo único que es más rápido en los archivos *.pyc* es la velocidad con que se cargan.
- Hay más detalles de este proceso, incluyendo un diagrama de flujo de la toma de decisiones, en la PEP 3147.
- El módulo **`compilleall`** puede crear archivos .pyc para todos los módulos en un directorio.

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 podés modificar usando las operaciones estándar de listas:

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

La función dir()

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

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'argv',
 'base_exec_prefix', 'base_prefix', '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',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
 'thread_info', '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']
```

Nota 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']
```

Paquetes

Los paquetes son una manera de estructurar los espacios de nombres de Python usando "nombres de módulos con puntos". Por ejemplo, el nombre de módulo **`A.B`** designa un submódulo llamado `B` en un paquete llamado `A`. Tal como el uso de módulos evita que los autores de diferentes módulos tengan que preocuparse de los respectivos nombres de variables globales, el uso de nombres de módulos con puntos evita que los autores de paquetes de muchos módulos, como NumPy o la Biblioteca de Imágenes de Python (Python Imaging Library, o PIL), tengan que preocuparse de los respectivos nombres de módulos.

Supón que se quiere 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/                          Paquete superior
      __init__.py               Inicializa el paquete de sonido
      formats/                  Subpaquete para conversiones de formato
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
```

```
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                Subpaquete para efectos de sonido
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                Subpaquete para filtros
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

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

Los archivos `__init__.py` se necesitan para hacer que Python trate los directorios como que contienen paquetes; esto se hace para prevenir directorios con un nombre común, como `string`, de esconder sin intención a módulos válidos que se suceden luego en el camino de búsqueda de módulos. En el caso más simple, `__init__.py` puede ser solamente un archivo vacío, pero también puede ejecutar código de inicialización para el paquete o configurar la variable `__all__`, descrita 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 submódulos es:

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

Esto también carga el submódulo **echo**, 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)
```

Nota 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.

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. Considerá 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 usás `import *`, también se considera una mala práctica en código de producción.

Recordá que no está mal usar `from paquete import submodulo_especifico`! De hecho, esta notación se recomienda a menos que el módulo que estás importando necesite usar submódulos con el mismo nombre desde otros paquetes.

Referencias internas en paquetes

Cuando se estructuran los paquetes en subpaquetes (como en el ejemplo **sound**), se puede 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 se puede escribir `import` relativos con la forma `from module import name`. Estos imports usan puntos adelante para indicar los paquetes actuales o padres involucrados en el import relativo. En el ejemplo **`surround`**, podrías hacer:

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

Nota 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.

Paquetes en múltiples directorios

Los paquetes soportan un atributo especial más, **`__path__`**. Este se inicializa, antes de que el código en ese archivo se ejecute, a una lista que contiene el nombre del directorio donde está el paquete. 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.

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.

Formateo elegante de la salida

Hasta ahora encontramos dos maneras de escribir valores: *declaraciones de expresión* y la función **`print()`**. (Una tercer manera es usando el método **`write()`** de los objetos tipo archivo; el archivo de salida estándar puede referenciarse como `sys.stdout`.

Frecuentemente querrás más control sobre el formateo de tu salida que simplemente imprimir valores separados por espacios. Hay dos maneras de formatear tu salida; la primera es hacer todo el manejo de las cadenas vos mismo: usando rebanado de cadenas y operaciones de concatenado se puede crear cualquier forma que puedas imaginar. El tipo *string* contiene algunos métodos útiles para emparejar cadenas a un determinado ancho; estas las discutiremos en breve. La otra forma es usar *formatted string literals* o el método **`str.format()`**.

El módulo **`string`** contiene una clase **`string.Template`** que ofrece otra forma de sustituir valores en las cadenas.

Nos queda una pregunta, por supuesto: ¿cómo convertís valores a cadenas? Afortunadamente, Python tiene maneras de convertir cualquier valor a una cadena: pásalos a las funciones **`repr()`** o **`str()`**.

La función **`str()`** devuelve 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 sintáxis equivalente). Para objetos que no tienen una representación en particular para consumo humano, **`str()`** devolverá 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 = 'Hola mundo.'
>>> str(s)
'Hola mundo.'
```

```
>>> repr(s)
"'Hola mundo.'"
>>> str(1 / 7)
'0.142857142857'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'El valor de x es ' + repr(x) + ', y es ' + repr(y) + '...'
>>> print(s)
El valor de x es 32.5, y es 40000...
>>> # El repr() de una cadena agrega apóstrofos y barras invertidas
... hola = 'hola mundo\n'
>>> holas = repr(hola)
>>> print(holas)
'hola mundo\n'
>>> # El argumento de repr() puede ser cualquier objeto Python:
... repr((x, y, ('carne', 'huevos')))
"(32.5, 40000, ('carne', 'huevos'))"
```

Acá hay dos maneras de escribir una tabla de cuadrados y cubos:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x * x).rjust(3), end=' ')
...     # notar el uso de 'end' en la linea anterior
...     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
```

```
>>> 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
```

(Notar que en el primer ejemplo, un espacio entre cada columna fue agregado por la manera en que **`print()`** trabaja: siempre agrega espacios entre sus argumentos)

Este ejemplo muestra el método **`str.rjust()`** de los objetos cadena, el cual ordena una cadena a la derecha en un campo del ancho dado llenándolo con espacios a la izquierda. Hay métodos similares **`str.ljust()`** y **`str.center()`**. Estos métodos no escriben nada, sólo devuelven una nueva cadena. Si la cadena de entrada es demasiado larga, no la truncan, sino la devuelven intacta; esto te romperá la alineación de tus columnas pero es normalmente mejor que la alternativa, que te estaría mintiendo sobre el valor. (Si realmente querés que se recorte, siempre podés agregarle 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'
```

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

```
>>> print('Somos los {} quienes decimos "{}!"'.format('caballeros', 'Nop'))
Somos los caballeros quienes decimos "Nop!"
```

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.

```
>>> print('{0} y {1}'.format('carne', 'huevos'))
carne y huevos
>>> print('{1} y {0}'.format('carne', 'huevos'))
huevos y carne
```

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

```
>>> print('Esta {comida} es {adjetivo}.'.format(
...       comida='carne', adjetivo='espantosa'))
Esta carne es espantosa.
```

Se pueden combinar arbitrariamente argumentos posicionales y nombrados:

```
>>> print('La historia de {0}, {1}, y {otro}.'.format('Bill', 'Manfred',
...                                                   otro='Georg'))
La historia de Bill, Manfred, y Georg.
```

Se pueden usar `'!a'` (aplica **`apply()`**), `'!s'` (aplica **`str()`**) y `'!r'` (aplica **`repr()`**) para convertir el valor antes de que se formatee.

```
>>> contents = 'anguilas'
>>> print('Mi aerodeslizador esta lleno de {}.'.format(contents))
Mi aerodeslizador esta lleno de anguilas.
>>> print('My hovercraft is full of {!r}.'.format(contents))
Mi aerodeslizador esta lleno de 'anguilas'.
```

Un `:` y especificador de formatos opcionales pueden ir luego del nombre del campo. Esto aumenta el control sobre cómo el valor es formateado. El siguiente ejemplo redondea Pi a tres lugares luego del punto decimal.

```
>>> import math
>>> print('El valor de PI es aproximadamente {0:.3f}.'.format(math.pi))
El valor de PI es aproximadamente 3.142.
```

Pasando un entero luego del `':'` causará que el campo sea de un mínimo número de caracteres de ancho. Esto es útil para hacer tablas lindas.

```
>>> tabla = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for nombre, telefono in tabla.items():
...     print('{0:10} ==> {1:10d}'.format(nombre, telefono))
...
Dcab       ==>       7678
Jack       ==>       4098
Sjoerd     ==>       4127
```

Si tienes una cadena de formateo realmente larga que no querrás separar, podría ser bueno que puedas hacer referencia a las variables a ser formateadas por el nombre en vez de la posición. Esto puede hacerse simplemente pasando el diccionario y usando corchetes `'[]'` para acceder a las claves

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

Esto se podría también hacer pasando la tabla como argumentos nombrados con la notación '**'.

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

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

Viejo formateo de cadenas

El operador `%` también puede usarse para formateo de cadenas. Interpreta el argumento de la izquierda con el estilo de formateo de **`sprintf()`** para ser aplicado al argumento de la derecha, y devuelve la cadena resultante de esta operación de formateo. Por ejemplo:

```
>>> import math
>>> print('El valor de PI es aproximadamente %5.3f.' % math.pi)
El valor de PI es aproximadamente 3.142.
```

Leyendo y escribiendo archivos

La función **`open()`** devuelve un *objeto archivo*, y se usa normalmente con dos argumentos: `open(nombre_de_archivo, modo)`.

```
>>> f = open('archivodetrabajo', 'w')
>>> print(f)
<_io.TextIOWrapper name='archivodetrabajo' mode='w' encoding='UTF-8'>
```

El primer argumento es una cadena conteniendo el nombre de archivo. El segundo argumento es otra cadena conteniendo unos pocos caracteres que describen la forma en que el archivo será usado. El *modo* puede ser `'r'` cuando el archivo será solamente leído, `'w'` para sólo escribirlo (un archivo existente con el mismo nombre será borrado), y `'a'` abre el archivo para agregarle información; cualquier dato escrito al archivo será automáticamente agregado al final. `'r+'` abre el archivo tanto para leerlo como para escribirlo. El argumento *modo* es opcional; si se omite se asume `'r'`.

Normalmente los archivos se abren en *modo texto*, lo que significa que puedes leer y escribir cadenas del y al archivo, las cuales se codifican utilizando un código específico. Si el código no es especificado, el valor predeterminado depende de la plataforma. Si se agrega `b` al modo el archivo se abre en *modo binario*: ahora los datos se leen y escriben en forma de objetos bytes. Se debería usar este modo para todos los archivos que no contengan texto.

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

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 llamá a `f.read(cantidad)`, el cual lee alguna cantidad de datos y los devuelve como una cadena de (en modo texto) o un objeto de bytes (en modo binario). *cantidad* es un argumento numérico opcional. Cuando se omite *cantidad* o es negativo, el contenido entero del archivo será leido y devuelto; es tu problema si el archivo es el doble de grande que la memoria de tu máquina. De otra manera, a lo sumo una *cantidad* de bytes son leídos y devueltos. Si se alcanzó el fin del archivo, `f.read()` devolverá una cadena vacía (`""`).

```
>>> f.read()
'Este es el archivo entero.\n'
>>> f.read()
''
```

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

```
>>> f.readline()
'Esta es la primer linea del archivo.\n'
>>> f.readline()
'Segunda linea del archivo\n'
>>> f.readline()
''
```

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

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

Esta es la primer linea del archivo
Segunda linea del archivo
```

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

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

```
>>> f.write('Esto es una prueba\n')
19
```

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:

```
>>> valor = ('la respuesta', 42)
>>> s = str(valor)  # Convierte la tupla a string
>>> f.write(s)
18
```

`f.tell()` devuelve 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, usá `f.seek(desplazamiento, desde_donde)`. La posición es calculada agregando el *desplazamiento* a un punto de referencia; el punto de referencia se selecciona del argumento *desde_donde*. Un valor *desde_donde* 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. *desde_donde* puede omitirse, el default es 0, usando el comienzo del archivo como punto de referencia.

```
>>> f = open('archivodetrabajo', 'rb+')
>>> f.write(b'0123456789abcdef')
>>> f.seek(5)     # Va al sexto byte en el archivo
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Va al tercer byte antes del final
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.

Cuando hayas terminado con un archivo, llamá a `f.close()` para cerrarlo y liberar cualquier recurso del sistema tomado por el archivo abierto. Luego de llamar `f.close()`, los intentos de usar el objeto archivo fallarán automáticamente.

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

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('archivodetrabajo', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

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

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 devuelve cadenas, que tendrán que ser pasadas a una función como **`int()`**, que toma una cadena como `'123'` y devuelve su valor numérico 123. Sin embargo, cuando quieres 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 estandar 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 guardada en un archivo o datos, o enviado a una máquina distante por una conexión de red.

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

```
>>> json.dumps([1, 'simple', 'lista'])
'[1, "simple", "lista"]'
```

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(x, 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.

Errores y excepciones

Hasta ahora los mensajes de error no habían sido más que mencionados, pero si probaste los ejemplos probablemente hayas visto algunos. Hay (al menos) dos tipos diferentes de errores: *errores de sintaxis* y *excepciones*.

Errores de sintaxis

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

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

El intérprete repite la línea culpable y muestra una pequeña 'flecha' que apunta al primer lugar donde se detectó el error. Este es causado por (o al menos detectado en) el símbolo 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.

Excepciones

Incluso si la declaración o expresión es sintácticamente correcta, puede generar un error cuando se intenta ejecutarla. Los errores detectados durante la ejecución se llaman *excepciones*, y no son incondicionalmente fatales: pronto aprenderás cómo manejarlos

en los programas en Python. Sin embargo, la mayoría de las excepciones no son manejadas por los programas, 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't convert 'int' object to str implicitly
```

La última línea de los mensajes de error indica qué sucedió. Las excepciones vienen de distintos 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 ocurrió. Esto es verdad para todas las excepciones predefinidas del intérprete, pero no necesita ser verdad 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 un detalle basado en el tipo de la excepción y qué la causó.
La parte anterior del mensaje de error muestra el contexto donde la excepción sucedió, en la forma de un *trazado del error* listando líneas fuente; sin embargo, no mostrará líneas leídas desde la entrada estándar.

Manejando excepciones

Es posible escribir programas que manejen determinadas excepciones. Mira 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 sea que el sistema operativo soporte); nota que una interrupción generada por el usuario se señaliza generando la excepción **`KeyboardInterrupt`**.

```
>>> while True:
...     try:
...         x = int(input("Por favor ingrese un número: "))
...         break
...     except ValueError:
...         print("Oops! No era válido. Intente nuevamente...")
...
```

La declaración `try` funciona de la siguiente manera:

- Primero, se ejecuta el *bloque try* (el código entre las declaración `try` y **except**).
- Si no ocurre ninguna excepción, el *bloque except* se saltea y termina la ejecución de la declaración `try`.
- Si ocurre una excepción durante la ejecución del *bloque try*, el resto del bloque se saltea. Luego, si su tipo coincide con la excepción nombrada luego de la palabra reservada **except**, se ejecuta el *bloque except*, y la ejecución continúa luego de la declaración `try`.
- Si ocurre una excepción que no coincide con la excepción nombrada en el `except`, esta se pasa a declaraciones `try` de más afuera; si no se encuentra nada que la maneje, es una *excepción no manejada*, y la ejecución se frena con un mensaje como los mostrados arriba.

Una declaración `try` puede tener más de un `except`, para especificar manejadores para distintas excepciones. A lo sumo un manejador será ejecutado. Sólo se manejan excepciones que ocurren en el correspondiente `try`, no en otros manejadores del mismo `try`.

Un `except` puede nombrar múltiples excepciones usando paréntesis, por ejemplo:

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

Una clase en una clausula `except` es compatible con una excepción si la misma esta en la misma clase o una clase base de la misma (pero no de la otra manera — una clausula except listando una clase derivada no es compatible con una clase base). Por ejemplo, el siguiente código imprimirá B, C, 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 clausulas de except estuvieran invertidas (con `except B` primero), habría impreso B, B, B — la primera clausula de except coincidente es usada.

El último **`except`** puede omitir nombrar qué excepción captura, para servir como comodín. Usa esto con extremo cuidado, ya que de esta manera es fácil ocultar un error real de programación. También puede usarse para mostrar un mensaje de error y luego re-generar la excepción (permitiéndole al que llama, manejar también la excepción):

```
import sys

try:
    f = open('miarchivo.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("Error OS: {0}".format(err))
except ValueError:
```

```
    print("No pude convertir el dato a un entero.")
except:
    print("Error inesperado:", sys.exc_info()[0])
    raise
```

Las declaraciones **`try`** ... **`except`** tienen un *bloque else* opcional, el cual, cuando está presente, debe seguir a los except. Es útil para aquel código que debe ejecutarse si el *bloque try* no genera una excepción. Por ejemplo:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('no pude abrir', arg)
    else:
        print(arg, 'tiene', len(f.readlines()), 'lineas')
        f.close()
```

El uso de **`else`** es mejor que agregar código adicional en el **`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 dependen del tipo de excepción.

El **`except`** puede especificar una variable luego del nombre de excepción. La variable se vincula a una instancia de excepción con los argumentos almacenados en `instance.args`. Por conveniencia, la instancia de excepción define **`__str__()`** para que se pueda mostrar los argumentos directamente, sin necesidad de hacer referencia a `.args`. También se puede instanciar la excepción primero, antes de generarla, y agregarle los atributos que se desee:

```
>>> try:
...     raise Exception('carne', 'huevos')
... except Exception as inst:
...     print(type(inst))    # la instancia de excepción
...     print(inst.args)     # argumentos guardados en .args
```

```
...     print(inst)          # __str__ permite imprimir args directamente,
...                          # pero puede ser cambiado en subclases de la exc
...     x, y = inst          # desempacar argumentos
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('carne', 'huevos')
('carne', 'huevos')
x = carne
y = huevos
```

Si una excepción tiene argumentos, estos se imprimen como la última parte (el 'detalle') del mensaje para las excepciones que no están manejadas.

Los manejadores de excepciones no manejan solamente las excepciones que ocurren en el *bloque try*, también manejan las excepciones que ocurren dentro de las funciones que se llaman (inclusive indirectamente) dentro del *bloque try*. Por ejemplo:

```
>>> def esto_falla():
...     x = 1/0
...
>>> try:
...     esto_falla()
... except ZeroDivisionError as err:
...     print('Manejando error en tiempo de ejecución:', err)
...
Manejando error en tiempo de ejecución: division by zero
```

Levantando excepciones

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

```
>>> raise NameError('Hola')
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
NameError: Hola
```

El único argumento a **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 sera instanciada implicitamente llamandoa su constructor sin argumentos:

```
raise ValueError  # atajo para 'raise ValueError()'
```

Si necesitas determinar cuando una excepción fue lanzada pero no quieres manejarla, una forma simplificada de la instrucción **raise** te permite relanzarla:

```
>>> try:
...     raise NameError('Hola')
... except NameError:
...     print('Voló una excepción!')
...     raise
...
Voló una excepción!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: Hola
```

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 Excepciones pueden ser definidas de la misma forma que cualquier otra clase, pero usualmente se mantienen simples, a menudo solo ofreciendo un número de atributos con información sobre el error que leerán los manejadores 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):
    """Clase base para excepciones en el módulo."""
    pass

class EntradaError(Error):
    """Excepción lanzada por errores en las entradas.

    Atributos:
        expresion -- expresión de entrada en la que ocurre el error
        mensaje -- explicación del error
    """

    def __init__(self, expresion, mensaje):
        self.expresion = expresion
        self.mensaje = mensaje

class TransicionError(Error):
    """Lanzada cuando una operacion intenta una transicion de estado no
    permitida.

    Atributos:
        previo -- estado al principio de la transición
        siguiente -- nuevo estado intentado
        mensaje -- explicación de por qué la transición no está permitida
    """
    def __init__(self, previo, siguiente, mensaje):
        self.previo = previo
        self.siguiente = siguiente
        self.mensaje = mensaje
```

La mayoría de las excepciones son definidas con nombres que terminan en “Error”, similares a los nombres de las excepciones estándar.

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

Definiendo acciones de limpieza

La declaración **try** tiene otra cláusula opcional que intenta definir acciones de limpieza que deben ser ejecutadas bajo ciertas circunstancias. Por ejemplo:

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

Una *cláusula finally* siempre es ejecutada antes de salir de la declaración **try**, ya sea que una excepción haya ocurrido o no. Cuando ocurre una excepción en la cláusula **try** y no fue manejada por una cláusula **except** (o ocurrió en una cláusula **except** o **else**), es relanzada luego de que se ejecuta la cláusula **finally**. El **finally** es también ejecutado “a la salida” cuando cualquier otra cláusula de la declaración **try** es dejada vía **break**, **continue** or **return**. Un ejemplo más complicado:

```
>>> def dividir(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("¡división por cero!")
...     else:
...         print("el resultado es", result)
...     finally:
...         print("ejecutando la clausula finally")
...
>>> dividir(2, 1)
el resultado es 2.0
ejecutando la clausula finally
```

```
>>> dividir(2, 0)
¡división por cero!
ejecutando la clausula finally
>>> divide("2", "1")
ejecutando la clausula finally
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 puedes ver, la cláusula **`finally`** es ejecutada siempre. La excepción **`TypeError`** lanzada al dividir dos cadenas de texto no es manejado 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.

Acciones predefinidas de limpieza

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

```
for linea in open("miarchivo.txt"):
    print(linea, 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 objetos como archivos sean usados de una forma que asegure que siempre se los libera rápido y en forma correcta:

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

Luego de que la declaración sea ejecutada, el archivo *f* siempre es cerrado, incluso si se encuentra un problema al procesar las líneas. Objetos que, como los archivos, provean acciones de limpieza predefinidas lo indicarán en su documentación.